MixtureLib

Nov 13, 2020

Main Info:

CHAPTER 1

Mixture Lib

1.1 Basic information

Implementation code for a mixture of models. The source code for the problem of the mixture of models and the task of the mixture of experts is presented.

All methods were implemented based on `pytorch` for simple parallelization by using `cuda`.

All information about this project can be found in the documentation.

1.2 Requirements and Installation

A simple instruction of installation using pip is provided near the source code.

More information about installation can be found in documentation installation page.

1.3 Example of use

A simple examples of module usage can be found in documentation example page.

CHAPTER 2

Installation

2.1 Requirements

- Python 3.6.2
- pip 20.0.2

2.2 Installing by using PyPi

2.2.1 Install

```
python3.6 -m pip install mixturelib
```

2.2.2 Uninstall

```
python3.6 -m pip uninstall mixturelib
```

2.3 Installing from GitHub source

2.3.1 Install

```
git clone https://github.com/andriygav/MixtureLib.git
cd MixtureLib
python3.6 -m pip install ./src/
```

2.3.2 Uninstall

```
python3.6 -m pip uninstall mixturelib
```

CHAPTER 3

Example

3.1 Requirements

It is recommended make virtualenv and install all next packages in this virtualenv.

```
torch==1.4.0
numpy==1.18.1
matplotlib==2.2.4
mixturelib==0.2.*
```

Include packages.

```
import torch
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from mixturelib.mixture import MixtureEM
from mixturelib.local_models import EachModelLinear
from mixturelib.hyper_models import HyperExpertNN, HyperModelDirichlet
```

3.2 Preparing the dataset

Generate dataset. This dataset contains two different planes.

```
np.random.seed(42)

N = 200
noise_component = 0.8
noise_target = 5
```

(continues on next page)

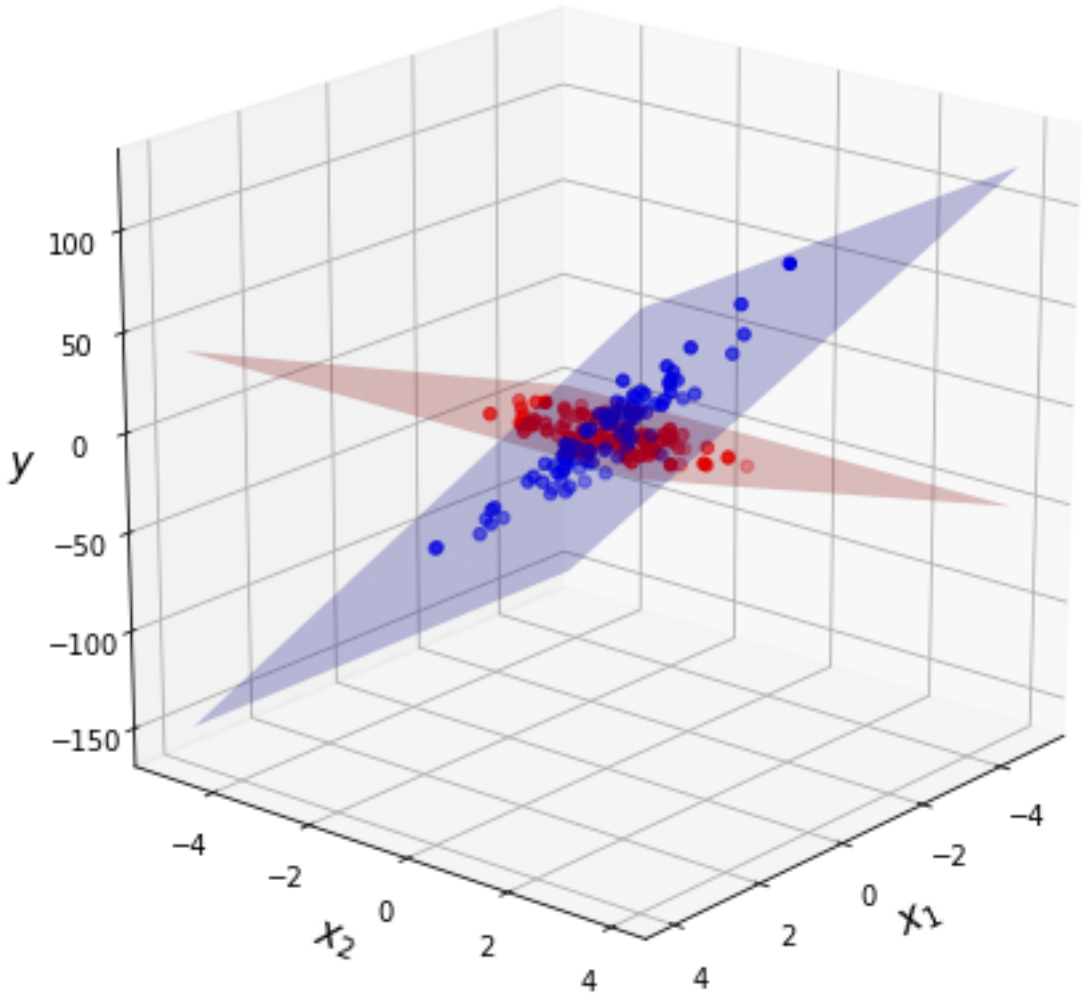
(continued from previous page)

```
X = np.random.randn(N, 2)
X[:N//2, 1] *= noise_component
X[N//2:, 0] *= noise_component

real_first_w = np.array([[10.], [0.]])
real_second_w = np.array([[0.], [30.]])

y = np.vstack([X[:N//2]@real_first_w, X[N//2:]@real_second_w])\
    + noise_target*np.random.randn(N, 1)
```

Dataset visualisation. Blue color corresponds to one local model and red corresponds to another.



Convert the dataset into `torch.tensor` format.

```
torch.random.manual_seed(42)
X_tr = torch.FloatTensor(X)
Y_tr = torch.FloatTensor(y)
```

3.3 Mixture of Model

Consider an example of a mixture of model. In this case the contribution of each model does not depend on the sample from dataset.

Init local models. We consider linear model *mixturelib.local_models.EachModelLinear* as local model.

```
torch.random.manual_seed(42)
first_model = EachModelLinear(input_dim=2)
secode_model = EachModelLinear(input_dim=2)

list_of_models = [first_model, secode_model]
```

Init hyper model — *mixturelib.hyper_models.HyperModelDirichlet*. In this case contribution of each model does not depend on sample from dataset. It is suggested that the contribution of each model has a Dirichlet distribution. The hyper model parameters is a parameter of Dirichlet distribution.

```
HpMd = HyperModelDirichlet(output_dim=2)
```

Init mixture model. The mixture model is simple function which weighs local models answers. Weights are generated by hyper model *HpMd*.

```
mixture = MixtureEM(HyperParameters={'beta': 1.},
                    HyperModel=HpMd,
                    ListOfModels=list_of_models,
                    model_type='sample')
```

Train mixture model on the give dataset.

```
mixture.fit(X_tr, Y_tr)
```

Local models parameters after training procedure. In our task, each model is a simple plane in 3D space.

```
predicted_first_w = mixture.ListOfModels[0].W.numpy()
predicted_second_w = mixture.ListOfModels[1].W.numpy()
```

Visualization of the real and predicted planes on the chart.

```
fig = plt.figure(figsize=(8, 8))

ax = fig.add_subplot(111, projection='3d')

grid_2d = np.array(np.meshgrid(range(-5, 5), range(-5, 5)))
xx, yy = grid_2d

first_z = (predicted_first_w.reshape([-1, 1, 1])*grid_2d).sum(axis=0)
second_z = (predicted_second_w.reshape([-1, 1, 1])*grid_2d).sum(axis=0)
ax.plot_surface(xx, yy, first_z, alpha = 0.25, color = 'red', label='predicted')
ax.plot_surface(xx, yy, second_z, alpha = 0.25, color = 'blue', label='predicted')
```

(continues on next page)

(continued from previous page)

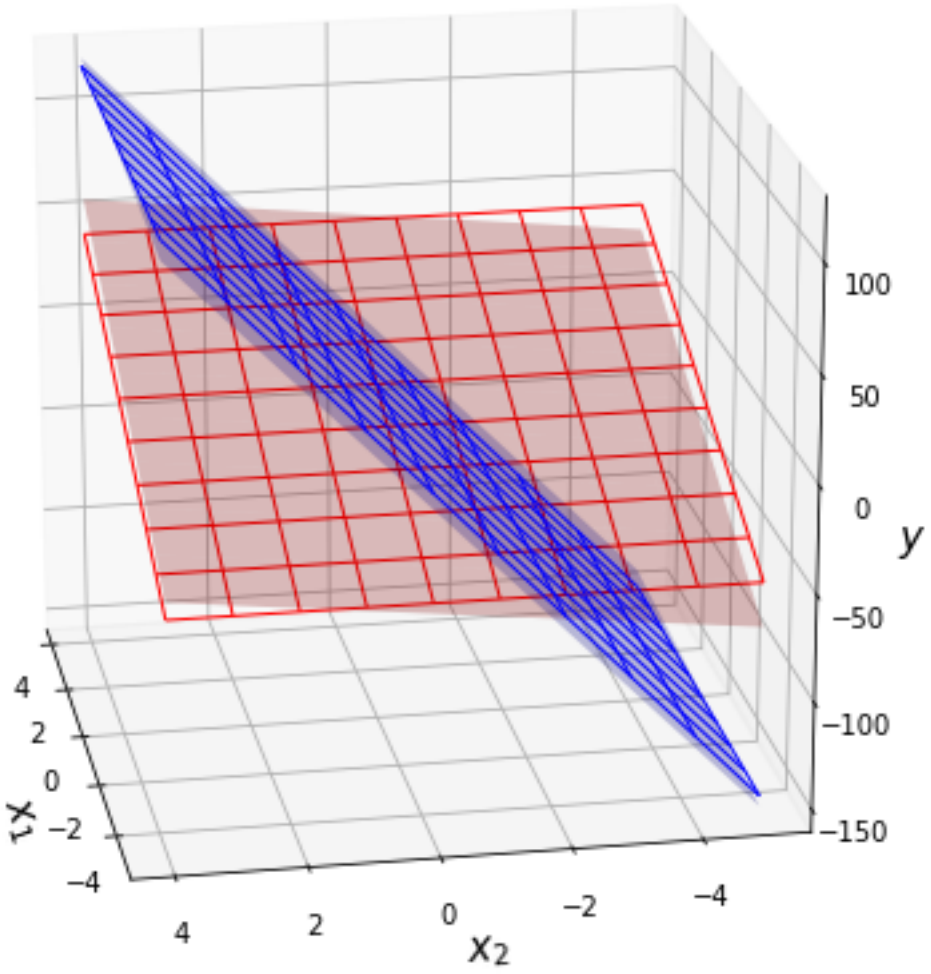
```
first_z = (real_first_w.reshape([-1, 1, 1])*grid_2d).sum(axis=0)
second_z = (real_second_w.reshape([-1, 1, 1])*grid_2d).sum(axis=0)
ax.plot_wireframe(xx, yy, first_z, linewidth=1, color = 'red')
ax.plot_wireframe(xx, yy, second_z, linewidth=1, color = 'blue')

ax.view_init(20, 170)

ax.set_xlabel('$x_1$', fontsize=15, fontweight="bold")
ax.set_ylabel('$x_2$', fontsize=15, fontweight="bold")
ax.set_zlabel('$y$', fontsize=15, fontweight="bold")

plt.show()
```

The surfaces with grid are real planes. The surfaces without grid are predicted planes.



3.4 Mixture of Experts

Now consider an example of a mixture of experts on the same dataset. In this case contribution of each model is depend on sample from dataset.

Init local models. We consider linear model *mixturelib.local_models.EachModelLinear* as local model.

```
torch.random.manual_seed(42)
first_model = EachModelLinear(input_dim=2)
secode_model = EachModelLinear(input_dim=2)

list_of_models = [first_model, secode_model]
```

Init hyper model — gate function *mixturelib.hyper_models.HyperExpertNN*. In this case contribution of each model is depend on sample from dataset. Gate function is a simple neural network with softmax on the last layer.

```
HpMd = HyperExpertNN(input_dim=2, hidden_dim=5,
                     output_dim=2, epochs=100)
```

Init mixture model. The mixture model is simple function which weighs local models answers. Weights are generated by hyper model *HpMd*.

```
mixture = MixtureEM(HyperParameters={'beta': 1.},
                    HyperModel=HpMd,
                    ListOfModels=list_of_models,
                    model_type='sample')
```

Train mixture model on the give dataset.

```
mixture.fit(X_tr, Y_tr)
```

Local models parameters after training procedure. In our task, each model is a simple plane in 3D space.

```
predicted_first_w = mixture.ListOfModels[0].W.numpy()
predicted_second_w = mixture.ListOfModels[1].W.numpy()
```

Visualization of the real and predicted planes on the chart.

```
fig = plt.figure(figsize=(8, 8))

ax = fig.add_subplot(111, projection='3d')

grid_2d = np.array(np.meshgrid(range(-5, 5), range(-5, 5)))
xx, yy = grid_2d

first_z = (predicted_first_w.reshape([-1, 1, 1])*grid_2d).sum(axis=0)
second_z = (predicted_second_w.reshape([-1, 1, 1])*grid_2d).sum(axis=0)
ax.plot_surface(xx, yy, first_z, alpha = 0.25, color = 'red', label='predicted')
ax.plot_surface(xx, yy, second_z, alpha = 0.25, color = 'blue', label='predicted')

first_z = (real_first_w.reshape([-1, 1, 1])*grid_2d).sum(axis=0)
second_z = (real_second_w.reshape([-1, 1, 1])*grid_2d).sum(axis=0)
ax.plot_wireframe(xx, yy, first_z, linewidth=1, color = 'red')
ax.plot_wireframe(xx, yy, second_z, linewidth=1, color = 'blue')

ax.view_init(20, 170)
```

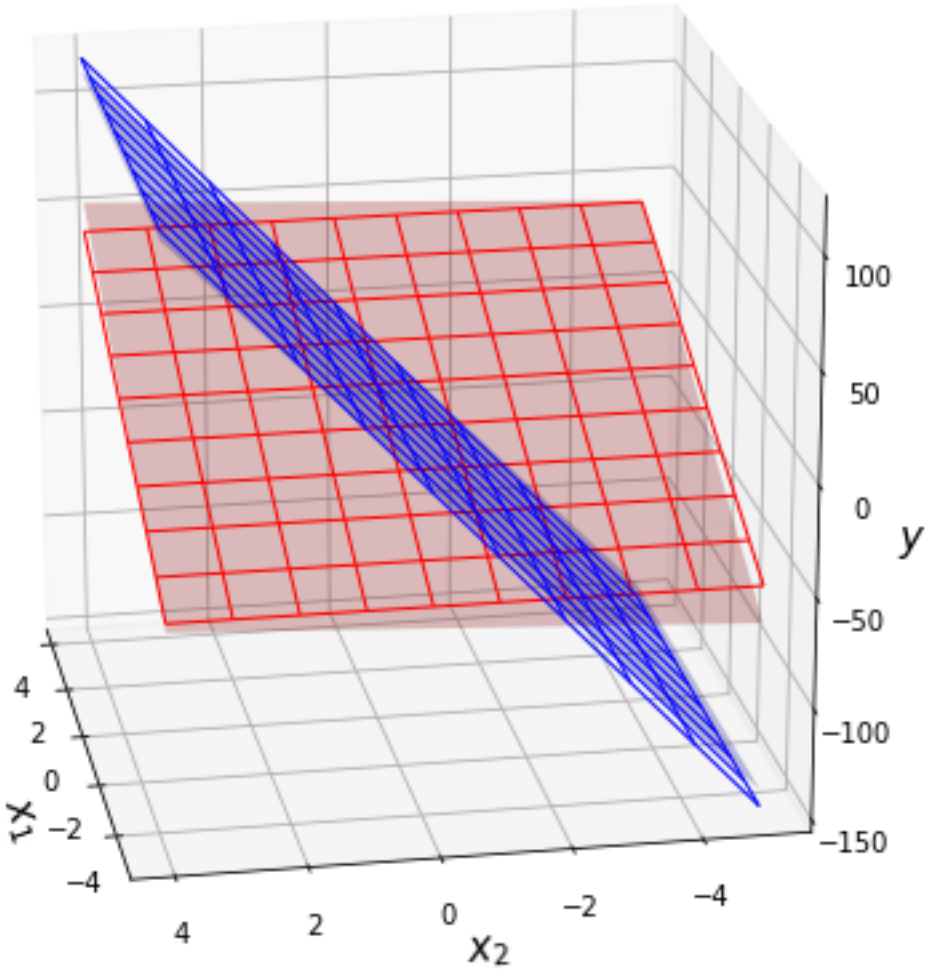
(continues on next page)

(continued from previous page)

```
ax.set_xlabel('$x_1$', fontsize=15, fontweight="bold")
ax.set_ylabel('$x_2$', fontsize=15, fontweight="bold")
ax.set_zlabel('$y$', fontsize=15, fontweight="bold")

plt.show()
```

The surfaces with grid are real planes. The surfaces without grid are predicted planes.



CHAPTER 4

Local Models

The *mixturelib.local_models* contains classes:

- *mixturelib.local_models.EachModel*
- *mixturelib.local_models.EachModelLinear*

class mixturelib.local_models.**EachModel**
Base class for all local models.

E_step(*X*, *Y*, *Z*, *HyperParameters*)
Doing E-step of EM-algorithm. Finds variational probability *q* of model parameters.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *1*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

LogLikeLihoodExpectation(*X*, *Y*, *HyperParameters*)
Returns expected log-likelihod of a given vector of answers for given data seperated. The expectation is taken according to the model parameters $\mathsf{E}p\big(Y|X\big)$.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

Returns The tensor of shape *num_elements* × *1*.

Return type FloatTensor

M_step(*X*, *Y*, *Z*, *HyperParameters*)
Doing M-step of EM-algorithm. Finds model hyper parameters.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *1*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

OptimizeHyperParameters(*X*, *Y*, *Z*, *HyperParameters*, *Parameter*)
Returns the local part of new Parameter.

> **Warning:** Returned local part must be aditive to Parameter, because new value of Parameter is sum of local part from all local model.

> **Warning:** The number *num_answers* can be just *1*.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *1*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.
- **Parameter** (*string*) – The key from *HyperParameters* dictionary. The name of the hyperparameter to be optimized.

Returns A local part of new *HyperParameters[Parameter]* value.

Return type FloatTensor

forward(*input*)
Returns model prediction for the given input data.

Parameters **input** (*FloatTensor.*) – The tensor of shape *num_elements* × *num_feature*.

Returns The tensor of shape *num_elements* × *num_answers*. Model answers for the given input data

Return type FloatTensor

class mixturelib.local_models.**EachModelLinear**(*input_dim=20*, *device='cpu'*, *A=None*, *w=None*, *OptimizedHyper={'A'*, *'beta'*, *'w_0'}*)
A model for solving the linear regression problem $\mathbf{y} = \mathbf{x}^\top \mathbf{w}$. The model uses an analytical solution for estimation $\mathbf{w}$. Also model finds a distribution of model parameters $\mathbf{w}$.

Distribution is $\mathbf{w} \sim \mathcal{N}\left(\hat{\mathbf{w}}, \mathbf{A}\right)$

Warning: The priors *A* and *w* must be set or not together.

Parameters

- **input_dim** (*int*) – The number of feature in each object. Must be positive.
- **device** (*string*) – The device for pytorch. Can be 'cpu' or 'gpu'. Default 'cpu'.
- **A** (*FloatTensor*) – The tensor of shape input_dim × input_dim. It is a prior covariance matrix for model parameters. Also can be the tensor of shape input_dim. In this case, it is a diagonal of prior covariance matrix for model parameters, and all nondiagonal values are zerous.
- **w** (*FloatTensor*) – The tensor of shape input_dim. It is a prior mean for model parameters
- **OptimizedHyper** (*set*) – The set of hyperparameters that will be optimized. Default all hyperparameters will be optimized.

Example:

```
>>> _ = torch.random.manual_seed(42) # Set random seed for repeatability
>>>
>>> w = torch.randn(2, 1) # Generate real parameter vector
>>> X = torch.randn(10, 2) # Generate features data
>>> Z = torch.ones(10, 1) # Set that all data correspond to this model
>>> Y = X@w + 0.1*torch.randn(10, 1) # Generate target data with noise 0.1
>>>
>>> linear_model = EachModelLinear(
...     input_dim=2) # Init linear model withot any priors
>>> hyper_parameters = {
...     'beta': torch.tensor(1.)} # Init noise level beta
>>>
>>> linear_model.E_step(
...     X[:8], Y[:8], Z[:8], hyper_parameters) # Optimise model parameter
>>>
>>> linear_model(X[8:]).view(-1) # Model prediction for the test part
tensor([-0.1975, -0.2427])
>>> Y[8:].view(-1) # Real target for test part
tensor([-0.2124, -0.1837])
```

B = None
Object **B** is a covariance matrix for variational distribution

E_step (*X*, *Y*, *Z*, *HyperParameters*)
Doing E-step of EM-algorithm. Finds variational probability q of model parameters.

Calculate analytical solution for estimate q in the class of normal distributions $q = \mathcal{N}(m, B)$, where $B = (A^{-1} + \beta \sum_{i=1} X_i X_i^{\mathsf{T}} \mathsf{E} z_i)$ and $m = B(A^{-1} w_0 + \beta \sum_{i=1} X_i Y_i \mathsf{E} z_i)$

Warning: HyperParameters must contain *beta* hyperparameter.

Warning: The number *num_answers* can be just *1*.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *1*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

LogLikeLihoodExpectation (*X*, *Y*, *HyperParameters*)

Returns expected log-likelihod of a given vector of answers for given data seperated. The expectation is taken according to the model parameters $\mathsf{E}p\big(Y|X\big)$.

Warning: HyperParameters must contain *beta* hyperparameter.

Warning: The number *num_answers* can be just *1*.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *num_answers*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

Returns The tensor of shape *num_elements* $\times$ *1*.

Return type FloatTensor

M_step (*X*, *Y*, *Z*, *HyperParameters*)

Doing M-step of EM-algorithm. Finds model hyper parameters.

Warning: HyperParameters must contain *beta* hyperparameter.

Warning: The number *num_answers* can be just *1*.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *1*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

OptimizeHyperParameters (*X*, *Y*, *Z*, *HyperParameters*, *Parameter*)

Returns the local part of new Parameter.

In this case *local_part* is inverse beta: $\frac{1}{\beta} = \frac{1}{num\_elements} \sum_{i=1}^{num\_elements} [Y_i^2 - 2Y_i X_i^{\mathsf{T}} \hat{\mathbf{w}} + X_i^{\mathsf{T}} \mathsf{E}[ww^{\mathsf{T}}] X_i]$

Warning: Return local part must be aditive to Parameter, because new value of Parameter is sum of local part from all local model.

Warning: HyperParameters must contain *beta* hyperparameter.

Warning: The number *num_answers* can be just *1*.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *1*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.
- **Parameter** (*string*) – The key from *HyperParameters* dictionary. The name of the hyperparameter to be optimized.

Returns A local part of new *HyperParameters[Parameter]* value.

Return type FloatTensor

W = None
Object **W** is a model parameters

forward(*input*)
Returns model prediction for the given input data. Linear prediction is $input@w$.

Warning: The number *num_answers* can be just *1*.

Parameters **input** (*FloatTensor.*) – The tensor of shape *num_elements* $\times$ *num_feature*.

Returns The tensor of shape *num_elements* $\times$ *num_answers*. Model answers for the given input data

Return type FloatTensor

CHAPTER 5

Regularizers

The *mixturelib.regularizers* contains classes:

- *mixturelib.regularizers.Regularizers*
- *mixturelib.regularizers.RegularizeModel*
- *mixturelib.regularizers.RegularizeFunc*

class mixturelib.regularizers.**RegularizeFunc**(*ListOfModels=None*, *R=<function RegularizeFunc.<lambda>>*, *epoch=100*, *device='cpu'*)

The class of regularization to create any relationship between prior means. The relationship between the parameters is set by using the link function.

In the M-step solves next optimisation problem $\sum_{k=1}^{num\_models} \left[-\frac{1}{2} w_k^0 A_k^{-1} w_k^0 + w_k^0 A_k^{-1} \mathsf{E} w_k\right] + R(W^0) \to \infty$.

Warning: All local models must be Linear model for the regression task. Also can be used *mixturelib.local_models.EachModelLinear*.

Warning: Link function represent a likelihood. This function will be maximizing during optimisation.

This Regularizer make correction on the M-step for each Linear Model.

Parameters

- **ListOfModels** (*list*) – A list of local models to be regularized.
- **device** (*string*) – The device for pytorch. Can be 'cpu' or 'gpu'. Default 'cpu'.
- **R** (*function*) – The link function between prior means for all local models. The function must be scalar with type FloatTensor.
- **epoch** (*int*) – The number of epoch for solving optimisation problem in the M-step.

Example:

```
>>> _ = torch.random.manual_seed(42) # Set random seed for repeatability
>>>
>>> w = torch.randn(2, 1) # Generate real parameter vector
>>> X = torch.randn(10, 2) # Generate features data
>>> Z = torch.ones(10, 1) # Set that all data correspond to this model
>>> Y = X@w + 0.1*torch.randn(10, 1) # Generate target data with noise 0.1
>>>
>>> first_model = EachModelLinear(
...     input_dim=2,
...     A=torch.tensor([1., 1.]),
...     w=torch.tensor([[0.], [0.]])) # Init first local model
>>> second_model = EachModelLinear(
...     input_dim=2,
...     A=torch.tensor([1., 1.]),
...     w=torch.tensor([[1.], [1.]])) # Init second local model
>>> hyper_parameters = {
...     'alpha': torch.tensor([1., 1e-10])} # Set regularization parameter
>>>
>>> first_model.w_0, first_model.W # First prior and paramaters before
(tensor([[0.],
         [0.]]),
tensor([[1.3314e-06],
        [8.6398e-06]]))
>>> second_model.w_0, second_model.W # Second prior and paramaters before
(tensor([[1.],
         [1.]]),
 tensor([[1.0000],
         [1.0000]]))
>>>
>>> Rg = RegularizeModel(
...     ListOfModels=[first_model, second_model],
...     R = lambda x: -(x**2).sum()) # Set regulariser
>>> _ = Rg.M_step(X, Y, Z, hyper_parameters) # Regularize
>>>
>>> first_model.w_0, first_model.W # First prior and paramaters after
(tensor([[4.8521e-06],
         [6.7789e-06]]),
 tensor([[1.3314e-06],
         [8.6398e-06]]))
>>> second_model.w_0, second_model.W # Second prior and paramaters after
(tensor([[0.9021],
         [0.9021]]),
 tensor([[1.0000],
         [1.0000]]))
```

E_step (*X*, *Y*, *Z*, *HyperParameters*)

The method does nothing.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

M_step (*X*, *Y*, *Z*, *HyperParameters*)

Make some regularization on the M-step.

Solves next optimisation problem $\sum_{k=1}^{num\_models} \left[-\frac{1}{2} w_k^0 A_k^{-1} w_k^0 + w_k^0 A_k^{-1} \mathsf{E} w_k\right] + R(W^0) \to \infty$.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

class mixturelib.regularizers.**RegularizeModel**(*ListOfModels=None*, *device='cpu'*)

The class of regularization to create a relationship between prior means. The relationship between the parameters in this case, is that the mean distributions should be equal.

> **Warning:** All local models must be Linear model for the regression task. Also can be used *mixturelib.local_models.EachModelLinear*.

This Regularizer make correction on the M-step for each Linear Model.

Parameters

- **ListOfModels** (*list*) – A list of local models to be regularized.
- **device** (*string*) – The device for pytorch. Can be 'cpu' or 'gpu'. Default 'cpu'.

Example:

```
>>> _ = torch.random.manual_seed(42) # Set random seed for repeatability
>>>
>>> w = torch.randn(2, 1) # Generate real parameter vector
>>> X = torch.randn(10, 2) # Generate features data
>>> Z = torch.ones(10, 1) # Set that all data correspond to this model
>>> Y = X@w + 0.1*torch.randn(10, 1) # Generate target data with noise 0.1
>>>
>>> first_model = EachModelLinear(
...     input_dim=2,
...     A=torch.tensor([1., 1.]),
...     w=torch.tensor([[0.], [0.]])) # Init first local model
>>> second_model = EachModelLinear(
...     input_dim=2,
...     A=torch.tensor([1., 1.]),
...     w=torch.tensor([[1.], [1.]])) # Init second local model
>>> hyper_parameters = {
...     'alpha': torch.tensor([1., 1e-10])} # Set regularization parameter
>>>
>>> first_model.w_0, first_model.W # First prior and paramaters before
(tensor([[0.],
         [0.]]),
tensor([[1.3314e-06],
        [8.6398e-06]]))
>>> second_model.w_0, second_model.W # Second prior and paramaters before
(tensor([[1.],
         [1.]]),
 tensor([[1.0000],
         [1.0000]]))
```

(continues on next page)

(continued from previous page)

```
>>>
>>> Rg = RegularizeModel(
...     ListOfModels=[first_model, second_model]) # Set regulariser
>>> _ = Rg.M_step(X, Y, Z, hyper_parameters) # Regularize
>>>
>>> first_model.w_0, first_model.W # First prior and paramaters after
(tensor([[0.3333],
         [0.5000]]),
 tensor([[1.3314e-06],
         [8.6398e-06]]))
>>> second_model.w_0, second_model.W # Second prior and paramaters after
(tensor([[0.6667],
         [0.5000]]),
 tensor([[1.0000],
         [1.0000]]))
```

E_step (*X*, *Y*, *Z*, *HyperParameters*)
The method does nothing.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

M_step (*X*, *Y*, *Z*, *HyperParameters*)
Make some regularization on the M-step.

For all local model from ListOfModels with prior, make regularization $w_k^0 = \left[A_k^{-1} + (num\_models - 1)\alpha\right]\left(A_k^{-1}\mathsf{E}w_k + \alpha\sum_{k'\neq k} w'_k\right)$

Warning: HyperParameters must contain *alpha* hyperparameter.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

class mixturelib.regularizers.**Regularizers**
Base class for all regulizers.

E_step (*X*, *Y*, *Z*, *HyperParameters*)
Make some regularization on the E-step.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.

- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *1*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

M_step (*X*, *Y*, *Z*, *HyperParameters*)
Make some regularization on the M-step.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *1*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

CHAPTER 6

Hyper Models

The *mixturelib.hyper_models* contains classes:

- *mixturelib.hyper_models.HyperModel*
- *mixturelib.hyper_models.HyperModelDirichlet*
- *mixturelib.hyper_models.HyperExpertNN*

class mixturelib.hyper_models.**HyperExpertNN**(*input_dim=20*, *hidden_dim=10*, *output_dim=10*, *epochs=100*, *device='cpu'*)

A hyper model for mixture of experts. The hyper model prediction on local models probability are depend on the object.

In this hyper model, the probability of each local model is a neural network prediction with softmax. Neural network is a three layer fully conected neural network.

Parameters

- **input_dim** (*int*) – The number of features.
- **hidden_dim** (*int*) – The number of parameters in hidden layer.
- **output_dim** (*int*) – The number of local models.
- **epochs** (*int*) – The number epoch to train neural network in each step.
- **device** – The device for pytorch. Can be 'cpu' or 'gpu'. Default 'cpu'.

Example:

```
>>> _ = torch.random.manual_seed(42) # Set random seed for repeatability
>>>
>>> w = torch.randn(2, 1) # Generate real parameter vector
>>> X = torch.randn(5, 2) # Generate features data
>>> Z = torch.distributions.dirichlet.Dirichlet(
...     torch.tensor([0.5, 0.5])).sample(
...         (5,)) # Set corresponding between data and local models.
>>> Y = X@w + 0.1*torch.randn(5, 1) # Generate target data with noise 0.1
```

(continues on next page)

(continued from previous page)

```
>>>
>>> hyper_model = HyperExpertNN(
...     input_dim=2,
...     output_dim=2) # Init hyper model with Diriclet weighting
>>> hyper_parameters = {} # Withor hyper parameters
>>>
>>> hyper_model.LogPiExpectation(
...     X, Y, hyper_parameters) # Log of probability before E step
tensor([[-0.4981, -0.9356],
        [-0.5176, -0.9063],
        [-0.4925, -0.9443],
        [-0.4957, -0.9395],
        [-0.4969, -0.9376]])
>>>
>>> hyper_model.E_step(X, Y, Z, hyper_parameters)
>>> hyper_model.LogPiExpectation(
...     X, Y, hyper_parameters)  # Log of probability after E step
tensor([[-0.6294, -0.7612],
        [-0.9327, -0.5000],
        [-0.3273, -1.2760],
        [-0.5775, -0.8239],
        [-0.5357, -0.8801]])
```

E_step (*X*, *Y*, *Z*, *HyperParameters*)
The method does nothing.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

LogPiExpectation (*X*, *Y*, *HyperParameters*)
Returns the expected value of each models log of probability.

Takes log softmax from the forward method.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

Returns The tensor of shape *num_elements* × *num_models*. The espected value of each models probability.

Return type FloatTensor

M_step (*X*, *Y*, *Z*, *HyperParameters*)
Doing M-step of EM-algorithm. Finds model parameters by using gradient descent.

Parameters are optimized with respect to the loss function $loss = -\sum_{i=1}^{num\_elements}\sum_{k=1}^{num\_models}\log \pi_k(x_i, V)$, where V is a neural network parameters.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

PredictPi (*X*, *HyperParameters*)

Returns the probability (weight) of each models.

Takes softmax from the forward method.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

Returns The tensor of shape *num_elements* × *num_models*. The probability (weight) of each models.

Return type FloatTensor

forward (*input*)

Returns model prediction for the given input data.

> **Warning:** The number *num_answers* can be just *1*.

Parameters **input** (*FloatTensor.*) – The tensor of shape *num_elements* × *num_feature*.

Returns The tensor of shape *num_elements* × *num_models*. Model prediction of probability for all local models for the given input data.

Return type FloatTensor

class mixturelib.hyper_models.**HyperModel**

Base class for all hyper models.

E_step (*X*, *Y*, *Z*, *HyperParameters*)

Doing E-step of EM-algorithm. Finds variational probability *q* of model parameters.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

LogPiExpectation (*X*, *Y*, *HyperParameters*)

Returns the expected value of each models probability.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.

- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

M_step (*X*, *Y*, *Z*, *HyperParameters*)
Doing M-step of EM-algorithm. Finds model hyper parameters.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

PredictPi (*X*, *HyperParameters*)
Returns the probability of each models.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

class mixturelib.hyper_models.**HyperModelDirichlet** (*output_dim=2*, *device='cpu'*)
A hyper model for mixture of model. The hyper model cannot predict local model for each object, because model probability does not depend on object.

In this hyper model, the probability of each local model is a vector from dirichlet distribution with parameter μ.

Parameters

- **output_dim** (*int*) – The number of local models.
- **device** – The device for pytorch. Can be 'cpu' or 'gpu'. Default 'cpu'.

Example:

```
>>> _ = torch.random.manual_seed(42) # Set random seed for repeatability
>>>
>>> w = torch.randn(2, 1) # Generate real parameter vector
>>> X = torch.randn(5, 2) # Generate features data
>>> Z = torch.distributions.dirichlet.Dirichlet(
...     torch.tensor([0.5, 0.5])).sample(
...         (5,)) # Set corresponding between data and local models.
>>> Y = X@w + 0.1*torch.randn(5, 1) # Generate target data with noise 0.1
>>>
>>> hyper_model = HyperModelDirichlet(
...     output_dim=2) # Init hyper model with Diriclet weighting
>>> hyper_parameters = {} # Withor hyper parameters
>>>
>>> hyper_model.LogPiExpectation(
...     X, Y, hyper_parameters) # Log of probability before E step
tensor([[-1.0000, -1.0000],
        [-1.0000, -1.0000],
        [-1.0000, -1.0000],
        [-1.0000, -1.0000],
```

(continues on next page)

(continued from previous page)

```
        [-1.0000, -1.0000]])
>>>
>>> hyper_model.E_step(X, Y, Z, hyper_parameters)
>>> hyper_model.LogPiExpectation(
...     X, Y, hyper_parameters)  # Log of probability after E step
tensor([[-0.7118, -0.8310],
        [-0.7118, -0.8310],
        [-0.7118, -0.8310],
        [-0.7118, -0.8310],
        [-0.7118, -0.8310]])
```

E_step (*X*, *Y*, *Z*, *HyperParameters*)

Doing E-step of EM-algorithm. Finds variational probability *q* of model parameters.

Calculate analytical solution for estimate *q* in the class of normal distributions $q = Dir(m)$, where $m = \mu + \gamma$, where $\gamma_k = \sum_{i=1}^{num\_elements} Z_{ik}$, and μ is prior.

> **Warning:** Now μ_k is *1* for all *k*, and can not be changed.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

LogPiExpectation (*X*, *Y*, *HyperParameters*)

Returns the expected value of each models log of probability.

Returns the expectation of $\log \pi$ value where π is a random value from Dirichlet distribution.

This function calculates by using Γ function

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

Returns The tensor of shape *num_elements* × *num_models*. The espected value of each models probability.

Return type FloatTensor

M_step (*X*, *Y*, *Z*, *HyperParameters*)

The method does nothing.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.

- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

PredictPi(*X*, *HyperParameters*)

Returns the probability (weight) of each models.

Return the same vector π for all object. Each $\pi = \frac{\mathbf{m}}{\sum \mathbf{m}_k}$, where $\mathbf{m}$ is a parameter of Dirichlet pdf.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* $\times$ *num_feature*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

Returns The tensor of shape *num_elements* $\times$ *num_models*. The probability (weight) of each models.

Return type FloatTensor

class mixturelib.hyper_models.**HyperModelGateSparsed**(*output_dim=2*, *gamma=1.0*, *mu=<sphinx.ext.autodoc.importer._MockObject object>*, *device='cpu'*)

A hyper model for mixture of model. Each i-th object from train dataset has own probability to each model π^i.

In this hyper model, the probability of each local model is a vector from dirichlet distribution with parameter μ, and l.

Parameters

- **output_dim** (*int*) – The number of local models.
- **device** – The device for pytorch. Can be 'cpu' or 'gpu'. Default 'cpu'.

Example:

```
>>> _ = torch.random.manual_seed(42) # Set random seed for repeatability
>>>
>>> w = torch.randn(2, 1) # Generate real parameter vector
>>> X = torch.randn(5, 2) # Generate features data
>>> Z = torch.distributions.dirichlet.Dirichlet(
...     torch.tensor([0.5, 0.5])).sample(
...         (5,)) # Set corresponding between data and local models.
>>> Y = X@w + 0.1*torch.randn(5, 1) # Generate target data with noise 0.1
>>>
>>> hyper_model = HyperModelGateSparsed(
...     output_dim=2) # Model with Diriclet weighting for each sample
>>> hyper_parameters = {} # Withor hyper parameters
>>>
>>> hyper_model.LogPiExpectation(
...     X, Y, hyper_parameters) # Log of probability before E step
tensor([[-1.3863, -1.3863],
            [-1.3863, -1.3863],
            [-1.3863, -1.3863],
            [-1.3863, -1.3863],
            [-1.3863, -1.3863]])
>>>
>>> hyper_model.E_step(X, Y, Z, hyper_parameters)
>>> hyper_model.LogPiExpectation(
...     X, Y, hyper_parameters)  # Log of probability after E step
tensor([[-1.9677, -0.4830],
            [-1.7785, -0.5417],
```

(continues on next page)

(continued from previous page)

```
            [-0.5509, -1.7521],
            [-0.7250, -1.3642],
            [-0.4839, -1.9644]])
```

E_step (*X*, *Y*, *Z*, *HyperParameters*)

Doing E-step of EM-algorithm. Finds variational probability *q* of model parameters.

Calculate analytical solution for estimate q in the class of normal distributions $q = Dir(m)$, where $m = \mu + \gamma$, where $\gamma_k = \sum_{i=1}^{num\_elements} Z_{ik}$, and μ is prior.

Warning: Now μ_k is *1* for all *k*, and can not be changed.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

LogPiExpectation (*X*, *Y*, *HyperParameters*)

Returns the expected value of each models log of probability.

Returns the expectation of $\log \pi$ value where π is a random value from Dirichlet distribution.

This function calculates by using $\digamma$ function

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

Returns The tensor of shape *num_elements* × *num_models*. The espected value of each models probability.

Return type FloatTensor

M_step (*X*, *Y*, *Z*, *HyperParameters*)

The method does nothing.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.
- **Z** (*FloatTensor*) – The tensor of shape *num_elements* × *num_models*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

PredictPi (*X*, *HyperParameters*)

Returns the probability (weight) of each models.

Return the same vector π for all object. Each $\pi = \frac{\mathbf{m}}{\sum \mathbf{m}_k}$, where $\mathbf{m}$ is a parameter of Dirichlet pdf.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **HyperParameters** (*dict*) – The dictionary of all hyper parametrs. Where *key* is string and *value* is FloatTensor.

Returns The tensor of shape *num_elements* × *num_models*. The probability (weight) of each models.

Return type FloatTensor

CHAPTER 7

Mixture

The *mixturelib.mixture* contains classes:

- *mixturelib.mixture.Mixture*
- *mixturelib.mixture.MixtureEM*

class mixturelib.mixture.**Mixture**

Base class for all mixtures.

fit(*X=None*, *Y=None*, *epoch=10*, *progress=None*)

A method that fit a hyper model and local models in one procedure.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** – The tensor of shape *num_elements* × *num_answers*.
- **epoch** (*function*) – The number of epoch of training.
- **progress** – The yield function for printing progress, like a tqdm. The function must take an iterator at the input and return the same data.

predict(*X*)

A method that predict value for given input data.

Parameters **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.

Returns The prediction of shape *num_elements* × *num_answers*.

Return type FloatTensor

class mixturelib.mixture.**MixtureEM**(*HyperParameters={}*, *HyperModel=None*, *ListOfModels=None*, *ListOfRegularizeModel=None*, *model_type='default'*, *device='cpu'*)

The implementation of EM-algorithm for solving the two stage optimisation problem.

Warning: All Hyper Parameters should be additive to models, when you wanna optimize them.

Parameters

- **HyperParameters** – The dictionary of all hyper parametrs. Where *key* is string and *value* is float or FloatTensor.
- **HyperModel** (*mixturelib.hyper_models.HyperModel*) – The hyper model which are weighted all local models.
- **ListOfModels** (*list*) – The list of models with E_step and M_step methods.
- **ListOfRegularizeModel** (*list*) – The list of regulizers with E_step and M_step methods.
- **model_type** (*string*) – Type os EM algorithm. Can be *default* or *sample*. In *default* EM model all objects uses in each local models with weights. In *sample* EM model all objects are sampled during to their weights and just sampled samples uses in local models.
- **device** (*string*) – The device for pytorch. Can be 'cpu' or 'gpu'. Default 'cpu'.

Example:

```
>>> _ = torch.random.manual_seed(42) # Set random seed for repeatability
>>>
>>> first_w = torch.randn(2, 1) # Generate first real parameter vector
>>> second_w = torch.randn(2, 1) # Generate second real parameter vector
>>> X = torch.randn(102, 2) # Generate features data
>>> Y = torch.cat(
...         [
...             X[:50]@first_w,
...             X[50:10]@second_w,
...             X[100:101]@first_w,
...             X[101:]@second_w
...         ])
...     + 0.01 * torch.randn(102, 1) # Generate target data with noise 0.1
>>>
>>> first_model = EachModelLinear(
...     input_dim=2,
...     A=torch.tensor([1., 1.]),
...     w=torch.tensor([0., 0.])) # Init first local model
>>> second_model = EachModelLinear(
...     input_dim=2,
...     A=torch.tensor([1., 1.]),
...     w=torch.tensor([[1.], [1.]])) # Init second local model
>>> hyper_model = HyperExpertNN(
...     input_dim=2,
...     output_dim=2) # Init hyper model with Diriclet weighting
>>> hyper_parameters = {'beta': 1.} # Withor hyper parameters
>>>
>>> mixture = MixtureEM(
...     HyperModel=hyper_model,
...     HyperParameters=hyper_parameters,
...     ListOfModels=[first_model, second_model],
...     model_type='sample') # Init hyper model
>>> mixture.fit(X[:100], Y[:100]) # Optimise model parameter
>>>
>>> mixture.predict(X[100:])[0].view(-1)
```

(continues on next page)

(continued from previous page)

```
tensor([-0.1245, -0.4357])
>>> Y[100:].view(-1)
tensor([-0.0936, -0.4177])
```

E_step (*X*, *Y*)

Doing E-step of EM-algorigthm. This method call E_step for all local models, for hyper model and for all regularizations step by step.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.

M_step (*X*, *Y*)

Doing M-step of EM-algorigthm. This method call M_step for all local models, for hyper model and for all regularizations step by step.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** (*FloatTensor*) – The tensor of shape *num_elements* × *num_answers*.

fit (*X=None*, *Y=None*, *epoch=10*, *progress=None*)

A method that fit a hyper model and local models in one procedure.

Call E-step and M-step in each epoch.

Parameters

- **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.
- **Y** – The tensor of shape *num_elements* × *num_answers*.
- **epoch** (*function*) – The number of epoch of training.
- **progress** – The yield function for printing progress, like a tqdm. The function must take an iterator at the input and return the same data.

predict (*X*)

A method that predict value for given input data.

For each x from X predicts $answer = \sum_{k=1}^{K} \pi_k(x) g_k(x)$, where g_k is a local model.

Parameters **X** (*FloatTensor*) – The tensor of shape *num_elements* × *num_feature*.

Returns

The prediction of shape *num_elements* × *num_answers*.

The probability of shape *num_elements* × *num_models*.

Return type FloatTensor, FloatTensor

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

O

P

R

W