reflame

Release 1.0.1

Thieu

Dec 03, 2023

QUICK START:

Reflame (REvolutionizing Functional Link Artificial neural networks by MEtaheuristic algorithms) is a Python library that implements a framework for training Functional Link Neural Network (FLNN) networks using Metaheuristic Algorithms. It provides a comparable alternative to the traditional FLNN network and is compatible with the Scikit-Learn library. With Reflame, you can perform searches and hyperparameter tuning using the functionalities provided by the Scikit-Learn library.

- **Free software:** GNU General Public License (GPL) V3 license
- **Provided Estimator**: FlnnRegressor, FlnnClassifier, MhaFlnnRegressor, MhaFlnnClassifier
- **Total Official Metaheuristic-based Flnn Regression**: > 200 Models
- **Total Official Metaheuristic-based Flnn Classification**: > 200 Models
- **Supported performance metrics**: >= 67 (47 regressions and 20 classifications)
- **Supported objective functions (as fitness functions or loss functions)**: >= 67 (47 regressions and 20 classifications)
- **Documentation:** https://reflame.readthedocs.io
- **Python versions:** >= 3.8.x
- **Dependencies:** numpy, scipy, scikit-learn, pandas, mealpy, permetrics, torch, skorch

CHAPTER

ONE

INSTALLATION

- Install the current PyPI release:

```
$ pip install reflame==1.0.1
```

- Install directly from source code:

```
$ git clone https://github.com/thieu1995/reflame.git
$ cd reflame
$ python setup.py install
```

- In case, you want to install the development version from Github:

```
$ pip install git+https://github.com/thieu1995/reflame
```

After installation, you can import Reflame as any other Python module:

```
$ python
>>> import reflame
>>> reflame.__version__
```

CHAPTER

TWO

EXAMPLES

In this section, we will explore the usage of the Reflame model with the assistance of a dataset. While all the preprocessing steps mentioned below can be replicated using Scikit-Learn, we have implemented some utility functions to provide users with convenience and faster usage.

Combine Reflame library like a normal library with scikit-learn:

```
### Step 1: Importing the libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from reflame import FlnnRegressor, FlnnClassifier, MhaFlnnRegressor, MhaFlnnClassifier

#### Step 2: Reading the dataset
dataset = pd.read_csv('Position_Salaries.csv')
X = dataset.iloc[:, 1:2].values
y = dataset.iloc[:, 2].values

#### Step 3: Next, split dataset into train and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=True,␣
↪random_state=100)

#### Step 4: Feature Scaling
scaler_X = MinMaxScaler()
scaler_X.fit(X_train)
X_train = scaler_X.transform(X_train)
X_test = scaler_X.transform(X_test)

le_y = LabelEncoder()  # This is for classification problem only
le_y.fit(y)
y_train = le_y.transform(y_train)
y_test = le_y.transform(y_test)

#### Step 5: Fitting FLNN-based model to the dataset

##### 5.1: Use standard FLNN model for regression problem
regressor = FlnnRegressor(expand_name="chebyshev", n_funcs=4, act_name="elu",
                      obj_name="MSE", max_epochs=100, batch_size=32, optimizer="SGD",␣
↪verbose=True)
regressor.fit(X_train, y_train)

##### 5.2: Use standard FLNN model for classification problem
```

(continues on next page)

(continued from previous page)

```
classifer = FlnnClassifier(expand_name="chebyshev", n_funcs=4, act_name="sigmoid",
                       obj_name="BCEL", max_epochs=100, batch_size=32, optimizer="SGD",␣
↪verbose=True)
classifer.fit(X_train, y_train)

##### 5.3: Use Metaheuristic-based FLNN model for regression problem
print(MhaFlnnClassifier.SUPPORTED_OPTIMIZERS)
print(MhaFlnnClassifier.SUPPORTED_REG_OBJECTIVES)
opt_paras = {"name": "GA", "epoch": 10, "pop_size": 30}
model = MhaFlnnRegressor(expand_name="chebyshev", n_funcs=3, act_name="elu",
                         obj_name="RMSE", optimizer="BaseGA", optimizer_paras=opt_paras,␣
↪verbose=True)
regressor.fit(X_train, y_train)

##### 5.4: Use Metaheuristic-based FLNN model for classification problem
print(MhaFlnnClassifier.SUPPORTED_OPTIMIZERS)
print(MhaFlnnClassifier.SUPPORTED_CLS_OBJECTIVES)
opt_paras = {"name": "GA", "epoch": 10, "pop_size": 30}
classifier = MhaFlnnClassifier(expand_name="chebyshev", n_funcs=4, act_name="sigmoid",
                          obj_name="NPV", optimizer="BaseGA", optimizer_paras=opt_paras,␣
↪verbose=True)
classifier.fit(X_train, y_train)

#### Step 6: Predicting a new result
y_pred = regressor.predict(X_test)

y_pred_cls = classifier.predict(X_test)
y_pred_label = le_y.inverse_transform(y_pred_cls)

#### Step 7: Calculate metrics using score or scores functions.
print("Try my AS metric with score function")
print(regressor.score(X_test, y_test, method="AS"))

print("Try my multiple metrics with scores function")
print(classifier.scores(X_test, y_test, list_methods=["AS", "PS", "F1S", "CEL", "BSL"]))


import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from reflame import MhaFlnnRegressor, MhaFlnnClassifier

#### Step 2: Reading the dataset
dataset = pd.read_csv('Position_Salaries.csv')
X = dataset.iloc[:, 1:2].values
y = dataset.iloc[:, 2].values

#### Step 3: Next, split dataset into train and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=True,␣
↪random_state=100)

#### Step 4: Feature Scaling
```

(continues on next page)

(continued from previous page)

```
scaler_X = MinMaxScaler()
scaler_X.fit(X_train)
X_train = scaler_X.transform(X_train)
X_test = scaler_X.transform(X_test)

le_y = LabelEncoder()       # This is for classification problem only
le_y.fit(y)
y_train = le_y.transform(y_train)
y_test = le_y.transform(y_test)

#### Step 5: Fitting FLNN-based model to the dataset

##### 5.1: Use standard FLNN model for regression problem
regressor = FlnnRegressor(hidden_size=10, act_name="relu")
regressor.fit(X_train, y_train)

##### 5.2: Use standard FLNN model for classification problem
classifer = FlnnClassifier(hidden_size=10, act_name="tanh")
classifer.fit(X_train, y_train)

##### 5.3: Use Metaheuristic-based FLNN model for regression problem
print(MhaFlnnClassifier.SUPPORTED_OPTIMIZERS)
print(MhaFlnnClassifier.SUPPORTED_REG_OBJECTIVES)
opt_paras = {"name": "GA", "epoch": 10, "pop_size": 30}
regressor = MhaFlnnRegressor(hidden_size=10, act_name="elu", obj_name="RMSE", optimizer=
↪"BaseGA", optimizer_paras=opt_paras)
regressor.fit(X_train, y_train)

##### 5.4: Use Metaheuristic-based FLNN model for classification problem
print(MhaFlnnClassifier.SUPPORTED_OPTIMIZERS)
print(MhaFlnnClassifier.SUPPORTED_CLS_OBJECTIVES)
opt_paras = {"name": "GA", "epoch": 10, "pop_size": 30}
classifier = MhaFlnnClassifier(hidden_size=10, act_name="elu", obj_name="KLDL",␣
↪optimizer="BaseGA", optimizer_paras=opt_paras)
classifier.fit(X_train, y_train)

#### Step 6: Predicting a new result
y_pred = regressor.predict(X_test)

y_pred_cls = classifier.predict(X_test)
y_pred_label = le_y.inverse_transform(y_pred_cls)

#### Step 7: Calculate metrics using score or scores functions.
print("Try my AS metric with score function")
print(regressor.score(data.X_test, data.y_test, method="AS"))

print("Try my multiple metrics with scores function")
print(classifier.scores(data.X_test, data.y_test, list_methods=["AS", "PS", "F1S", "CEL",
↪ "BSL"]))
```

Utilities everything that Reflame provided:

```
### Step 1: Importing the libraries
from reflame import Data, FlnnRegressor, FlnnClassifier, MhaFlnnRegressor,␣
↪MhaFlnnClassifier
from sklearn.datasets import load_digits

#### Step 2: Reading the dataset
X, y = load_digits(return_X_y=True)
data = Data(X, y)

#### Step 3: Next, split dataset into train and test set
data.split_train_test(test_size=0.2, shuffle=True, random_state=100)

#### Step 4: Feature Scaling
data.X_train, scaler_X = data.scale(data.X_train, scaling_methods=("minmax"))
data.X_test = scaler_X.transform(data.X_test)

data.y_train, scaler_y = data.encode_label(data.y_train)  # This is for classification␣
↪problem only
data.y_test = scaler_y.transform(data.y_test)

#### Step 5: Fitting FLNN-based model to the dataset

##### 5.1: Use standard FLNN model for regression problem
regressor = FlnnRegressor(expand_name="chebyshev", n_funcs=4, act_name="tanh",
                      obj_name="MSE", max_epochs=100, batch_size=32, optimizer="SGD",␣
↪verbose=True)
regressor.fit(data.X_train, data.y_train)

##### 5.2: Use standard FLNN model for classification problem
classifer = FlnnClassifier(expand_name="chebyshev", n_funcs=4, act_name="tanh",
                      obj_name="BCEL", max_epochs=100, batch_size=32, optimizer="SGD",␣
↪verbose=True)
classifer.fit(data.X_train, data.y_train)

##### 5.3: Use Metaheuristic-based FLNN model for regression problem
print(MhaFlnnClassifier.SUPPORTED_OPTIMIZERS)
print(MhaFlnnClassifier.SUPPORTED_REG_OBJECTIVES)
opt_paras = {"name": "GA", "epoch": 10, "pop_size": 30}
model = MhaFlnnRegressor(expand_name="chebyshev", n_funcs=3, act_name="elu",
                         obj_name="RMSE", optimizer="BaseGA", optimizer_paras=opt_paras,␣
↪verbose=True)
regressor.fit(data.X_train, data.y_train)

##### 5.4: Use Metaheuristic-based FLNN model for classification problem
print(MhaFlnnClassifier.SUPPORTED_OPTIMIZERS)
print(MhaFlnnClassifier.SUPPORTED_CLS_OBJECTIVES)
opt_paras = {"name": "GA", "epoch": 10, "pop_size": 30}
classifier = MhaFlnnClassifier(expand_name="chebyshev", n_funcs=4, act_name="sigmoid",
                           obj_name="NPV", optimizer="BaseGA", optimizer_paras=opt_paras,␣
↪verbose=True)
classifier.fit(data.X_train, data.y_train)

#### Step 6: Predicting a new result
```

(continues on next page)

(continued from previous page)

```
y_pred = regressor.predict(data.X_test)

y_pred_cls = classifier.predict(data.X_test)
y_pred_label = scaler_y.inverse_transform(y_pred_cls)

#### Step 7: Calculate metrics using score or scores functions.
print("Try my AS metric with score function")
print(regressor.score(data.X_test, data.y_test, method="AS"))

print("Try my multiple metrics with scores function")
print(classifier.scores(data.X_test, data.y_test, list_methods=["AS", "PS", "F1S", "CEL",
↪ "BSL"]))
```

A real-world dataset contains features that vary in magnitudes, units, and range. We would suggest performing normalization when the scale of a feature is irrelevant or misleading. Feature Scaling basically helps to normalize the data within a particular range.

CHAPTER

THREE

REFLAME PACKAGE

3.1 reflame.model package

3.1.1 reflame.model.mha_flnn module

class reflame.model.mha_flnn.**MhaFlnnClassifier**(*expand_name='chebyshev'*, *n_funcs=4*, *act_name='none'*, *obj_name=None*, *optimizer='BaseGA'*, *optimizer_paras=None*, *verbose=False*)

Bases: *BaseMhaFlnn*, ClassifierMixin

Defines the general class of Metaheuristic-based FLNN model for Classification problems that inherit the BaseMhaFlnn and ClassifierMixin classes.

Parameters

- **expand_name** (*str, default="chebyshev"*) – The expand function that will be used. The supported expand functions are: {"chebyshev", "legendre", "gegenbauer", "laguerre", "hermite", "power", "trigonometric"}
- **n_funcs** (*int, default=4*) – The first *n_funcs* in expand functions list will be used. Valid value from 1 to 10.
- **act_name** (*str, default='none'*) – Activation function for the hidden layer. The supported activation functions are: {"none", "relu", "prelu", "gelu", "elu", "selu", "rrelu", "tanh", "hard_tanh", "sigmoid", "hard_sigmoid", "swish", "hard_swish", "soft_plus", "mish", "soft_sign", "tanh_shrink", "soft_shrink", "hard_shrink"}
- **obj_name** (*str, default="AS"*) – Current supported objective functions, please check it here: https://github.com/thieu1995/permetrics
- **optimizer** (*str or instance of Optimizer class (from Mealpy library), default = "BaseGA"*) – The Metaheuristic Algorithm that use to solve the feature selection problem. Current supported list, please check it here: https://github.com/thieu1995/mealpy. If a custom optimizer is passed, make sure it is an instance of *Optimizer* class.
- **optimizer_paras** (*None or dict of parameter, default=None*) – The parameter for the *optimizer* object. If *None*, the default parameters of optimizer is used (defined in https://github.com/thieu1995/mealpy.) If *dict* is passed, make sure it has at least *epoch* and *pop_size* parameters.
- **verbose** (*bool, default=False*) – Whether to print progress messages to stdout.
- **[Optional]** (*obj_weights*) – The objective weights for multiple objective functions

Examples

```
>>> from reflame import Data, MhaFlnnClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=100, random_state=1)
>>> data = Data(X, y)
>>> data.split_train_test(test_size=0.2, random_state=1)
>>> data.X_train_scaled, scaler = data.scale(data.X_train, method="MinMaxScaler")
>>> data.X_test_scaled = scaler.transform(data.X_test)
>>> opt_paras = {"name": "GA", "epoch": 10, "pop_size": 30}
>>> print(MhaFlnnClassifier.SUPPORTED_CLS_OBJECTIVES)
{'PS': 'max', 'NPV': 'max', 'RS': 'max', ...., 'KLDL': 'min', 'BSL': 'min'}
>>> model = MhaFlnnClassifier(expand_name="chebyshev", n_funcs=4, act_name="none",␣
↪obj_name="BSL", optimizer="BaseGA", optimizer_paras=opt_paras)
>>> model.fit(data.X_train_scaled, data.y_train)
>>> pred = model.predict(data.X_test_scaled)
>>> print(pred)
array([1, 0, 1, 0, 1])
```

CLS_OBJ_LOSSES = ['CEL', 'HL', 'KLDL', 'BSL']

create_network(*X*, *y*)

evaluate(*y_true*, *y_pred*, *list_metrics=('AS', 'RS')*)

Return the list of performance metrics on the given test data and labels.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Predicted values for *X*.
- **list_metrics** (*list, default=("AS", "RS")*) – You can get metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns

results – The results of the list metrics

Return type

dict

objective_function(*solution=None*)

Evaluates the fitness function for classification metric

Parameters

solution (*np.ndarray, default=None*) –

Returns

result – The fitness value

Return type

float

score(*X*, *y*, *method='AS'*)

Return the metric on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for *X*.
- **method** (*str, default="AS"*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns
result – The result of selected metric

Return type
float

scores(*X*, *y*, *list_methods=('AS', 'RS')*)

Return the list of metrics on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for *X*.
- **list_methods** (*list, default=("AS", "RS")*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns
results – The results of the list metrics

Return type
dict

set_fit_request(**, lb: bool | None | str = '$UNCHANGED$', save_population: bool | None | str = '$UNCHANGED$', ub: bool | None | str = '$UNCHANGED$'*) → *MhaFlnnClassifier*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

- **lb** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `lb` parameter in `fit`.
- **save_population** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `save_population` parameter in `fit`.
- **ub** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `ub` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_predict_request`(*, *return_prob: bool | None | str = '$UNCHANGED$'*) → *MhaFlnnClassifier*

Request metadata passed to the `predict` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
return_prob (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `return_prob` parameter in `predict`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *method: bool | None | str = '$UNCHANGED$'*) → *MhaFlnnClassifier*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

method (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `method` parameter in `score`.

Returns

self – The updated object.

Return type

object

class reflame.model.mha_flnn.**MhaFlnnRegressor**(*expand_name='chebyshev'*, *n_funcs=4*, *act_name='none'*, *obj_name='MSE'*, *optimizer='BaseGA'*, *optimizer_paras=None*, *verbose=False*, *obj_weights=None*)

Bases: *BaseMhaFlnn*, `RegressorMixin`

Defines the general class of Metaheuristic-based FLNN model for Regression problems that inherit the BaseMhaFlnn and RegressorMixin classes.

Parameters

- **expand_name** (*str, default="chebyshev"*) – The expand function that will be used. The supported expand functions are: {"chebyshev", "legendre", "gegenbauer", "laguerre", "hermite", "power", "trigonometric"}
- **n_funcs** (*int, default=4*) – The first *n_funcs* in expand functions list will be used. Valid value from 1 to 10.
- **act_name** (*str, default='none'*) – Activation function for the hidden layer. The supported activation functions are: {"none", "relu", "prelu", "gelu", "elu", "selu", "rrelu", "tanh", "hard_tanh", "sigmoid", "hard_sigmoid", "swish", "hard_swish", "soft_plus", "mish", "soft_sign", "tanh_shrink", "soft_shrink", "hard_shrink"}

- **obj_name** (*str, default="MSE"*) – Current supported objective functions, please check it here: https://github.com/thieu1995/permetrics
- **optimizer** (*str or instance of Optimizer class (from Mealpy library), default = "BaseGA"*) – The Metaheuristic Algorithm that use to solve the feature selection problem. Current supported list, please check it here: https://github.com/thieu1995/mealpy. If a custom optimizer is passed, make sure it is an instance of *Optimizer* class.
- **optimizer_paras** (*None or dict of parameter, default=None*) – The parameter for the *optimizer* object. If *None*, the default parameters of optimizer is used (defined in https://github.com/thieu1995/mealpy.) If *dict* is passed, make sure it has at least *epoch* and *pop_size* parameters.
- **verbose** (*bool, default=False*) – Whether to print progress messages to stdout.
- **[Optional]** (*obj_weights*) – The objective weights for multiple objective functions

Examples

```
>>> from reflame import MhaFlnnRegressor, Data
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_samples=200, random_state=1)
>>> data = Data(X, y)
>>> data.split_train_test(test_size=0.2, random_state=1)
>>> data.X_train_scaled, scaler = data.scale(data.X_train, method="MinMaxScaler")
>>> data.X_test_scaled = scaler.transform(data.X_test)
>>> opt_paras = {"name": "GA", "epoch": 10, "pop_size": 30}
>>> model = MhaFlnnRegressor(expand_name="chebyshev", n_funcs=4, act_name="none",␣
↪obj_name="RMSE", optimizer="BaseGA", optimizer_paras=opt_paras)
>>> model.fit(data.X_train_scaled, data.y_train)
>>> pred = model.predict(data.X_test_scaled)
>>> print(pred)
```

create_network(*X*, *y*)

Returns

- **network** (*FLNN, an instance of FLNN network*)
- **obj_scaler** (*ObjectiveScaler, the objective scaler that used to scale output*)

evaluate(*y_true*, *y_pred*, *list_metrics=('MSE', 'MAE')*)

Return the list of performance metrics of the prediction.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Predicted values for *X*.
- **list_metrics** (*list, default=("MSE", "MAE")*) – You can get metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns

results – The results of the list metrics

Return type
dict

`objective_function`(*solution=None*)

Evaluates the fitness function for regression metric

Parameters
solution (*np.ndarray, default=None*) –

Returns
result – The fitness value

Return type
float

`score`(*X*, *y*, *method='RMSE'*)

Return the metric of the prediction.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **method** (*str, default="RMSE"*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns
result – The result of selected metric

Return type
float

`scores`(*X*, *y*, *list_methods=('MSE', 'MAE')*)

Return the list of metrics of the prediction.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **list_methods** (*list, default=("MSE", "MAE")*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns
results – The results of the list metrics

Return type
dict

`set_fit_request`(**, lb: bool | None | str = '$UNCHANGED$', save_population: bool | None | str = '$UNCHANGED$', ub: bool | None | str = '$UNCHANGED$'*) → *MhaFlnnRegressor*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

- **lb** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `lb` parameter in `fit`.
- **save_population** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `save_population` parameter in `fit`.
- **ub** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `ub` parameter in `fit`.

Returns

self – The updated object.

Return type

object

set_predict_request(*, *return_prob: bool | None | str = '$UNCHANGED$'*) → *MhaFlnnRegressor*

Request metadata passed to the `predict` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

return_prob (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `return_prob` parameter in `predict`.

Returns

self – The updated object.

Return type

object

set_score_request(*, *method: bool | None | str = '$UNCHANGED$'*) → *MhaFlnnRegressor*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

method (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `method` parameter in `score`.

Returns

self – The updated object.

Return type

object

3.1.2 reflame.model.standard_flnn module

class reflame.model.standard_flnn.**FlnnClassifier**(*expand_name='chebyshev'*, *n_funcs=4*, *act_name='none'*, *obj_name='NLLL'*, *max_epochs=1000*, *batch_size=32*, *optimizer='SGD'*, *optimizer_paras=None*, *verbose=False*, ***kwargs*)

Bases: *BaseFlnn*

Defines the class for traditional FLNN network for Classification problems that inherit the BaseFlnn class

Parameters

- **expand_name** (*str, default="chebyshev"*) – The expand function that will be used. The supported expand functions are: {"chebyshev", "legendre", "gegenbauer", "laguerre", "hermite", "power", "trigonometric"}
- **n_funcs** (*int, default=4*) – The first *n_funcs* in expand functions list will be used. Valid value from 1 to 10.
- **act_name** (*{"none", "relu", "leaky_relu", "celu", "prelu", "gelu", "elu", "selu", "rrelu", "tanh", "hard_tanh",}*) – "sigmoid", "hard_sigmoid", "log_sigmoid", "silu", "swish", "hard_swish", "soft_plus", "mish", "soft_sign", "tanh_shrink", "soft_shrink", "hard_shrink", "softmin", "softmax", "log_softmax" }, default='none' Activation function for the hidden layer.
- **obj_name** (*str, default=NLLL*) – The name of objective for classification problem (binary and multi-class classification)
- **max_epochs** (*int, default=1000*) – Maximum number of epochs / iterations / generations
- **batch_size** (*int, default=32*) – The batch size
- **optimizer** (*str, default = "SGD"*) – The gradient-based optimizer from Pytorch. List of supported optimizer is: ["Adadelta", "Adagrad", "Adam", "Adamax", "AdamW", "ASGD", "LBFGS", "NAdam", "RAdam", "RMSprop", "Rprop", "SGD"]
- **optimizer_paras** (*dict or None, default=None*) – The dictionary parameters of the selected optimizer.
- **verbose** (*bool, default=True*) – Whether to print progress messages to stdout.

Examples

```
>>> from reflame import FlnnClassifier, Data
>>> from sklearn.datasets import make_regression
>>>
>>> ## Make dataset
>>> X, y = make_regression(n_samples=200, n_features=10, random_state=1)
>>> ## Load data object
>>> data = Data(X, y)
>>> ## Split train and test
>>> data.split_train_test(test_size=0.2, random_state=1, inplace=True)
>>> ## Scale dataset
>>> data.X_train, scaler = data.scale(data.X_train, scaling_methods=("minmax"))
>>> data.X_test = scaler.transform(data.X_test)
```

(continues on next page)

(continued from previous page)

```
>>> ## Create model
>>> model = FlnnClassifier(expand_name="chebyshev", n_funcs=4, act_name="none",
>>>                        obj_name="CEL", max_epochs=100, batch_size=32,␣
↪optimizer="SGD", verbose=True)
>>> ## Train the model
>>> model.fit(data.X_train, data.y_train)
>>> ## Test the model
>>> y_pred = model.predict(data.X_test)
>>> ## Calculate some metrics
>>> print(model.score(X=data.X_test, y=data.y_test, method="RMSE"))
>>> print(model.scores(X=data.X_test, y=data.y_test, list_methods=["R2", "NSE",
↪"MAPE"]))
>>> print(model.evaluate(y_true=data.y_test, y_pred=y_pred, list_metrics=["R2", "NSE
↪", "MAPE", "NNSE"]))
```

CLS_OBJ_BINARY_1 = ['PNLLL', 'HEL', 'BCEL', 'CEL', 'BCELL']

CLS_OBJ_BINARY_2 = ['NLLL']

CLS_OBJ_LOSSES = ['CEL', 'HEL', 'KLDL']

CLS_OBJ_MULTI = ['NLLL', 'CEL']

SUPPORTED_LOSSES = {'BCEL': <class 'torch.nn.modules.loss.BCELoss'>, 'BCELL': <class 'torch.nn.modules.loss.BCEWithLogitsLoss'>, 'CEL': <class 'torch.nn.modules.loss.CrossEntropyLoss'>, 'GNLLL': <class 'torch.nn.modules.loss.GaussianNLLLoss'>, 'HEL': <class 'torch.nn.modules.loss.HingeEmbeddingLoss'>, 'KLDL': <class 'torch.nn.modules.loss.KLDivLoss'>, 'NLLL': <class 'torch.nn.modules.loss.NLLLoss'>, 'PNLLL': <class 'torch.nn.modules.loss.PoissonNLLLoss'>}

create_network(X, y) → Tuple[NeuralNetClassifier, *ObjectiveScaler*]

Returns

- **network** (*FLNN, an instance of FLNN network*)
- **obj_scaler** (*ObjectiveScaler, the objective scaler that used to scale output*)

evaluate(*y_true*, *y_pred*, *list_metrics=('AS', 'RS')*)

Return the list of performance metrics on the given test data and labels.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Predicted values for *X*.
- **list_metrics** (*list, default=("AS", "RS")*) – You can get metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns

results – The results of the list metrics

Return type

dict

fit(*X*, *y*)

score(*X*, *y*, *method='AS'*)

Return the metric on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for *X*.
- **method** (*str, default="AS"*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns
result – The result of selected metric

Return type
float

scores(*X*, *y*, *list_methods=('AS', 'RS')*)

Return the list of metrics on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for *X*.
- **list_methods** (*list, default=("AS", "RS")*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns
results – The results of the list metrics

Return type
dict

set_predict_request(**, return_prob: bool | None | str = '$UNCHANGED$'*) → *FlnnClassifier*

Request metadata passed to the `predict` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

return_prob (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `return_prob` parameter in `predict`.

Returns

self – The updated object.

Return type

object

`set_score_request`(*, *method: bool | None | str = '$UNCHANGED$'*) → *FlnnClassifier*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

method (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `method` parameter in `score`.

Returns

self – The updated object.

Return type

object

class reflame.model.standard_flnn.**FlnnRegressor**(*expand_name='chebyshev'*, *n_funcs=4*, *act_name='none'*, *obj_name='MSE'*, *max_epochs=1000*, *batch_size=32*, *optimizer='SGD'*, *optimizer_paras=None*, *verbose=False*, ***kwargs*)

Bases: *BaseFlnn*

Defines the class for traditional FLNN network for Regression problems that inherit the BaseFlnn and RegressorMixin classes.

Parameters

- **expand_name** (*str, default="chebyshev"*) – The expand function that will be used. The supported expand functions are: {"chebyshev", "legendre", "gegenbauer", "laguerre", "hermite", "power", "trigonometric"}
- **n_funcs** (*int, default=4*) – The first *n_funcs* in expand functions list will be used. Valid value from 1 to 10.
- **act_name** (*{"none", "relu", "leaky_relu", "celu", "prelu", "gelu", "elu", "selu", "rrelu", "tanh", "hard_tanh", "sigmoid",}*) – "hard_sigmoid", "log_sigmoid", "silu", "swish", "hard_swish", "soft_plus", "mish", "soft_sign", "tanh_shrink", "soft_shrink", "hard_shrink", "softmin", "softmax", "log_softmax" }, default='none' Activation function for the hidden layer.
- **obj_name** (*str, default=None*) – The name of objective for the problem, also depend on the problem is classification and regression.
- **max_epochs** (*int, default=1000*) – Maximum number of epochs / iterations / generations
- **batch_size** (*int, default=32*) – The batch size
- **optimizer** (*str, default = "SGD"*) – The gradient-based optimizer from Pytorch. List of supported optimizer is: ["Adadelta", "Adagrad", "Adam", "Adamax", "AdamW", "ASGD", "LBFGS", "NAdam", "RAdam", "RMSprop", "Rprop", "SGD"]
- **optimizer_paras** (*dict or None, default=None*) – The dictionary parameters of the selected optimizer.
- **verbose** (*bool, default=True*) – Whether to print progress messages to stdout.

Examples

```
>>> from reflame import FlnnRegressor, Data
>>> from sklearn.datasets import make_regression
>>>
>>> ## Make dataset
>>> X, y = make_regression(n_samples=200, n_features=10, random_state=1)
>>> ## Load data object
>>> data = Data(X, y)
>>> ## Split train and test
>>> data.split_train_test(test_size=0.2, random_state=1, inplace=True)
>>> ## Scale dataset
>>> data.X_train, scaler = data.scale(data.X_train, scaling_methods=("minmax"))
>>> data.X_test = scaler.transform(data.X_test)
>>> ## Create model
>>> model = FlnnRegressor(expand_name="chebyshev", n_funcs=4, act_name="none",
```

(continues on next page)

(continued from previous page)

```
>>>                          obj_name="MSE", max_epochs=100, batch_size=32,␣
↪optimizer="SGD", verbose=True)
>>> ## Train the model
>>> model.fit(data.X_train, data.y_train)
>>> ## Test the model
>>> y_pred = model.predict(data.X_test)
>>> ## Calculate some metrics
>>> print(model.score(X=data.X_test, y=data.y_test, method="RMSE"))
>>> print(model.scores(X=data.X_test, y=data.y_test, list_methods=["R2", "NSE",
↪"MAPE"]))
>>> print(model.evaluate(y_true=data.y_test, y_pred=y_pred, list_metrics=["R2", "NSE
↪", "MAPE", "NNSE"]))
```

SUPPORTED_LOSSES = {'MAE': <class 'torch.nn.modules.loss.L1Loss'>, 'MSE': <class 'torch.nn.modules.loss.MSELoss'>}

create_network(*X*, *y*)

Returns

- **network** (*FLNN, an instance of FLNN network*)
- **obj_scaler** (*ObjectiveScaler, the objective scaler that used to scale output*)

evaluate(*y_true*, *y_pred*, *list_metrics=('MSE', 'MAE')*)

Return the list of performance metrics of the prediction.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Predicted values for *X*.
- **list_metrics** (*list, default=("MSE", "MAE")*) – You can get metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns

results – The results of the list metrics

Return type

dict

score(*X*, *y*, *method='RMSE'*)

Return the metric of the prediction.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **method** (*str, default="RMSE"*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns
result – The result of selected metric

Return type
float

`scores`(*X*, *y*, *list_methods=('MSE', 'MAE')*)

Return the list of metrics of the prediction.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **list_methods** (*list, default=("MSE", "MAE")*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns
results – The results of the list metrics

Return type
dict

`set_predict_request`(**, return_prob: bool | None | str = '$UNCHANGED$'*) → *FlnnRegressor*

Request metadata passed to the `predict` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
return_prob (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `return_prob` parameter in `predict`.

Returns
self – The updated object.

Return type
object

`set_score_request`(*, *method: bool | None | str = '$UNCHANGED$'*) → *FlnnRegressor*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
method (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `method` parameter in `score`.

Returns
self – The updated object.

Return type
object

3.2 reflame.utils package

3.2.1 reflame.utils.activation module

`reflame.utils.activation.`**`celu`**(*x*, *alpha=1.0*)

`reflame.utils.activation.`**`elu`**(*x*, *alpha=1*)

`reflame.utils.activation.`**`gelu`**(*x*, *alpha=0.044715*)

`reflame.utils.activation.`**`hard_shrink`**(*x*, *alpha=0.5*)

`reflame.utils.activation.`**`hard_sigmoid`**(*x*, *lower=-2.5*, *upper=2.5*)

`reflame.utils.activation.`**`hard_swish`**(*x*, *lower=-3.0*, *upper=3.0*)

`reflame.utils.activation.`**`hard_tanh`**(*x*, *lower=-1.0*, *upper=1.0*)

reflame.utils.activation.**leaky_relu**(*x*, *alpha=0.01*)

reflame.utils.activation.**log_sigmoid**(*x*)

reflame.utils.activation.**log_softmax**(*x*)

reflame.utils.activation.**mish**(*x*, *beta=1.0*)

reflame.utils.activation.**none**(*x*)

reflame.utils.activation.**prelu**(*x*, *alpha=0.5*)

reflame.utils.activation.**relu**(*x*)

reflame.utils.activation.**rrelu**(*x*, *lower=0.125*, *upper=0.3333333333333333*)

reflame.utils.activation.**selu**(*x*, *alpha=1.67326324*, *scale=1.05070098*)

reflame.utils.activation.**sigmoid**(*x*)

reflame.utils.activation.**silu**(*x*)

reflame.utils.activation.**soft_plus**(*x*, *beta=1.0*)

reflame.utils.activation.**soft_shrink**(*x*, *alpha=0.5*)

reflame.utils.activation.**soft_sign**(*x*)

reflame.utils.activation.**softmax**(*x*)

reflame.utils.activation.**softmin**(*x*)

reflame.utils.activation.**swish**(*x*)

reflame.utils.activation.**tanh**(*x*)

reflame.utils.activation.**tanh_shrink**(*x*)

3.2.2 reflame.utils.data_toolkit module

class reflame.utils.data_toolkit.**BoxCoxScaler**(*lmbda=None*)

Bases: BaseEstimator, TransformerMixin

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

inverse_transform(*X*)

transform(*X*)

class reflame.utils.data_toolkit.**Data**(*X=None*, *y=None*, *name='Unknown'*)

Bases: object

The structure of our supported Data class

Parameters

- **X** (*np.ndarray*) – The features of your data
- **y** (*np.ndarray*) – The labels of your data

SUPPORT = {'scaler': ['standard', 'minmax', 'max-abs', 'log1p', 'loge', 'sqrt', 'sinh-arc-sinh', 'robust', 'box-cox', 'yeo-johnson']}

static check_y(*y*)

static encode_label(*y*)

static scale(*X*, *scaling_methods=('standard',)*, *list_dict_paras=None*)

set_train_test(*X_train=None*, *y_train=None*, *X_test=None*, *y_test=None*)

Function use to set your own X_train, y_train, X_test, y_test in case you don't want to use our split function

Parameters

- **X_train** (*np.ndarray*) –
- **y_train** (*np.ndarray*) –
- **X_test** (*np.ndarray*) –
- **y_test** (*np.ndarray*) –

split_train_test(*test_size=0.2*, *train_size=None*, *random_state=41*, *shuffle=True*, *stratify=None*, *inplace=True*)

The wrapper of the split_train_test function in scikit-learn library.

class reflame.utils.data_toolkit.**DataTransformer**(*scaling_methods=('standard',)*, *list_dict_paras=None*)

Bases: BaseEstimator, TransformerMixin

SUPPORTED_SCALERS = {'box-cox': <class 'reflame.utils.data_toolkit.BoxCoxScaler'>, 'log1p': <class 'reflame.utils.data_toolkit.Log1pScaler'>, 'loge': <class 'reflame.utils.data_toolkit.LogeScaler'>, 'max-abs': <class 'sklearn.preprocessing._data.MaxAbsScaler'>, 'minmax': <class 'sklearn.preprocessing._data.MinMaxScaler'>, 'robust': <class 'sklearn.preprocessing._data.RobustScaler'>, 'sinh-arc-sinh': <class 'reflame.utils.data_toolkit.SinhArcSinhScaler'>, 'sqrt': <class 'reflame.utils.data_toolkit.SqrtScaler'>, 'standard': <class 'sklearn.preprocessing._data.StandardScaler'>, 'yeo-johnson': <class 'reflame.utils.data_toolkit.YeoJohnsonScaler'>}

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

inverse_transform(*X*)

transform(*X*)

class reflame.utils.data_toolkit.**FeatureEngineering**

Bases: object

create_threshold_binary_features(*X*, *threshold*)

Perform feature engineering to add binary indicator columns for values below the threshold. Add each new column right after the corresponding original column.

Args: X (numpy.ndarray): The input 2D matrix of shape (n_samples, n_features). threshold (float): The threshold value for identifying low values.

Returns: numpy.ndarray: The updated 2D matrix with binary indicator columns.

class reflame.utils.data_toolkit.**LabelEncoder**

Bases: object

Encode categorical features as integer labels.

static check_y(*y*)

fit(*y*)

Fit label encoder to a given set of labels.

3.2.2.1 Parameters:

y

[array-like] Labels to encode.

fit_transform(*y*)

Fit label encoder and return encoded labels.

Parameters
y (*array-like of shape (n_samples,)*) – Target values.

Returns
y – Encoded labels.

Return type
array-like of shape (n_samples,)

inverse_transform(*y*)

Transform integer labels to original labels.

3.2.2.2 Parameters:

y

[array-like] Encoded integer labels.

3.2.2.3 Returns:

original_labels

[array-like] Original labels.

transform(*y*)

Transform labels to encoded integer labels.

3.2.2.4 Parameters:

y

[array-like (1-D vector)] Labels to encode.

3.2.2.5 Returns:

encoded_labels
[array-like] Encoded integer labels.

class reflame.utils.data_toolkit.**Log1pScaler**

Bases: BaseEstimator, TransformerMixin

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

inverse_transform(*X*)

transform(*X*)

class reflame.utils.data_toolkit.**LogeScaler**

Bases: BaseEstimator, TransformerMixin

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

inverse_transform(*X*)

transform(*X*)

class reflame.utils.data_toolkit.**ObjectiveScaler**(*obj_name='sigmoid'*, *ohe_scaler=None*)

Bases: object

For label scaler in classification (binary and multiple classification)

inverse_transform(*data*)

transform(*data*)

class reflame.utils.data_toolkit.**SinhArcSinhScaler**(*epsilon=0.1*, *delta=1.0*)

Bases: BaseEstimator, TransformerMixin

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

inverse_transform(*X*)

transform(*X*)

class reflame.utils.data_toolkit.**SqrtScaler**

Bases: BaseEstimator, TransformerMixin

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

inverse_transform(*X*)

transform(*X*)

class reflame.utils.data_toolkit.**TimeSeriesDifferencer**(*interval=1*)

Bases: object

difference(*X*)

inverse_difference(*diff_data*)

class reflame.utils.data_toolkit.**YeoJohnsonScaler**(*lmbda=None*)

Bases: BaseEstimator, TransformerMixin

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

inverse_transform(*X*)

transform(*X*)

3.2.3 reflame.utils.evaluator module

reflame.utils.evaluator.**get_all_classification_metrics**()

reflame.utils.evaluator.**get_all_regression_metrics**()

reflame.utils.evaluator.**get_metrics**(*problem*, *y_true*, *y_pred*, *metrics=None*, *testcase='test'*)

3.2.4 reflame.utils.expand_util module

reflame.utils.expand_util.**expand_chebyshev**(*x*, *n_funcs=5*)

reflame.utils.expand_util.**expand_gegenbauer**(*x*, *n_funcs=5*, *a=1.0*)

reflame.utils.expand_util.**expand_hermite**(*x*, *n_funcs=5*)

reflame.utils.expand_util.**expand_laguerre**(*x*, *n_funcs=5*)

reflame.utils.expand_util.**expand_legendre**(*x*, *n_funcs=5*)

reflame.utils.expand_util.**expand_power**(*x*, *n_funcs=5*)

reflame.utils.expand_util.**expand_trigonometric**(*x*, *n_funcs=5*, *a0=1.0*)

3.2.5 reflame.utils.validator module

reflame.utils.validator.**check_bool**(*name: str*, *value: bool*, *bound=(True, False)*)

reflame.utils.validator.**check_float**(*name: str*, *value: int*, *bound=None*)

reflame.utils.validator.**check_int**(*name: str*, *value: int*, *bound=None*)

reflame.utils.validator.**check_str**(*name: str*, *value: str*, *bound=None*)

reflame.utils.validator.**check_tuple_float**(*name: str*, *values: tuple*, *bounds=None*)

reflame.utils.validator.**check_tuple_int**(*name: str*, *values: tuple*, *bounds=None*)

reflame.utils.validator.**is_in_bound**(*value*, *bound*)

reflame.utils.validator.**is_str_in_list**(*value: str*, *my_list: list*)

3.3 Submodules

3.4 reflame.base_flnn module

class reflame.base_flnn.**BaseFlnn**(*expand_name='chebyshev'*, *n_funcs=4*, *act_name='none'*)

Bases: BaseEstimator

Defines the most general class for FLNN network that inherits the BaseEstimator class of Scikit-Learn library.

Parameters

- **expand_name** (*str, default="chebyshev"*) – The expand function that will be used. The supported expand functions are: {"chebyshev", "legendre", "gegenbauer", "laguerre", "hermite", "power", "trigonometric"}
- **n_funcs** (*int, default=4*) – The first *n_funcs* in expand functions list will be used. Valid value from 1 to 10.
- **act_name** (*str, default='none'*) – Activation function for the hidden layer. The supported activation functions are: {"none", "relu", "prelu", "gelu", "elu", "selu", "rrelu", "tanh", "hard_tanh", "sigmoid", "hard_sigmoid", "swish", "hard_swish", "soft_plus", "mish", "soft_sign", "tanh_shrink", "soft_shrink", "hard_shrink"}

CLS_OBJ_LOSSES = None

SUPPORTED_CLS_METRICS = {'AS': 'max', 'BSL': 'min', 'CEL': 'min', 'CKS': 'max', 'F1S': 'max', 'F2S': 'max', 'FBS': 'max', 'GINI': 'min', 'GMS': 'max', 'HL': 'min', 'HS': 'max', 'JSI': 'max', 'KLDL': 'min', 'LS': 'max', 'MCC': 'max', 'NPV': 'max', 'PS': 'max', 'ROC-AUC': 'max', 'RS': 'max', 'SS': 'max'}

SUPPORTED_REG_METRICS = {'A10': 'max', 'A20': 'max', 'A30': 'max', 'ACOD': 'max', 'APCC': 'max', 'AR': 'max', 'AR2': 'max', 'CI': 'max', 'COD': 'max', 'COR': 'max', 'COV': 'max', 'CRM': 'min', 'DRV': 'min', 'EC': 'max', 'EVS': 'max', 'GINI': 'min', 'GINI_WIKI': 'min', 'JSD': 'min', 'KGE': 'max', 'MAAPE': 'min', 'MAE': 'min', 'MAPE': 'min', 'MASE': 'min', 'ME': 'min', 'MRB': 'min', 'MRE': 'min', 'MSE': 'min', 'MSLE': 'min', 'MedAE': 'min', 'NNSE': 'max', 'NRMSE': 'min', 'NSE': 'max', 'OI': 'max', 'PCC': 'max', 'PCD': 'max', 'R': 'max', 'R2': 'max', 'R2S': 'max', 'RAE': 'min', 'RMSE': 'min', 'RSE': 'min', 'RSQ': 'max', 'SMAPE': 'min', 'VAF': 'max', 'WI': 'max'}

create_network(*X*, *y*)

evaluate(*y_true*, *y_pred*, *list_metrics=None*)

Return the list of performance metrics of the prediction.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Predicted values for *X*.
- **list_metrics** (*list*) – You can get metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns

results – The results of the list metrics

Return type
dict

fit(*X*, *y*)

static load_model(*load_path='history'*, *filename='model.pkl'*)

predict(*X*, *return_prob=False*)

Inherit the predict function from BaseFlnn class, with 1 more parameter *return_prob*.

Parameters

- **X** (*{array-like, sparse matrix} of shape (n_samples, n_features)*) – The input data.
- **return_prob** (*bool, default=False*) – It is used for classification problem:
  - If True, the returned results are the probability for each sample
  - If False, the returned results are the predicted labels

save_loss_train(*save_path='history'*, *filename='loss.csv'*)

Save the loss (convergence) during the training process to csv file.

Parameters

- **save_path** (*saved path (relative path, consider from current executed script path)*) –
- **filename** (*name of the file, needs to have ".csv" extension*) –

save_metrics(*y_true*, *y_pred*, *list_metrics=('RMSE', 'MAE')*, *save_path='history'*, *filename='metrics.csv'*)

Save evaluation metrics to csv file

Parameters

- **y_true** (*ground truth data*) –
- **y_pred** (*predicted output*) –
- **list_metrics** (*list of evaluation metrics*) –
- **save_path** (*saved path (relative path, consider from current executed script path)*) –
- **filename** (*name of the file, needs to have ".csv" extension*) –

save_model(*save_path='history'*, *filename='model.pkl'*)

Save model to pickle file

Parameters

- **save_path** (*saved path (relative path, consider from current executed script path)*) –
- **filename** (*name of the file, needs to have ".pkl" extension*) –

save_y_predicted(*X*, *y_true*, *save_path='history'*, *filename='y_predicted.csv'*)

Save the predicted results to csv file

Parameters

- **X** (*The features data, nd.ndarray*) –
- **y_true** (*The ground truth data*) –

- **save_path** (*saved path (relative path, consider from current executed script path)*) –
- **filename** (*name of the file, needs to have ".csv" extension*) –

score(*X*, *y*, *method=None*)

Return the metric of the prediction.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **method** (*str, default="RMSE"*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns

result – The result of selected metric

Return type

float

scores(*X*, *y*, *list_methods=None*)

Return the list of metrics of the prediction.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **list_methods** (*list, default=("MSE", "MAE")*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns

results – The results of the list metrics

Return type

dict

set_predict_request(*, *return_prob: bool | None | str = '$UNCHANGED$'*) → *BaseFlnn*

Request metadata passed to the `predict` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.

- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
return_prob (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `return_prob` parameter in `predict`.

Returns
self – The updated object.

Return type
object

`set_score_request`(*, *method: bool | None | str = '$UNCHANGED$'*) → *BaseFlnn*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
method (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `method` parameter in `score`.

Returns
self – The updated object.

Return type
object

class reflame.base_flnn.**BaseMhaFlnn**(*expand_name='chebyshev'*, *n_funcs=4*, *act_name='none'*, *obj_name=None*, *optimizer='BaseGA'*, *optimizer_paras=None*, *verbose=True*)

Bases: *BaseFlnn*

Defines the most general class for Metaheuristic-based FLNN model that inherits the BaseFlnn class

Parameters

- **expand_name** (*str, default="chebyshev"*) – The expand function that will be used. The supported expand functions are: {"chebyshev", "legendre", "gegenbauer", "laguerre", "hermite", "power", "trigonometric"}
- **n_funcs** (*int, default=4*) – The first *n_funcs* in expand functions list will be used. Valid value from 1 to 10.
- **act_name** (*str, default='none'*) – Activation function for the hidden layer. The supported activation functions are: {"none", "relu", "prelu", "gelu", "elu", "selu", "rrelu", "tanh", "hard_tanh", "sigmoid", "hard_sigmoid", "swish", "hard_swish", "soft_plus", "mish", "soft_sign", "tanh_shrink", "soft_shrink", "hard_shrink"}
- **obj_name** (*None or str, default=None*) – The name of objective for the problem, also depend on the problem is classification and regression.
- **optimizer** (*str or instance of Optimizer class (from Mealpy library), default = "BaseGA"*) – The Metaheuristic Algorithm that use to solve the feature selection problem. Current supported list, please check it here: https://github.com/thieu1995/mealpy. If a custom optimizer is passed, make sure it is an instance of *Optimizer* class.
- **optimizer_paras** (*None or dict of parameter, default=None*) – The parameter for the *optimizer* object. If *None*, the default parameters of optimizer is used (defined in https://github.com/thieu1995/mealpy.) If *dict* is passed, make sure it has at least *epoch* and *pop_size* parameters.
- **verbose** (*bool, default=True*) – Whether to print progress messages to stdout.

SUPPORTED_CLS_OBJECTIVES = {'AS': 'max', 'BSL': 'min', 'CEL': 'min', 'CKS': 'max', 'F1S': 'max', 'F2S': 'max', 'FBS': 'max', 'GINI': 'min', 'GMS': 'max', 'HL': 'min', 'HS': 'max', 'JSI': 'max', 'KLDL': 'min', 'LS': 'max', 'MCC': 'max', 'NPV': 'max', 'PS': 'max', 'ROC-AUC': 'max', 'RS': 'max', 'SS': 'max'}

SUPPORTED_OPTIMIZERS = ['OriginalABC', 'OriginalACOR', 'AugmentedAEO', 'EnhancedAEO', 'ImprovedAEO', 'ModifiedAEO', 'OriginalAEO', 'MGTO', 'OriginalAGTO', 'DevALO', 'OriginalALO', 'OriginalAO', 'OriginalAOA', 'IARO', 'LARO', 'OriginalARO', 'OriginalASO', 'OriginalAVOA', 'OriginalArchOA', 'AdaptiveBA', 'DevBA', 'OriginalBA', 'DevBBO', 'OriginalBBO', 'OriginalBBOA', 'OriginalBES', 'ABFO', 'OriginalBFO', 'OriginalBMO', 'DevBRO', 'OriginalBRO', 'OriginalBSA', 'ImprovedBSO', 'OriginalBSO', 'CleverBookBeesA', 'OriginalBeesA', 'ProbBeesA', 'OriginalCA', 'OriginalCDO', 'OriginalCEM', 'OriginalCGO', 'DevCHIO', 'OriginalCHIO', 'OriginalCOA', 'OCRO', 'OriginalCRO', 'OriginalCSA', 'OriginalCSO', 'OriginalCircleSA', 'OriginalCoatiOA', 'JADE', 'OriginalDE', 'SADE', 'SAP_DE', 'DevDMOA', 'OriginalDMOA', 'OriginalDO', 'DevEFO', 'OriginalEFO', 'OriginalEHO', 'AdaptiveEO', 'ModifiedEO', 'OriginalEO', 'OriginalEOA', 'LevyEP', 'OriginalEP', 'CMA_ES', 'LevyES', 'OriginalES', 'Simple_CMA_ES', 'OriginalESOA', 'OriginalEVO', 'OriginalFA', 'DevFBIO', 'OriginalFBIO', 'OriginalFFA', 'OriginalFFO', 'OriginalFLA', 'DevFOA', 'OriginalFOA', 'WhaleFOA', 'OriginalFOX', 'OriginalFPA', 'BaseGA', 'EliteMultiGA', 'EliteSingleGA', 'MultiGA', 'SingleGA', 'OriginalGBO', 'DevGCO', 'OriginalGCO', 'OriginalGJO', 'OriginalGOA', 'DevGSKA', 'OriginalGSKA', 'Matlab101GTO', 'Matlab102GTO', 'OriginalGTO', 'GWO_WOA', 'IGWO', 'OriginalGWO', 'RW_GWO', 'OriginalHBA', 'OriginalHBO', 'OriginalHC', 'SwarmHC', 'OriginalHCO', 'OriginalHGS', 'OriginalHGSO', 'OriginalHHO', 'DevHS', 'OriginalHS', 'OriginalICA', 'OriginalINFO', 'OriginalIWO', 'DevJA', 'LevyJA', 'OriginalJA', 'DevLCO', 'ImprovedLCO', 'OriginalLCO', 'OriginalMA', 'OriginalMFO', 'OriginalMGO', 'OriginalMPA', 'OriginalMRFO', 'WMQIMRFO', 'OriginalMSA', 'DevMVO', 'OriginalMVO', 'OriginalNGO', 'ImprovedNMRA', 'OriginalNMRA', 'OriginalNRO', 'OriginalOOA', 'OriginalPFA', 'OriginalPOA', 'AIW_PSO', 'CL_PSO', 'C_PSO', 'HPSO_TVAC', 'LDW_PSO', 'OriginalPSO', 'P_PSO', 'OriginalPSS', 'DevQSA', 'ImprovedQSA', 'LevyQSA', 'OppoQSA', 'OriginalQSA', 'OriginalRIME', 'OriginalRUN', 'GaussianSA', 'OriginalSA', 'SwarmSA', 'DevSARO', 'OriginalSARO', 'DevSBO', 'OriginalSBO', 'DevSCA', 'OriginalSCA', 'QleSCA', 'OriginalSCSO', 'ImprovedSFO', 'OriginalSFO', 'L_SHADE', 'OriginalSHADE', 'OriginalSHIO', 'OriginalSHO', 'ImprovedSLO', 'ModifiedSLO', 'OriginalSLO', 'DevSMA', 'OriginalSMA', 'DevSOA', 'OriginalSOA', 'OriginalSOS', 'DevSPBO', 'OriginalSPBO', 'OriginalSRSR', 'DevSSA', 'OriginalSSA', 'OriginalSSDO', 'OriginalSSO', 'OriginalSSpiderA', 'OriginalSSpiderO', 'OriginalSTO', 'OriginalSeaHO', 'OriginalServalOA', 'OriginalTDO', 'DevTLO', 'ImprovedTLO', 'OriginalTLO', 'OriginalTOA', 'DevTPO', 'OriginalTS', 'OriginalTSA', 'OriginalTSO', 'EnhancedTWO', 'LevyTWO', 'OppoTWO', 'OriginalTWO', 'DevVCS', 'OriginalVCS', 'OriginalWCA', 'OriginalWDO', 'OriginalWHO', 'HI_WOA', 'OriginalWOA', 'OriginalWaOA', 'OriginalWarSO', 'OriginalZOA']

SUPPORTED_REG_OBJECTIVES = {'A10': 'max', 'A20': 'max', 'A30': 'max', 'ACOD': 'max', 'APCC': 'max', 'AR': 'max', 'AR2': 'max', 'CI': 'max', 'COD': 'max', 'COR': 'max', 'COV': 'max', 'CRM': 'min', 'DRV': 'min', 'EC': 'max', 'EVS': 'max', 'GINI': 'min', 'GINI_WIKI': 'min', 'JSD': 'min', 'KGE': 'max', 'MAAPE': 'min', 'MAE': 'min', 'MAPE': 'min', 'MASE': 'min', 'ME': 'min', 'MRB': 'min', 'MRE': 'min', 'MSE': 'min', 'MSLE': 'min', 'MedAE': 'min', 'NNSE': 'max', 'NRMSE': 'min', 'NSE': 'max', 'OI': 'max', 'PCC': 'max', 'PCD': 'max', 'R': 'max', 'R2': 'max', 'R2S': 'max', 'RAE': 'min', 'RMSE': 'min', 'RSE': 'min', 'RSQ': 'max', 'SMAPE': 'min', 'VAF': 'max', 'WI': 'max'}

fit(*X*, *y*, *lb=(-1.0,)*, *ub=(1.0,)*, *save_population=False*)

objective_function(*solution=None*)

set_fit_request(**, lb: bool | None | str = '$UNCHANGED$', save_population: bool | None | str = '$UNCHANGED$', ub: bool | None | str = '$UNCHANGED$'*) → *BaseMhaFlnn*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

- **lb** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `lb` parameter in `fit`.
- **save_population** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `save_population` parameter in `fit`.
- **ub** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `ub` parameter in `fit`.

Returns

self – The updated object.

Return type

object

set_predict_request(*, *return_prob: bool | None | str = '$UNCHANGED$'*) → *BaseMhaFlnn*

Request metadata passed to the `predict` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters
return_prob (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for return_prob parameter in predict.

Returns
self – The updated object.

Return type
object

set_score_request(*, *method: bool | None | str = '$UNCHANGED$'*) → *BaseMhaFlnn*

Request metadata passed to the score method.

Note that this method is only relevant if enable_metadata_routing=True (see sklearn.set_config()). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to score if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to score.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters
method (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for method parameter in score.

Returns
self – The updated object.

Return type
object

class reflame.base_flnn.**FLNN**(*size_input=5*, *size_output=1*, *expand_name='chebyshev'*, *n_funcs=4*, *act_name='elu'*)

Bases: object

This class defines the general Functional Link Neural Network (FLNN) model

Parameters

- **size_input** (*int, default=5*) – The number of input features
- **size_output** (*int, default=1*) – The number of output labels
- **expand_name** (*str, default="chebyshev"*) – The expand function that will be used. The supported expand functions are: {"chebyshev", "legendre", "gegenbauer", "laguerre", "hermite", "power", "trigonometric"}
- **n_funcs** (*int, default=4*) – The first *n_funcs* in expand functions list will be used. Valid value from 1 to 10.
- **act_name** (*str, default='none'*) – Activation function for the hidden layer. The supported activation functions are: {"none", "relu", "prelu", "gelu", "elu", "selu", "rrelu", "tanh", "hard_tanh", "sigmoid", "hard_sigmoid", "swish", "hard_swish", "soft_plus", "mish", "soft_sign", "tanh_shrink", "soft_shrink", "hard_shrink"}

fit(*X*, *y*)

Fit the model to data matrix X and target(s) y.

Parameters

- **X** (*ndarray or sparse matrix of shape (n_samples, n_features)*) – The input data.
- **y** (*ndarray of shape (n_samples,) or (n_samples, n_outputs)*) – The target values (class labels in classification, real numbers in regression).

Returns
self – Returns a trained FLNN model.

Return type
object

get_weights()

get_weights_size()

predict(*X*)

Predict using the Extreme Learning Machine model.

Parameters
X(*{array-like, sparse matrix} of shape (n_samples, n_features)*) – The input data.

Returns
y – The predicted values.

Return type
ndarray of shape (n_samples, n_outputs)

set_weights(*weights*)

transform_X(*X*)

update_weights_from_solution(*solution*)

3.5 reflame.base_flnn_torch module

class reflame.base_flnn_torch.**BaseFlnn**(*expand_name='chebyshev'*, *n_funcs=4*, *act_name='none'*, *obj_name=None*, *max_epochs=1000*, *batch_size=32*, *optimizer='SGD'*, *optimizer_paras=None*, *verbose=False*)

Bases: BaseEstimator

Defines the most general class for FLNN network that inherits the BaseEstimator class of Scikit-Learn library.

Parameters

- **expand_name** (*str, default="chebyshev"*) – The expand function that will be used. The supported expand functions are: {"chebyshev", "legendre", "gegenbauer", "laguerre", "hermite", "power", "trigonometric"}
- **n_funcs** (*int, default=4*) – The first *n_funcs* in expand functions list will be used. Valid value from 1 to 10.
- **act_name** (*{"none", "relu", "leaky_relu", "celu", "prelu", "gelu", "elu", "selu", "rrelu", "tanh", "hard_tanh",}*) – "sigmoid", "hard_sigmoid", "log_sigmoid", "silu", "swish", "hard_swish", "soft_plus", "mish", "soft_sign", "tanh_shrink", "soft_shrink", "hard_shrink", "softmin", "softmax", "log_softmax" }, default='none' Activation function for the hidden layer.
- **obj_name** (*str, default=None*) – The name of objective for the problem, also depend on the problem is classification and regression.
- **max_epochs** (*int, default=1000*) – Maximum number of epochs / iterations / generations
- **batch_size** (*int, default=32*) – The batch size
- **optimizer** (*str, default = "SGD"*) – The gradient-based optimizer from Pytorch. List of supported optimizer is: ["Adadelta", "Adagrad", "Adam", "Adamax", "AdamW", "ASGD", "LBFGS", "NAdam", "RAdam", "RMSprop", "Rprop", "SGD"]
- **optimizer_paras** (*dict or None, default=None*) – The dictionary parameters of the selected optimizer.
- **verbose** (*bool, default=True*) – Whether to print progress messages to stdout.

CLS_OBJ_LOSSES = None

SUPPORTED_CLS_METRICS = {'AS': 'max', 'BSL': 'min', 'CEL': 'min', 'CKS': 'max', 'F1S': 'max', 'F2S': 'max', 'FBS': 'max', 'GINI': 'min', 'GMS': 'max', 'HL': 'min', 'HS': 'max', 'JSI': 'max', 'KLDL': 'min', 'LS': 'max', 'MCC': 'max', 'NPV': 'max', 'PS': 'max', 'ROC-AUC': 'max', 'RS': 'max', 'SS': 'max'}

SUPPORTED_LOSSES = {'MAE': <class 'torch.nn.modules.loss.L1Loss'>, 'MSE': <class 'torch.nn.modules.loss.MSELoss'>}

SUPPORTED_OPTIMIZERS = ['Adadelta', 'Adagrad', 'Adam', 'Adamax', 'AdamW', 'ASGD', 'LBFGS', 'NAdam', 'RAdam', 'RMSprop', 'Rprop', 'SGD']

```
SUPPORTED_REG_METRICS = {'A10':  'max', 'A20':  'max', 'A30':  'max', 'ACOD': 'max',
'APCC': 'max', 'AR': 'max', 'AR2':  'max', 'CI': 'max', 'COD': 'max', 'COR': 'max',
'COV': 'max', 'CRM': 'min', 'DRV': 'min', 'EC': 'max', 'EVS': 'max', 'GINI': 'min',
'GINI_WIKI': 'min', 'JSD': 'min', 'KGE': 'max', 'MAAPE': 'min', 'MAE': 'min',
'MAPE': 'min', 'MASE': 'min', 'ME': 'min', 'MRB': 'min', 'MRE': 'min', 'MSE': 'min',
'MSLE': 'min', 'MedAE': 'min', 'NNSE': 'max', 'NRMSE': 'min', 'NSE': 'max', 'OI':
'max', 'PCC': 'max', 'PCD': 'max', 'R': 'max', 'R2':  'max', 'R2S': 'max', 'RAE':
'min', 'RMSE': 'min', 'RSE': 'min', 'RSQ': 'max', 'SMAPE': 'min', 'VAF': 'max',
'WI': 'max'}
```

create_network(*X*, *y*)

evaluate(*y_true*, *y_pred*, *list_metrics=None*)

Return the list of performance metrics of the prediction.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Predicted values for *X*.
- **list_metrics** (*list*) – You can get metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns

results – The results of the list metrics

Return type

dict

fit(*X*, *y*)

static load_model(*load_path='history'*, *filename='model.pkl'*)

predict(*X*, *return_prob=False*)

Inherit the predict function from BaseFlnn class, with 1 more parameter *return_prob*.

Parameters

- **X** (*{array-like, sparse matrix} of shape (n_samples, n_features)*) – The input data.
- **return_prob** (*bool, default=False*) – It is used for classification problem:
  - If True, the returned results are the probability for each sample
  - If False, the returned results are the predicted labels

save_loss_train(*save_path='history'*, *filename='loss.csv'*)

Save the loss (convergence) during the training process to csv file.

Parameters

- **save_path** (*saved path (relative path, consider from current executed script path)*) –
- **filename** (*name of the file, needs to have ".csv" extension*) –

save_metrics(*y_true*, *y_pred*, *list_metrics=('RMSE', 'MAE')*, *save_path='history'*, *filename='metrics.csv'*)

Save evaluation metrics to csv file

Parameters

- **y_true** (*ground truth data*) –
- **y_pred** (*predicted output*) –
- **list_metrics** (*list of evaluation metrics*) –
- **save_path** (*saved path (relative path, consider from current executed script path)*) –
- **filename** (*name of the file, needs to have ".csv" extension*) –

save_model(*save_path='history'*, *filename='model.pkl'*)

Save model to pickle file

Parameters

- **save_path** (*saved path (relative path, consider from current executed script path)*) –
- **filename** (*name of the file, needs to have ".pkl" extension*) –

save_y_predicted(*X*, *y_true*, *save_path='history'*, *filename='y_predicted.csv'*)

Save the predicted results to csv file

Parameters

- **X** (*The features data, nd.ndarray*) –
- **y_true** (*The ground truth data*) –
- **save_path** (*saved path (relative path, consider from current executed script path)*) –
- **filename** (*name of the file, needs to have ".csv" extension*) –

score(*X*, *y*, *method=None*)

Return the metric of the prediction.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **method** (*str, default="RMSE"*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns

result – The result of selected metric

Return type

float

scores(*X*, *y*, *list_methods=None*)

Return the list of metrics of the prediction.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape (`n_samples, n_samples_fitted`), where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **list_methods** (*list, default=("MSE", "MAE")*) – You can get all metrics from Permetrics library: https://github.com/thieu1995/permetrics

Returns

results – The results of the list metrics

Return type

dict

set_predict_request(**, return_prob: bool | None | str = '$UNCHANGED$'*) → *BaseFlnn*

Request metadata passed to the `predict` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

return_prob (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `return_prob` parameter in `predict`.

Returns

self – The updated object.

Return type

object

set_score_request(*, *method: bool | None | str = '$UNCHANGED$'*) → *BaseFlnn*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

method (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `method` parameter in `score`.

Returns

self – The updated object.

Return type

object

class reflame.base_flnn_torch.**FLNN**(*size_input=10*, *size_output=1*, *expand_name='chebyshev'*, *n_funcs=4*, *act_name='none'*)

Bases: `Module`

SUPPORTED_ACTIVATIONS = ['none', 'threshold', 'relu', 'rrelu', 'hardtanh', 'relu6', 'sigmoid', 'hardsigmoid', 'tanh', 'silu', 'mish', 'hardswish', 'elu', 'celu', 'selu', 'glu', 'gelu', 'hardshrink', 'leakyrelu', 'logsigmoid', 'softplus', 'softshrink', 'multiheadattention', 'prelu', 'softsign', 'tanhshrink', 'softmin', 'softmax', 'logsoftmax']

SUPPORTED_EXPANDS = ['chebyshev', 'legendre', 'gegenbauer', 'laguerre', 'hermite', 'power', 'trigonometric']

SUPPORTED_N_FUNCS = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

transform_X(*X*)

CHAPTER

FOUR

CITATION REQUEST

Note:

```
If you want to understand how Metaheuristic is applied to Functional Link Neural Network,
↪ you need to read the paper
    titled "A resource usage prediction system using functional-link and genetic␣
↪algorithm neural network for multivariate cloud metrics".
    The paper can be accessed at the following `this link <https://doi.org/10.1016/j.
↪procs.2020.03.063>`_
```

Please include these citations if you plan to use this library:

```
@software{nguyen_van_thieu_2023_8249046,
  author       = {Nguyen Van Thieu},
  title        = {Revolutionizing Functional Link Neural Network by Metaheuristic␣
↪Algorithms: reflame - A Python Library},
  month        = 11,
  year         = 2023,
  publisher    = {Zenodo},
  doi          = {10.5281/zenodo.8249045},
  url          = {https://github.com/thieu1995/reflame}
}

@article{van2023mealpy,
  title={MEALPY: An open-source library for latest meta-heuristic algorithms in Python},
  author={Van Thieu, Nguyen and Mirjalili, Seyedali},
  journal={Journal of Systems Architecture},
  year={2023},
  publisher={Elsevier},
  doi={10.1016/j.sysarc.2023.102871}
}

@inproceedings{nguyen2019building,
        author = {Thieu Nguyen and Binh Minh Nguyen and Giang Nguyen},
        booktitle = {International Conference on Theory and Applications of Models of␣
↪Computation},
        organization = {Springer},
        pages = {501--517},
        title = {Building Resource Auto-scaler with Functional-Link Neural Network and␣
↪Adaptive Bacterial Foraging Optimization},
        year = {2019},
        url={https://doi.org/10.1007/978-3-030-14812-6_31},
```

(continues on next page)

(continued from previous page)

```
        doi={10.1007/978-3-030-14812-6_31}
}

@inproceedings{nguyen2018resource,
        author = {Thieu Nguyen and Nhuan Tran and Binh Minh Nguyen and Giang Nguyen},
        booktitle = {2018 IEEE 11th Conference on Service-Oriented Computing and␣
↪Applications (SOCA)},
        organization = {IEEE},
        pages = {49--56},
        title = {A Resource Usage Prediction System Using Functional-Link and Genetic␣
↪Algorithm Neural Network for Multivariate Cloud Metrics},
        year = {2018},
        url={https://doi.org/10.1109/SOCA.2018.00014},
        doi={10.1109/SOCA.2018.00014}
}
```

```

If you have an open-ended or a research question, you can contact me via nguyenthieu2102@gmail.com
```

CHAPTER

FIVE

IMPORTANT LINKS

- Official source code repo: https://github.com/thieu1995/reflame
- Official document: https://reflame.readthedocs.io/
- Download releases: https://pypi.org/project/reflame/
- Issue tracker: https://github.com/thieu1995/reflame/issues
- Notable changes log: https://github.com/thieu1995/reflame/blob/master/ChangeLog.md
- **This project also related to our another projects which are "optimization" and "machine learning", check it here:**
  - https://github.com/thieu1995/mealpy
  - https://github.com/thieu1995/metaheuristics
  - https://github.com/thieu1995/opfunu
  - https://github.com/thieu1995/enoppy
  - https://github.com/thieu1995/permetrics
  - https://github.com/thieu1995/MetaCluster
  - https://github.com/thieu1995/pfevaluator
  - https://github.com/thieu1995/intelelm
  - https://github.com/aiir-team

CHAPTER

SIX

LICENSE

The project is licensed under GNU General Public License (GPL) V3 license.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

F

G

H

I

L

M

N

O

P

R

S

T

U

Y