Python Extreme Learning Machine (ELM) Documentation

Release 0.1.1

Augusto Almeida

February 10, 2015

Contents

CHAPTER 1

Basics:

1.1 Python Extreme Learning Machine (ELM)

Python Extreme Learning Machine (ELM) is a machine learning technique used for classification/regression tasks.

- Free software: BSD license
- Documentation: https://elm.readthedocs.org.

1.1.1 Features

- ELM Kernel
- ELM Random Neurons
- MLTools

1.2 Installation

At the command line:

```
$ pip install elm
```

Or, if you have virtualenv installed:

```
$ virtualenv venv
$ source venv/bin/activate
$ pip install elm
```

Note: If you found an error while using `ELMKernel.search_param()` or `ELMRandom.search_param()`, probably is because **pip** installed an outdated version of **optunity** (currently their pip package and github are not synced).

To fix it, do:

```
# Download package
$ pip install -d . elm
# Unzip it
$ tar -xf elm*.tar.gz
```

```
# Install requirements.txt
$ cd elm*; pip install -r requirements.txt
```

1.3 Usage

To use Python Extreme Learning Machine (ELM) in a project:

```
import elm

# download an example dataset from
# https://github.com/acba/elm/tree/develop/tests/data


# load dataset
data = elm.read("iris.data")

# create a classifier
elmk = elm.ELMKernel()

# search for best parameter for this dataset
# define "kfold" cross-validation method, "accuracy" as a objective function
# to be optimized and perform 10 searching steps.
# best parameters will be saved inside 'elmk' object
elmk.search_param(data, cv="kfold", of="accuracy", eval=10)

# split data in training and testing sets
# use 80% of dataset to training and shuffle data before splitting
tr_set, te_set = elm.split_sets(data, training_percent=.8, perm=True)

#train and test
# results are Error objects
tr_result = elmk.train(tr_set)
te_result = elmk.test(te_set)

print(te_result.get_accuracy)
```

CHAPTER 2

API Reference

2.1 elm package

2.1.1 `elm.elmk` Module

This file contains ELMKernel classes and all developed methods.

class `elm.elmk.`**`ELMKernel`**(*params=[]*)

Bases: `elm.mltools.MLTools`

A Python implementation of ELM Kernel defined by Huang[1].

An ELM is a single-hidden layer feedforward network (SLFN) proposed by Huang back in 2006, in 2012 the author revised and introduced a new concept of using kernel functions to his previous work.

This implementation currently accepts both methods proposed at 2012, random neurons and kernel functions to estimate classifier/regression functions.

Let the dimensionality "d" of the problem be the sum of "t" size (number of targets per pattern) and "f" size (number of features per pattern). So, d = t + f

The data will be set as Pattern = (Target | Features).

If database has *N* patterns, its size follows *Nxd*.

Note: [1] Paper reference: Huang, 2012, "Extreme Learning Machine for Regression and Multiclass Classification"

Variables

- **output_weight** (*numpy.ndarray*) – a column vector (*Nx1*) calculated after training, represent :math:beta.
- **training_patterns** (*numpy.ndarray*) – a matrix (*Nxd*) containing all patterns used for training.

  Need to save all training patterns to perform kernel calculation at testing and prediction phase.
- **param_kernel_function** (*str*) – kernel function that will be used for training.
- **param_c** (*float*) – regularization coefficient (*C*) used for training.
- **param_kernel_params** (*list of float*) – kernel function parameters that will be used for training.

Other Parameters

- **regressor_name** (*str*) – The name of classifier/regressor.
- **available_kernel_functions** (*list of str*) – List with all available kernel functions.
- **default_param_kernel_function** (*str*) – Default kernel function if not set at class constructor.
- **default_param_c** (*float*) – Default parameter c value if not set at class constructor.
- **default_param_kernel_params** (*list of float*) – Default kernel function parameters if not set at class constructor.

Note:

•**regressor_name**: defaults to "elmk".

•**default_param_kernel_function**: defaults to "rbf".

•**default_param_c**: defaults to 9.

•**default_param_kernel_params**: defaults to [-15].

`__init__`(*params=[]*)

Class constructor.

Parameters **params** (*list*) – first argument (*str*) is an available kernel function, second argument (*float*) is the coefficient *C* of regularization and the third and last argument is a list of arguments for the kernel function.

Example

```
>>> import elm
>>> params = ["linear", 5, []]
>>> elmk = elm.ELMKernel(params)
```

`get_available_kernel_functions`()

Return available kernel functions.

`predict`(*horizon=1*)

Predict next targets based on previous training.

Parameters **horizon** (*int*) – number of predictions.

Returns numpy.ndarray: a column vector containing all predicted targets.

`print_parameters`()

Print parameters values.

`search_param`(*database*, *dataprocess=None*, *path_filename=(u'', u'')*, *save=False*, *cv=u'ts'*, *of=u'rmse'*, *kf=None*, *eval=50*)

Search best hyperparameters for classifier/regressor based on optunity algorithms.

Parameters

- **database** (*numpy.ndarray*) – a matrix containing all patterns that will be used for training/testing at some cross-validation method.
- **dataprocess** (*DataProcess*) – an object that will pre-process database before training. Defaults to None.

- **path_filename** (*tuple*) – *TODO*.
- **save** (*bool*) – *TODO*.
- **cv** (*str*) – Cross-validation method. Defaults to "ts".
- **of** (*str*) – Objective function to be minimized at optunity.minimize. Defaults to "rmse".
- **kf** (*list of str*) – a list of kernel functions to be used by the search. Defaults to None, this set all available functions.
- **eval** (*int*) – Number of steps (evaluations) to optunity algorithm.

Each set of hyperparameters will perform a cross-validation method chosen by param cv.

Available *cv* methods:

- **"ts"** `mltools.time_series_cross_validation()` Perform a time-series cross-validation suggested by Hydman.
- **"kfold"** `mltools.kfold_cross_validation()` Perform a k-fold cross-validation.

Available *of* function:

- "accuracy", "rmse", "mape", "me".

See also:

http://optunity.readthedocs.org/en/latest/user/index.html

`test` (*testing_matrix*, *predicting=False*)
Calculate test predicted values based on previous training.

Parameters

- **testing_matrix** (*numpy.ndarray*) – a matrix containing all patterns that will be used for testing.
- **predicting** (*bool*) – Don't set.

Returns `Error`: testing error object containing expected, predicted targets and all error metrics.

Note: Testing matrix must have target variables as the first column.

`train` (*training_matrix*, *params=[ ]*)
Calculate output_weight values needed to test/predict data.

If params is provided, this method will use at training phase. Else, it will use the default value provided at object initialization.

Parameters

- **training_matrix** (*numpy.ndarray*) – a matrix containing all patterns that will be used for training.
- **params** (*list*) – a list of parameters defined at `ELMKernel.__init__()`

Returns `Error`: training error object containing expected, predicted targets and all error metrics.

Note: Training matrix must have target variables as the first column.

`train_iterative` (*database_matrix*, *params=[ ]*, *sliding_window=168*, *k=1*)
Training method used by Fred 09 paper.

2.1.2 `elm.elmr` Module

This file contains ELMKernel classes and all developed methods.

class `elm.elmr.`**`ELMRandom`**(*params=[ ]*)

Bases: `elm.mltools.MLTools`

A Python implementation of ELM Random Neurons defined by Huang[1].

An ELM is a single-hidden layer feedforward network (SLFN) proposed by Huang back in 2006, in 2012 the author revised and introduced a new concept of using kernel functions to his previous work.

This implementation currently accepts both methods proposed at 2012, random neurons and kernel functions to estimate classifier/regression functions.

Let the dimensionality "d" of the problem be the sum of "t" size (number of targets per pattern) and "f" size (number of features per pattern). So, d = t + f

The data will be set as Pattern = (Target | Features).

If database has *N* patterns, its size follows *Nxd*.

Note: [1] Paper reference: Huang, 2012, "Extreme Learning Machine for Regression and Multiclass Classification"

Variables

- **input_weight** (*numpy.ndarray*) – a random matrix (*Lxd-1*) needed to calculate H(**x**).
- **output_weight** (*numpy.ndarray*) – a column vector (*Nx1*) calculated after training, represent :math:beta.
- **bias_of_hidden_neurons** (*numpy.ndarray*) – a random column vector (*Lx1*) needed to calculate H(**x**).
- **param_function** (*str*) – function that will be used for training.
- **param_c** (*float*) – regularization coefficient (*C*) used for training.
- **param_l** (*list of float*) – number of neurons that will be used for training.
- **param_opt** (*bool*) – a boolean used to calculate an optimization when number of training patterns are much larger than neurons (N >> L).

Other Parameters

- **regressor_name** (*str*) – The name of classifier/regressor.
- **available_functions** (*list of str*) – List with all available functions.
- **default_param_function** (*str*) – Default function if not set at class constructor.
- **default_param_c** (*float*) – Default parameter c value if not set at class constructor.
- **default_param_l** (*integer*) – Default number of neurons if not set at class constructor.
- **default_param_opt** (*bool*) – Default boolean optimization flag.

Note:

- **regressor_name**: defaults to "elmr".
- **default_param_function**: defaults to "sigmoid".
- **default_param_c**: defaults to 2 ** -6.

•**default_param_l**: defaults to 500.

•**default_param_opt**: defaults to False.

__init__(*params=[ ]*)
Class constructor.

Parameters **params** (*list*) – first argument (*str*) is an available function, second argument (*float*) is the coefficient *C* of regularization, the third is the number of hidden neurons and the last argument is an optimization boolean.

Example

```
>>> import elm
>>> params = ["sigmoid", 1, 500, False]
>>> elmr = elm.ELMRandom(params)
```

get_available_functions()
Return available functions.

predict(*horizon=1*)
Predict next targets based on previous training.

Parameters **horizon** (*int*) – number of predictions.

Returns numpy.ndarray: a column vector containing all predicted targets.

print_parameters()
Print current parameters.

search_param(*database*, *dataprocess=None*, *path_filename=(u'', u'')*, *save=False*, *cv=u'ts'*, *of=u'rmse'*, *f=None*, *eval=50*)
Search best hyperparameters for classifier/regressor based on optunity algorithms.

Parameters

- **database** (*numpy.ndarray*) – a matrix containing all patterns that will be used for training/testing at some cross-validation method.
- **dataprocess** (*DataProcess*) – an object that will pre-process database before training. Defaults to None.
- **path_filename** (*tuple*) – *TODO*.
- **save** (*bool*) – *TODO*.
- **cv** (*str*) – Cross-validation method. Defaults to "ts".
- **of** (*str*) – Objective function to be minimized at optunity.minimize. Defaults to "rmse".
- **f** (*list of str*) – a list of functions to be used by the search. Defaults to None, this set all available functions.
- **eval** (*int*) – Number of steps (evaluations) to optunity algorithm.

Each set of hyperparameters will perform a cross-validation method chosen by param cv.

Available *cv* methods:

- **"ts"** `mltools.time_series_cross_validation()` Perform a time-series cross-validation suggested by Hydman.
- **"kfold"** `mltools.kfold_cross_validation()` Perform a k-fold cross-validation.

Available *of* function:

- "accuracy", "rmse", "mape", "me".

See also:

http://optunity.readthedocs.org/en/latest/user/index.html

test (*testing_matrix*, *predicting=False*)
Calculate test predicted values based on previous training.

Parameters

- **testing_matrix** (*numpy.ndarray*) – a matrix containing all patterns that will be used for testing.
- **predicting** (*bool*) – Don't set.

Returns `Error`: testing error object containing expected, predicted targets and all error metrics.

Note: Testing matrix must have target variables as the first column.

train (*training_matrix*, *params=*[ ])
Calculate output_weight values needed to test/predict data.

If params is provided, this method will use at training phase. Else, it will use the default value provided at object initialization.

Parameters

- **training_matrix** (*numpy.ndarray*) – a matrix containing all patterns that will be used for training.
- **params** (*list*) – a list of parameters defined at `ELMKernel.__init__()`

Returns `Error`: training error object containing expected, predicted targets and all error metrics.

Note: Training matrix must have target variables as the first column.

train_iterative (*database_matrix*, *params=*[ ], *sliding_window=168*, *k=1*)
Training method used by Fred 09 paper.

2.1.3 `elm.mltools` Module

This file contains MLTools class and all developed methods.

class `elm.mltools.`**CVError** (*fold_errors*)
Bases: `object`

CVError is a class that saves `Error` objects from all folds of a cross-validation method.

Variables

- **fold_errors** (list of `Error`) – a list of all Error objects created through cross-validation process.
- **all_fold_errors** (*dict*) – a dictionary containing lists of error values of all folds.
- **all_fold_mean_errors** (*dict*) – a dictionary containing the mean of *all_fold_errors* lists.

`calc_metrics``()`

Calculate a folds mean of all error metrics.

Available error metrics are “rmse”, “mse”, “mae”, “me”, “mpe”, “mape”, “std”, “hr”, “hr+”, “hr-” and “accuracy”.

`print_errors``()`

Print a mean of all error through all folds.

class `elm.mltools.`**`Error`**`(`*expected*, *predicted*, *regressor_name=u‘’*`)`

Bases: `object`

Error is a class that saves expected and predicted values to calculate error metrics.

Variables

- **regressor_name** (*str*) – Deprecated.
- **expected_targets** (*numpy.ndarray*) – array of expected values.
- **predicted_targets** (*numpy.ndarray*) – array of predicted values.
- **dict_errors** (*dict*) – a dictionary containing all calculated errors and their values.

`calc_metrics``()`

Calculate all error metrics.

Available error metrics are “rmse”, “mse”, “mae”, “me”, “mpe”, “mape”, “std”, “hr”, “hr+”, “hr-” and “accuracy”.

`get``(`*error*`)`

Calculate and return value of an error.

Parameters **error** (*str*) – Error to be calculated.

Returns value of desired error.

Return type float

`get_anderson``()`

Anderson-Darling test for data coming from a particular distribution.

Returns statistic value, critical values and significance values.

Return type tuple

Note: Need scipy.stats module to perform Anderson-Darling test.

`get_shapiro``()`

Perform the Shapiro-Wilk test for normality.

Returns statistic value and p-value.

Return type tuple

Note: Need scipy.stats module to perform Shapiro-Wilk test.

`print_errors``()`

Print all errors metrics.

Note: For better printing format, install `prettytable`.

`print_values``()`

Print expected and predicted values.

class `elm.mltools.`**`MLTools`**

Bases: `object`

A Python implementation of several methods needed for machine learning classification/regression.

Variables

- **last_training_pattern** (*numpy.ndarray*) – Full path to the package to test.
- **has_trained** (*boolean*) – package_name str
- **cv_best_rmse** (*float*) – package_name str

`load_regressor``(`*file_name*`)`

Load classifier/regressor to memory.

`save_regressor``(`*file_name*`)`

Save current classifier/regressor to file_name file.

`elm.mltools.`**`kfold_cross_validation`**`(`*ml*, *database*, *params*, *number_folds=10*, *dataprocess=None*`)`

Performs a k-fold cross-validation.

Parameters

- **ml** (`ELMKernel` or `ELMRandom`) –
- **database** (*numpy.ndarray*) – uses 'data' matrix to perform cross-validation.
- **params** (*list*) – list of parameters from *ml* to train/test.
- **number_folds** (*int*) – number of folds to be created from training and testing matrices.
- **dataprocess** (`DataProcess`) – an object that will pre-process database before training. Defaults to None.

Returns tuple of `CVError` from training and testing.

Return type tuple

`elm.mltools.`**`read`**`(`*file_name*`)`

Read data from txt file.

Parameters **file_name** (*str*) – path and file name.

Returns numpy.ndarray: a matrix containing all read data.

`elm.mltools.`**`split_sets`**`(`*data*, *training_percent=None*, *n_test_samples=None*, *perm=False*`)`

Split data matrix into training and test matrices.

Training matrix size will be set using the training_percent parameter, so its samples are the firsts samples found at data matrix, the rest of samples will be testing matrix.

If neither training_percent or number_test_samples are set, an error will happen, only one of the parameters can be set at a time.

Parameters

- **data** (*numpy.ndarray*) – A matrix containing nxf patterns features.
- **training_percent** (*float*) – An optional parameter used to calculate the number of patterns of training matrix.
- **n_test_samples** (*int*) – An optional parameter used to set the number of patterns of testing matrix.

- **perm** (*bool*) – A flag to choose if should permute(shuffle) database before splitting sets.

Returns Both training and test matrices.

Return type tuple

`elm.mltools.`**`time_series_cross_validation`**(*ml*, *database*, *params*, *number_folds=10*, *dataprocess=None*)

Performs a k-fold cross-validation on a Time Series as described by Rob Hyndman.

See also:

http://robjhyndman.com/hyndsight/crossvalidation/

Parameters

- **ml** (`ELMKernel` or `ELMRandom`) –
- **database** (*numpy.ndarray*) – uses 'data' matrix to perform cross-validation.
- **params** (*list*) – list of parameters from *ml* to train/test.
- **number_folds** (*int*) – number of folds to be created from training and testing matrices.
- **dataprocess** (`DataProcess`) – an object that will pre-process database before training. Defaults to None.

Returns tuple of `CVError` from training and testing.

Return type tuple

`elm.mltools.`**`write`**(*file_name*, *data*)

Write data to txt file.

Parameters

- **file_name** (*str*) – path and file name.
- **data** (*numpy.ndarray*) – data to be written.

Project Info

3.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

3.1.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/acba/elm/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Python Extreme Learning Machine (ELM) could always use more documentation, whether as part of the official Python Extreme Learning Machine (ELM) docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/acba/elm/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.1.2 Get Started!

Ready to contribute? Here's how to set up *elm* for local development.

1. Fork the *elm* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/elm.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv elm
   $ cd elm/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 elm tests
   $ python setup.py test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

3.1.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.

3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/acba/elm/pull_requests and make sure that the tests pass for all supported Python versions.

3.1.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_elm
```

3.2 Credits

3.2.1 Development Lead

- Augusto Almeida <acba@cin.ufpe.br>

3.2.2 Contributors

None yet. Why not be the first?

3.3 History

3.4 0.1.0 (2015-02-03)

- First release on PyPI.

3.5 0.1.1 (2015-02-10)

- Fixed some package issues.
- Added Python 2.7 support
- Added extra parameter to search_param method.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

Index