<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Getting started</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1.1 Installation</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1.2 Tutorial</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Main functions (l1l2py)</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2.1 Stage I: Minimal Model Selection</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2.2 Stage II: Nested lists generation</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>2.3 Complete model selection</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>Algorithms (l1l2py.algorithms)</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>3.1 Implementation details</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>3.2 Regularization Algorithms</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>3.3 Utility Functions</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>Tools (l1l2py.tools)</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>4.1 Range generators</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>4.2 Data normalizers</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>4.3 Error functions</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>4.4 Cross Validation utilities</td>
<td>28</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Index</td>
<td></td>
<td>33</td>
</tr>
</tbody>
</table>
L1L2Py is a Python package to perform feature selection by means of $\ell_1\ell_2$ regularization with double optimization.

L1L2Py makes use of NumPy to provide fast N-dimensional array manipulation. It is licensed under GNU General Public License (GPL) version 3.

L1L2Py is based on the minimization of the (naive) $\ell_1\ell_2$ functional introduced in [Zou05] using the algorithm studied from the theoretical viewpoint in [DeMol09a]. The current implementation exploits the minimization algorithm proposed in [Beck09].

L1L2Py is the Python implementation of the one proposed and applied in [DeMol09b]. It consists of two stages. The first one identifies the minimal set of relevant variables (in terms of prediction error). Starting from the minimal list, the second stage selects the family of (almost completely) nested lists of relevant variables for increasing values of linear correlation.
L1L2Py is a simple and lightweight python package to perform variable selection. The algorithms proposed and implemented are been well studied in different experimental settings.

The package is self contained and gives all the needed tools to generate sparse solution for a given linear classification or regression problem.

1.1 Installation

L1L2Py is available open-source under the GNU GPLv3 license. It requires Python version 2.5 or higher and the NumPy package.

There are two ways you can get it:

• **Automatic Installation (recommended)**
  L1L2Py is available on the Python Package Index and can be installed via easy-install
  $ easy_install -U L1L2Py

  or pip
  $ pip install -U L1L2Py

• **Manual Installation**
  Download the latest official version L1L2Py-1.0.5.tar.gz, then:
  $ tar xzvf L1L2Py-1.0.5.tar.gz
  $ cd L1L2Py-1.0.5
  $ python setup.py install

  Using this manual installation, if the testing framework Nose is installed, is possible to run the given Unit Test running the nosetests script
  $ nosetests
  ....................................
  Ran 36 tests in 3.002s

Moreover, in order to generate the plots showed in the following tutorial the Matplotlib package (with the mplot3d toolkit enabled) is required.
1.2 Tutorial

This tutorial aims to show how the $\ell_1\ell_2$ regularization with double optimization is able to perform variable selection. Moreover, the tutorial shows how to use L1L2Py on a synthetic dataset generated with a given function which simulates a linear regression problem with a subset of relevant and linearly correlated variables.

Even if it's not mandatory, in order to better understand this tutorial the reader should read the method described in [DeMol09b], or at least the introduction of Main functions (l1l2py).

1.2.1 Synthetic Data Generation

Using the script `dataset_generation.py` (L1L2Py-1.0.5/docs/tutorial/dataset_generation.py), synthetic data can be generated for a supervised regression problem. The script contains a function (called `correlated_dataset`) which generates a data matrix with some relevant, correlated and noisy variables.

Running the script with only two parameters (i.e. the data matrix dimensions) two text files, namely `data.txt` and `labels.txt` are generated in the same directory containing the script file.

```bash
$ python dataset_generation.py 100 40
Generation of 100 samples with 40 variables... done
$ ls
dataset_generation.py data.txt labels.txt
```

The script generates a random dataset with 3 groups of 5 correlated variables. In total, there are 15 relevant variables and, following the example above, $40 - 15 = 25$ noisy variables. The weight assigned to each relevant variable is 1.0.

The data matrix and the labels matrix generated with the script and used in this tutorial can be found in the directory `L1L2py-1.0.5/docs/tutorial`, where the script itself is located.

To familiarize with the l1l2py code, the two files can be copied where needed and used following the tutorial steps below (alternatively, different datasets can be generated using either the script or the function `correlated_dataset`).

1.2.2 Preparing the data

First, starting a new python terminal, import the needed packages:

```python
>>> import numpy as np
>>> import l1l2py
```

and load the data from disk (change file paths as needed):

```python
>>> X = np.loadtxt('tutorial/data.txt')
>>> Y = np.loadtxt('tutorial/labels.txt')
```

Then, split the data in two blocks, training-set and test-set using the standard NumPy functions `numpy.vsplit` and `numpy.hsplit`

```python
>>> train_data, test_data = np.vsplit(X, 2)
>>> print train_data.shape, test_data.shape
(50, 40) (50, 40)
>>> train_labels, test_labels = np.hsplit(Y, 2)
>>> print train_labels.shape, test_labels.shape
(50,) (50,)
```

as shown, each set contains 50 samples.
1.2.3 Setting parameters ranges

At this point a correct range for the regularization parameters has to be chosen. The function `l1l2py.algorithms.l1_bound` can be used to devise an optimal range for the sparsity parameter $\tau$.

```python
>>> train_data_centered = l1l2py.tools.center(train_data)
>>> tau_max = l1l2py.algorithms.l1_bound(train_data_centered, train_labels)
>>> print tau_max
10.5458157567
```

Note that the matrix is centered, because the same normalization will be used when running the model selection procedure (see later).

Using this parameter to solve a Lasso optimization problem, a void solution would be obtained:

```python
>>> beta = l1l2py.algorithms.l1l2_regularization(train_data_centered, ...
... train_labels, 0.0, tau_max)
>>> print np.allclose(np.zeros_like(beta), beta)
True
```

A good choice for the extreme values for $\tau$ could be

```python
>>> tau_max = tau_max - 1e-2
>>> tau_min = tau_max * 1e-2
>>> beta_max = l1l2py.algorithms.l1l2_regularization(train_data_centered, ...
... train_labels, 0.0, tau_max)
>>> beta_min = l1l2py.algorithms.l1l2_regularization(train_data_centered, ...
... train_labels, 0.0, tau_min)
>>> print len(beta_max.nonzero()[0]), len(beta_min.nonzero()[0])
1 10
```

The minimum value of $\tau$ should be set in order to get a solution with more non-zero variables than the number of hypothetical number of relevant groups of correlated variables (in the our case we know to have 3 groups).

The range of $\tau$ values can therefore be set as:

```python
>>> tau_range = l1l2py.tools.geometric_range(tau_min, tau_max, 20)
```

For the regularization parameter $\lambda$ a wide range of values is advisable

```python
>>> lambda_range = l1l2py.tools.geometric_range(1e-6, 1e-3, 7)
```

as for the correlation parameter $\mu$. For this simple example some different levels of correlation are set, starting from 0.0

```python
>>> mu_range = [0.0, 0.001, 0.01, 0.1, 1.0]
```

1.2.4 Run the model selection

To correctly use the Stage I: Minimal Model Selection, cross validation splits must be generated:

```python
>>> cv_splits = l1l2py.tools.kfold_splits(train_labels, k=5) #5-fold CV
```

Now, call the `l1l2py.model_selection` function to get the results of model selection (the complete execution of the function will take some minutes)

```python
>>> out = l1l2py.model_selection(train_data, train_labels, ...
... test_data, test_labels, ...
... mu_range, tau_range, lambda_range, ...
... cv_splits, ...
... cv_error_function=l1l2py.tools.regression_error, ...
... error_function=l1l2py.tools.regression_error, ...
... data_normalizer=l1l2py.tools.center, ...
... labels_normalizer=None)
```
1.2.5 Analyze the results

The optimal value of $\tau$ and $\lambda$ found in the Stage I: Minimal Model Selection are:

```python
>>> print out['tau_opt'], out['lambda_opt']
0.451073293459 0.000316227766017
```

The module `plot.py` (L1L2py-1.0.5/docs/tutorial/plot.py), provides a function (called `kcv_errors`) to plot the mean cross validation error (remember that for some high values of $\tau$, the solution could be void on some cross validation splits, see Stage I: Minimal Model Selection, so the mean could be evaluated on a subset of $(\tau, \lambda)$ pairs)

```python
>>> from matplotlib import pyplot as plt
>>> from plot import kcv_errors

```  

```python
>> from matplotlib import pyplot as plt
>>> from plot import kcv_errors

```

```
```python
tau_max_idx = out['kcv_err_ts'].shape[0]
```  

```
```python
kcv_errors(out['kcv_err_ts'],
... np.log10(tau_range[:tau_max_idx]), np.log10(lambda_range),
... r'$\log_{10}(\tau)$', r'$\log_{10}(\lambda)$')
```  

```python
plt.show()
```  

Since the error increases rapidly with the highest value of $\tau$, is useful to show the error surface removing the (corresponding) last row from the mean errors matrix

```python
>>> tau_max_idx -= 1
```  

```
```python
kcv_errors(out['kcv_err_ts'][:tau_max_idx,:],
... np.log10(tau_range[:tau_max_idx]), np.log10(lambda_range),
... r'$\log_{10}(\tau)$', r'$\log_{10}(\lambda)$')
```  

```python
plt.show()
```
The (almost completely) nested list of relevant variables is stored in the `selected_list` entry of the resulting `dict` object:

```python
>>> for mu, sel in zip(mu_range, out['selected_list']):
...     print "%.3f:" % mu, sel.nonzero()[0]
0.000: [ 3 4 5 10 12 14]
0.001: [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14]
0.010: [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14]
0.100: [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14]
1.000: [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 22 24 29 32 35 38 39]
```

Remembering that in the used dataset we have 3 groups of 5 relevant variables, with indexes from 0 to 14, the result shows that the minimal list contains two variables belonging to the first group (indexes 3 and 4), one variable belonging to the second group (index 5) and three variables belonging to the third group (indexes 10, 12 and 14), without any noisy variables! Incrementing the correlation parameter all the relevant variables can be included obtaining a model with (almost) constant prediction power.

In fact, the test error evaluated by the RLS solution computed on the selected variables (with the optimal value of $\lambda$) is:

```python
>>> for mu, err in zip(mu_range, out['err_ts_list']):
...     print "%.3f: %.3f" % (mu, err)
0.000: 2.131
0.001: 2.236
0.010: 2.224
0.100: 2.224
1.000: 2.227
```

### 1.2.6 Appendix: functions used in this tutorial

```python
plot.kcv_errors(errors, range_x, range_y, label_x, label_y)
```

Plot a 3D error surface.

**Parameters**

- `errors`: (N, D) ndarray
  - Error matrix.
range_x: array_like of N values
   First axis values.
range_y: array_like of D values
   Second axis values.
label_x: str
   First axis label.
label_y: str
   Second axis label.

Examples
>>> errors = numpy.empty((20, 10))
>>> x = numpy.arange(20)
>>> y = numpy.arange(10)
>>> for i in range(20):
...   for j in range(10):
...     errors[i, j] = (x[i] * y[j])
...  >> kcv_errors(errors, x, y, 'x', 'y')
  >>> plt.show()

dataset_generation.correlated_dataset(num_samples, num_variables, groups, weights, variables_stdev=1.0, correlations_stdev=0.01, labels_stdev=0.01)

Random supervised dataset generation with correlated variables.
The function returns a supervised training set with num_samples examples with num_variables variables.

Parameters
num_samples: int
   Number of samples.
num_variables: int
   Number of variables.
groups: tuple of int
   For each group of relevant variables indicates the group cardinality.
weights: array_like of sum(groups) float
   True regression model.
variables_stdev: float, optional (default is 1.0)
   Standard deviation of the zero-mean Gaussian distribution generating variables column vectors.
correlations_stdev: float, optional (default is 1e-2)
   Standard deviation of the zero-mean Gaussian distribution generating errors between variables which belong to the same group
labels_stdev: float, optional (default is 1e-2)
   Standard deviation of the zero-mean Gaussian distribution generating regression errors.

Returns
X: (num_samples, num_variables) ndarray
   Data matrix.
The data will have \( \text{len(groups)} \) correlated groups of variables, where for each one the function generates a column vector \( \mathbf{x} \) of \( \text{num_samples} \) values drawn from a zero-mean Gaussian distribution with standard deviation equal to \( \text{variables_stdev} \).

For each variable of the group associated with the \( \mathbf{x} \) vector, the function generates the values as

\[
\mathbf{x}^j = \mathbf{x} + \epsilon_x,
\]

where \( \epsilon_x \) is additive noise drawn from a zero-mean Gaussian distribution with standard deviation equal to \( \text{correlations_stdev} \).

The regression values will be generated as

\[
\mathbf{Y} = \tilde{\mathbf{X}} \tilde{\mathbf{\beta}} + \epsilon_y,
\]

where \( \tilde{\mathbf{\beta}} \) is the \texttt{weights} parameter, a list of \( \text{sum(groups)} \) coefficients of the relevant variables, \( \tilde{\mathbf{X}} \) is the submatrix containing only the column related to the relevant variables and \( \epsilon_y \) is additive noise drawn from a zero-mean Gaussian distribution with standard deviation equal to \( \text{labels_stdev} \).

At the end the function returns the matrices \( \mathbf{X} \) and \( \mathbf{Y} \) where

\[
\mathbf{X} = [\tilde{\mathbf{X}}; \mathbf{X}_N]
\]

is the concatenation of the matrix \( \tilde{\mathbf{X}} \) with the relevant variables with \( \text{num_variables} - \text{sum(groups)} \) noisy variables generated independently using values drawn from a zero-mean Gaussian distribution with standard deviation equal to \( \text{variables_stdev} \).

**Examples**

```python
>>> X, Y = correlated_dataset(30, 40, (5, 5, 5), [3.0]*15)
>>> X.shape
(30, 40)
>>> Y.shape
(30, 1)
```
This module implements the two main stages of the $\ell_1\ell_2$ with double optimization variable selection, as in [DeMol09b].

Given a supervised training set $(X, Y)$, the aim is to select a linear model built on few relevant input variables with good prediction ability.

The linear model is $X\beta$, where $\beta$ is found as the minimizer of the (naive) elastic-net functional combined with a regularized least squares functional.

\[
\frac{1}{n} \| Y - X\beta \|^2_2 + \mu \| \beta \|^2_2 + \tau \| \beta \|^1_1
\]

\[
\frac{1}{n} \| Y - \tilde{X}\tilde{\beta} \|^2_2 + \lambda \| \tilde{\beta} \|^2_2
\]

in which $\tilde{\beta}$ and $\tilde{X}$ represent, respectively, the weights vector and the input matrix restricted to the genes selected by the $\ell_1\ell_2$ selection.

The optimal solution depends on two regularization parameters, $\tau$ and $\lambda$ and one correlation parameter $\mu$ and is found in two different stages:

- **Stage I** (minimal_model)
  This stage aims at selecting the optimal pair of regularization parameters $\tau_{opt}$ and $\lambda_{opt}$ within a k-fold cross validation loop for a fixed and small value of the correlation parameter $\mu$.
  The function follows exactly the pseudocode described in [DeMol09b] (pag.7 - Stage I).

- **Stage II** (nested_models)
  For fixed $\tau_{opt}$ and $\lambda_{opt}$, Stage II identifies the set of relevant lists of variables for increasing values of the correlation parameter $\mu$.

  **Note:** For $\tau_{opt}$ and $\lambda_{opt}$ the lists of relevant variables have same prediction power [DeMol09a].

The function performs exactly the pseudocode described in [DeMol09b] (pag.7 - Stage II).

This module also provide a wrapper function (model_selection) that runs the two stages sequentially.

### 2.1 Stage I: Minimal Model Selection

```
1112py.minimal_model(data, labels, mu, tau_range, lambda_range, cv_splits, error_function,
data_normalizer=None, labels_normalizer=None)
```

Minimal model selection.
Given a supervised training set (data and labels), for a fixed value of $\mu$ (should be minimum), it finds the values in $\tau$ range and $\lambda$ range minimizing the prediction error via cross validation (see error functions in the 11l2py.tools module).

Cross validation splits must be provided (cv_splits) as a list of pairs containing training-set and validation-set indexes (see cross validation tools in the 11l2py.tools module).

Data and labels will be normalized on each split using the function data_normalizer and labels_normalizer (see data normalization functions in the 11l2py.tools module).

| Warning: On each cross validation split the number of valid solutions (not void) may be different (on high values of $\tau$). The function calculates the optimum value of $\tau$ for which the model is not void on all cross validation splits. This means than in extreme cases the output could be void. |

**Parameters**

- **data**: (N, P) ndarray  
  Data matrix.
- **labels**: (N,) or (N, 1) ndarray  
  Labels vector.
- **mu**: float  
  Minimum $l_2$ norm penalty ($l1l2$ functional).
- **tau_range**: array_like of $T$ floats  
  $l_1$ norm penalties ($l1l2$ functional).
- **lambda_range**: array_like of $L$ floats  
  $l_2$ norm penalties ($RLS$ functional).
- **cv_splits**: array_like of tuples  
  Each tuple contains two lists with the training set and testing set indexes.
- **error_function**: function object  
  Cross validation error function.
- **data_normalizer**: function object, optional (default is None)  
  Data normalization function.
- **labels_normalizer**: function object, optional (default is None)  
  Labels normalization function.

**Returns**

- **err_ts**: (< $T$, $L$) ndarray  
  Matrix of average cross validation error on the training set. The first dimension depends on the number of valid $\tau$ values, even zero.
- **err_tr**: (< $T$, $L$) ndarray  
  Matrix of average cross validation error on the training set. The first dimension depends on the number of valid $\tau$ values, even zero.

**Raises**

- **ValueError**:  
  If the given range of $\tau$ values produces all void solutions with the given data splits.
2.2 Stage II: Nested lists generation

`l1l2py.nested_models(data, labels, test_data, test_labels, mu_range, tau, lambda_, error_function, data_normalizer=None, labels_normalizer=None, return_predictions=False)`

The function generates the models with the (almost) nested lists of selected variables.

Given a training set (data and labels) and a test set (test_data and test_labels), for fixed values of tau and lambda (should be the optimal values estimated at Stage I), it calculates one model for each increasing value in mu_range.

Data and labels will be normalized using the function `data_normalizer` and `labels_normalizer` (see data normalization functions in the l1l2py.tools module).

The function returns test and training errors using the `error_function` provided (see error functions in the l1l2py.tools module).

**Parameters**

- **data**: (N, P) ndarray
  - Data matrix.
- **labels**: (N,) or (N, 1) ndarray
  - Labels vector.
- **test_data**: (T, P) ndarray
  - Test set matrix.
- **test_labels**: (T,) or (T, 1) ndarray
  - Test set labels vector.
- **mu_range**: array_like of M floats
  - L2 norm penalties (l1l2 functional).
- **tau**: float
  - Optimal L1 norm penalty (l1l2 functional).
- **lambda_**: float
- **error_function**: function object
  - Error function.
- **data_normalizer**: function object, optional (default is None)
  - Data normalization function.
- **labels_normalizer**: function object, optional (default is None)
  - Labels normalization function.

**Returns**

- **beta_list**: list of M (S,1) ndarray
  - Models calculated for each value in mu_range.
- **selected_list**: list of M (P,) ndarray of boolean
  - Selected feature for each models calculated.
- **err_ts_list**: list of M floats
  - Test error for the models calculated.
- **err_tr_list**: list of M floats
  - Training error for the models calculated.
Training error for the models calculated.

**prediction_ts_list**: list of M (T, 1) ndarray
Prediction vector calculated for each value in `mu_range` on the test set.

**prediction_tr_list**: list of M (N, 1) ndarray
Prediction vector calculated for each value in `mu_range` on the training set.

**Raises**

**ValueError**: If the given value of `tau` produces a void solution with the given data.

## 2.3 Complete model selection

**l1l2py.model_selection**(*data*, *labels*, *test_data*, *test_labels*, *mu_range*, *tau_range*, *lambda_range*, *cv_splits*, *cv_error_function*, *error_function*, *data_normalizer=None*, *labels_normalizer=None*, *sparse=False*, *regularized=True*, *return_predictions=False*)

Complete model selection procedure.

It executes the two stages implemented in `minimal_model` and `nested_models` and returns their output wrapped in a dictionary.

Note that the error function calculated in the **Stage I** may have more than one minimum.

By default the less sparse but more regularized solution (minimum value of `tau` and maximum value of `lambda`) is selected, in the set of `(tau, lambda)` pairs with minimum error.

The boolean parameters **sparse** and **regularized** allow to change this behaviour.

**Note**: See the functions documentation for details on each stage and the meaning of each parameter. The **Parameters** section describes only the **sparse** and **regularized** parameters.

**Parameters**

- **sparse**: bool, optional (default is `False`)
  - If `True`, the function selects at **Stage I** the sparsest solution with minimum cross validation error.

- **regularized**: bool, optional (default is `True`)
  - If `True`, the function selects at **Stage I** the most regularized solution with minimum cross validation error.

**Returns**

- **out**: dict
  - Output dictionary. According with the parameters the dictionary has the following keys:
    - **kcv_err_ts**: [(T, L) ndarray] [**Stage I**] Mean cross validation errors on the training set.
    - **kcv_err_tr**: [(T, L) ndarray] [**Stage I**] Mean cross validation errors on the training set.
    - **tau_opt**: [float] Optimal value of tau selected in `tau_range`.
    - **lambda_opt**: [float] Optimal value of lambda selected in `lambda_range`.
**beta_list**
list of M (S,1) ndarray [STAGE II] Models calculated for each value in mu_range.

**selected_list**
list of M (P,) ndarray of boolean [STAGE II] Selected variables for each model calculated.

**err_ts_list**
list of M floats [STAGE II] List of Test errors evaluated for all the models.

**err_tr_list**
list of M floats [STAGE II] List of Training errors evaluated for all the models.

**prediction_ts_list**
list of M two-dimensional ndarray, optional [STAGE II] Prediction vectors for the models evaluated on the test set.

**prediction_tr_list**
list of M two-dimensional ndarray, optional [STAGE II] Prediction vectors for the models evaluated on the training set.
ALGORITHMS (L1L2PY.ALBORITHMS)

In order to describe the function implemented in this module, we have to assume some notation.

Assuming to have a centered data matrix \( X \in \mathbb{R}^{n \times p} \) and a column vector of regression values \( Y \in \mathbb{R}^n \) or binary labels \( Y \in \{-1, 1\}^n \), we want to minimize the regression/classification error solving a Regularized Least Square (RLS) problem.

In this module two main algorithms are implemented. The first one solves a classical RLS problem with a penalty on the \( \ell_2 \)-norm of the vector \( \beta \) (also called ridge_regression)

\[
\beta^* = \arg \min_{\beta} \left\{ \frac{1}{n} \| Y - X \beta \|_2^2 + \mu \| \beta \|_2^2 \right\},
\]

with \( \mu > 0 \).

The second one minimizes a functional with a linear combination of two penalties on the \( \ell_1 \)-norm and \( \ell_2 \)-norm of the vector \( \beta \) (also called l1l2_regularization)

\[
\beta^* = \arg \min_{\beta} \left\{ \frac{1}{n} \| Y - X \beta \|_2^2 + \mu \| \beta \|_2^2 + \tau \| \beta \|_1 \right\},
\]

with \( \mu > 0 \) and \( \tau > 0 \).

### 3.1 Implementation details

While (3.1) has closed-form solution, for (3.2) there are many different approaches. In this module we provide an Iterative Shrinkage-Thresholding Algorithm (ISTA) proposed in [DeMol09a] exploiting a faster variation (called FISTA) proposed in [Beck09].

Starting from a null vector \( \beta \), each step updates the value of \( \beta \) until convergence:

\[
\beta^{(k+1)} = S_{\sigma} \left( (1 - \frac{\mu}{\sigma}) \beta^k + \frac{1}{n \sigma} X^T [Y - X \beta^k] \right)
\]

where, \( S_{\gamma>0} \) is the soft-thresholding function

\[
S_{\gamma}(x) = \text{sign}(x) \max(0, |x| - \gamma/2)
\]

The constant \( \sigma \) is a (theoretically optimal) step size which depends on the data:

\[
\sigma = \frac{e}{n} + \mu,
\]
where \( e \) is the maximum eigenvalue of the matrix \( X^T X \).

The convergence is reached when for each \( j \in \{0, \ldots, d - 1\} \):

\[
|\beta^k_j - \beta^{k-1}_j| \leq |\beta^k_j| \ast (tol/k),
\]

where \( tol > 0 \) and before \( k \) reaches a fixed maximum number of iterations.

### 3.2 Regularization Algorithms

**l1l2py.algorithms.ridge_regression** (*data, labels, mu=0.0*)

Implementation of the Regularized Least Squares solver.

It solves the ridge regression problem with parameter \( \mu \) on the \( l2 \)-norm.

**Parameters**

- **data**: (N, P) ndarray
  - Data matrix.
- **labels**: (N,) or (N, 1) ndarray
  - Labels vector.
- **mu**: float, optional (default is 0.0)
  - \( l2 \)-norm penalty.

**Returns**

- **beta**: (P, 1) ndarray
  - Ridge regression solution.

**Examples**

```python
>>> X = numpy.array([[0.1, 1.1, 0.3], [0.2, 1.2, 1.6], [0.3, 1.3, -0.6]])
>>> beta = numpy.array([0.1, 0.1, 0.0])
>>> Y = numpy.dot(X, beta)
>>> beta = l1l2py.algorithms.ridge_regression(X, Y, 1e3).T
>>> len(numpy.flatnonzero(beta))
3
```

**l1l2py.algorithms.l1l2_regularization** (*data, labels, mu, tau*, beta=None, kmax=100000, tolerance=1e-05, return_iterations=False, adaptive=False)

Implementation of the Fast Iterative Shrinkage–Thresholding Algorithm to solve a least squares problem with \( l1l2 \) penalty.

It solves the \( l1l2 \) regularization problem with parameter \( \mu \) on the \( l2 \)-norm and parameter \( \tau \) on the \( l1 \)-norm.

**Parameters**

- **data**: (N, P) ndarray
  - Data matrix.
- **labels**: (N,) or (N, 1) ndarray
  - Labels vector.
- **mu**: float
  - \( l2 \)-norm penalty.
- **tau**: float
  - \( l1 \)-norm penalty.
- **beta**: float
  - Optional
- **kmax**: int
  - Maximum number of iterations (default: 100,000).
- **tolerance**: float
  - Convergence tolerance (default: 1e-05).
- **return_iterations**: bool (default: False)
  - Return iteration count.
- **adaptive**: bool (default: False)
  - Use adaptive step size.

**Examples**

```python
>>> X = numpy.array([[0.1, 1.1, 0.3], [0.2, 1.2, 1.6], [0.3, 1.3, -0.6]])
>>> Y = numpy.dot(X, beta)
>>> beta = l1l2py.algorithms.l1l2_regularization(X, Y, 1e3, 1e-3).T
>>> len(numpy.flatnonzero(beta))
3
```
beta : (P,) or (P, 1) ndarray, optional (default is None)
    Starting value for the iterations. If None, then iterations starts from the empty model.

kmax : int, optional (default is 1e5)
    Maximum number of iterations.

tolerance : float, optional (default is 1e-5)
    Convergence tolerance.

return_iterations : bool, optional (default is False)
    If True, returns the number of iterations performed. The algorithm has a predefined minimum number of iterations equal to 10.

adaptive : bool, optional (default is False)
    If True, minimization is performed calculating an adaptive step size for each iteration.

Returns
    beta : (P, 1) ndarray
        l1l2 solution.

    k : int, optional
        Number of iterations performed.

Examples
>>> X = numpy.array([[0.1, 1.1, 0.3], [0.2, 1.2, 1.6], [0.3, 1.3, -0.6]])
>>> beta = numpy.array([0.1, 0.1, 0.0])
>>> Y = numpy.dot(X, beta)
>>> beta = l1l2py.algorithms.l1l2_regularization(X, Y, 0.1, 0.1)
>>> len(numpy.flatnonzero(beta))
1

3.3 Utility Functions

l1l2py.algorithms.l1_bound(data, labels)
    Estimation of an useful maximum bound for the l1 penalty term.

    Fixing mu close to 0.0 and using the maximum value calculated with this function as tau in the l1l2 regularization, the solution vector contains only zero elements.

    For each value of tau smaller than the maximum bound the solution vector contains at least one non zero element.

Parameters
    data : (N, P) ndarray
        Data matrix.

    labels : (N,) or (N, 1) ndarray
        Labels vector.

Returns
    tau_max : float
        Maximum tau.
Examples

```python
>>> X = numpy.array([[0.1, 1.1, 0.3], [0.2, 1.2, 1.6], [0.3, 1.3, -0.6]])
>>> beta = numpy.array([0.1, 0.1, 0.0])
>>> Y = numpy.dot(X, beta)
>>> tau_max = l1l2py.algorithms.l1_bound(X, Y)
>>> l1l2py.algorithms.l1l2_regularization(X, Y, 0.0, tau_max).T
array([[ 0., 0., 0.]]
>>> beta = l1l2py.algorithms.l1l2_regularization(X, Y, 0.0, tau_max - 1e-5)
>>> len(numpy.flatnonzero(beta))
1
```

`l1l2py.algorithms.l1l2_path` *(data, labels, mu, tau_range, beta=None, kmax=100000, tolerance=1e-05, adaptive=False)*

Efficient solution of different l1l2 regularization problems on increasing values of the l1-norm parameter.

Finds the l1l2 regularization path for each value in tau_range and fixed value of mu. The values in tau_range are used during the computation in reverse order, while the output path has the same ordering of the tau values.

**Note:** For efficiency purposes, if mu = 0.0 and the number of non-zero values is higher than N for a given value of tau (that means algorithm has reached the limit of allowed iterations), the following solutions (for smaller values of tau) are simply the least squares solutions.

**Warning:** The number of solutions can differ from len(tau_range). The function returns only the solutions with at least one non-zero element. For values higher than tau_max a solution have all zero values.

**Parameters**

- **data** : (N, P) ndarray
  Data matrix.

- **labels** : (N,) or (N, 1) ndarray
  Labels vector.

- **mu** : float
  l2-norm penalty.

- **tau_range** : array_like of float
  l1-norm penalties in increasing order.

- **beta** : (P,) or (P, 1) ndarray, optional (default is None)
  Starting value of the iterations. If None, then iterations starts from the empty model.

- **kmax** : int, optional (default is 1e5)
  Maximum number of iterations.

- **tolerance** : float, optional (default is 1e-5)
  Convergence tolerance.

- **adaptive** : bool, optional (default is False)
  If True, minimization is performed calculating an adaptive step size for each iteration.

**Returns**

- **beta_path** : list of (P,) or (P, 1) ndarrays
  l1l2 solutions with at least one non-zero element.
Note
The acceleration method based on \textit{warm starts}, implemented in this function, is been theoretically proved in [Hale08].
This module contains useful functions to be used in combination with the main functions of the package.

The functions included in this module are divided in four groups:

- **Range generators**
- **Data normalizers**
- **Error functions**
- **Cross Validation utilities**

### 4.1 Range generators

l1l2py.tools.linear_range(min_value, max_value, number)

Linear range of values between min_value and max_value.

Sequence of number evenly spaced values from min_value to max_value.

**Parameters**
- min_value : float
- max_value : float
- number : int

**Returns**
- range : (number,) ndarray

**Examples**

```python
>>> l1l2py.tools.linear_range(min_value=0.0, max_value=10.0, number=4)
array([ 0.  ,  3.33333333,  6.66666667, 10.  ])
>>> l1l2py.tools.linear_range(min_value=0.0, max_value=10.0, number=2)
array([ 0.  , 10.  ])
>>> l1l2py.tools.linear_range(min_value=0.0, max_value=10.0, number=1)
array([ 0.  ])
>>> l1l2py.tools.linear_range(min_value=0.0, max_value=10.0, number=0)
array([], dtype=float64)
```

l1l2py.tools.geometric_range(min_value, max_value, number)

Geometric range of values between min_value and max_value.

Sequence of number values from min_value to max_value generated by a geometric sequence.

**Parameters**
- min_value : float
- max_value : float
- number : int
Returns

range : (number,) ndarray

Raises

ZeroDivisionError :

If min_value is 0.0 or number is 1

Examples

>>> l1l2py.tools.geometric_range(min_value=0.0, max_value=10.0, number=4)
Traceback (most recent call last):
...
ZeroDivisionError: float division

>>> l1l2py.tools.geometric_range(min_value=0.1, max_value=10.0, number=4)
array([ 0.1 , 0.46415888, 2.15443469, 10. ])

>>> l1l2py.tools.geometric_range(min_value=0.1, max_value=10.0, number=2)
array([ 0.1, 10. ])

>>> l1l2py.tools.geometric_range(min_value=0.1, max_value=10.0, number=1)
Traceback (most recent call last):
...
ZeroDivisionError: float division

>>> l1l2py.tools.geometric_range(min_value=0.1, max_value=10.0, number=0)
array([], dtype=float64)

Note

The geometric sequence of \( n \) elements between \( a \) and \( b \) is

\[
a, ar^1, ar^2, \ldots, ar^{n-1}
\]

where the ratio \( r \) is

\[
r = \left( \frac{b}{a} \right)^{\frac{1}{n-1}}
\]

4.2 Data normalizers

l1l2py.tools.center (matrix, optional_matrix=None, return_mean=False)

Center columns of a matrix setting each column to zero mean.

The function returns the centered matrix given as input. Optionally centers an optional_matrix with respect to the mean value evaluated for matrix.

Note: A one dimensional matrix is considered as a column vector.

Parameters

matrix : (N,) or (N, P) ndarray

Input matrix whose columns are to be centered.

optional_matrix : (N,) or (N, P) ndarray, optional (default is None)

Optional matrix whose columns are to be centered using mean of matrix. It must have the same number of columns as matrix.

return_mean : bool, optional (default is False)

If True returns mean of matrix.

Returns

matrix_centered : (N,) or (N, P) ndarray
Centered matrix.

**optional_matrix_centered**: (N,) or (N, P) ndarray, optional

Center optional_matrix with respect to matrix

**mean**: float or (P) ndarray, optional

Mean of matrix columns.

**Examples**

```python
def center(X)
    return (X - X.mean(axis=0)) / X.std(axis=0)
```

```python
X = np.array([[1, 2, 3], [4, 5, 6]])
X = center(X)
print(X)
```

```python
X = np.array([[1, 2, 3], [4, 5, 6]])
mean = center(X, return_mean=True)
print(mean)
```

```python
x = np.array([1, 2, 3])
# 1-dimensional matrix
x = center(x, return_mean=True)
print(x)
```

```python
X = np.array([[1, 2, 3], [4, 5, 6]])
X = center(X, X[:, :2])
```

```python
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

**l1l2py.tools.standardize**(matrix, optional_matrix=None, return_factors=False)

Standardize columns of a matrix setting each column with zero mean and unitary standard deviation.

The function returns the standardized matrix given as input. Optionally it standardizes an optional_matrix with respect to the mean and standard deviation evaluated for matrix.

**Note**: A one dimensional matrix is considered as a column vector.

**Parameters**

- **matrix**: (N,) or (N, P) ndarray
  - Input matrix whose columns are to be standardized to mean 0 and standard deviation 1.

- **optional_matrix**: (N,) or (N, P) ndarray, optional (default is None)
  - Optional matrix whose columns are to be standardized using mean and standard deviation of matrix. It must have same number of columns as matrix.

- **return_factors**: bool, optional (default is False)
  - If True, returns mean and standard deviation of matrix.

**Returns**

- **matrix_standardized**: (N,) or (N, P) ndarray
  - Standardized matrix.

- **optional_matrix_standardized**: (N,) or (N, P) ndarray, optional
  - Standardized optional_matrix with respect to matrix

- **mean**: float or (P) ndarray, optional
  - Mean of matrix columns.

- **std**: float or (P) ndarray, optional
  - Standard deviation of matrix columns.
Standard deviation of matrix columns.

**Raises**

**ValueError**:

If matrix has only one row.

**Examples**

```python
generate code examples here...
```

**4.3 Error functions**

**`l1l2py.tools.regression_error(labels, predictions)`**

Returns regression error.

The regression error is the sum of the quadratic differences between the labels values and the predictions values, over the number of samples.

**Parameters**

- `labels`: array_like, shape (N,)
  - Regression labels.

- `predictions`: array_like, shape (N,)
  - Regression labels predicted.

**Returns**

- `error`: float
  - Regression error calculated.

**Note**

The regression error is calculated using the formula

\[
error = \frac{\sum_{i=1}^{N} |l_i - p_i|^2}{N}
\]

l_i \in \text{labels}, p_i \in \text{predicted}

**`l1l2py.tools.classification_error(labels, predictions)`**

Evaluate the binary classification error.

The classification error is based on the sign of the predictions values, with respect to the sign of the data labels.
The function assumes that \texttt{labels} contains positive values for one class and negative values for the other one.

\textbf{Warning:} For efficiency reasons, the values in \texttt{labels} are not checked by the function.

The function \texttt{classification_error} computes the classification error as follows:

\[ \text{error} = \frac{1}{N} \sum_{i=1}^{N} f(l_i, p_i) \]

where

\[ f(l_i, p_i) = \begin{cases} 1 & \text{if } \text{sign}(l_i) \neq \text{sign}(p_i) \\ 0 & \text{otherwise} \end{cases} \]

\textbf{Parameters}

- \texttt{labels} : array_like, shape (N,)
  
  Classification labels (usually contains only 1s and -1s).

- \texttt{predictions} : array_like, shape (N,)
  
  Classification labels predicted.

\textbf{Returns}

- \texttt{error} : float
  
  Classification error evaluated.

\textbf{Examples}

```python
>>> l1l2py.tools.classification_error(labels=[1, 1, 1], predictions=[1, 1, 1])
0.0
>>> l1l2py.tools.classification_error(labels=[1, 1, 1], predictions=[1, 1, -1])
0.33333333333333331
>>> l1l2py.tools.classification_error(labels=[1, 1, 1], predictions=[1, -1, -1])
0.66666666666666663
>>> l1l2py.tools.classification_error(labels=[1, 1, 1], predictions=[-1, -1, -1])
1.0
>>> l1l2py.tools.classification_error(labels=[1, 1, 1], predictions=[10, -2, -3])
0.66666666666666663
```

\textbf{Note}

The classification error is calculated using this formula

\[ \text{error} = \frac{1}{N} \sum_{i=1}^{N} f(l_i, p_i) \]

where

\[ f(l_i, p_i) = \begin{cases} 1 & \text{if } \text{sign}(l_i) \neq \text{sign}(p_i) \\ 0 & \text{otherwise} \end{cases} \]

\textbf{Warning:} The classification error is calculated using the \texttt{numpy.sign} function. Keep in mind that the \texttt{sign(x)} returns 0 if \( x = 0 \).

\texttt{l1l2py.tools.balanced_classification_error} (labels, predictions, error_weights=None)

Returns the binary classification error balanced across the size of classes.

This function returns a balanced classification error. With the default value for \texttt{error_weights}, the function assigns greater weight to the errors belonging to the smaller class.

\textbf{Parameters}

- \texttt{labels} : array_like, shape (N,)
  
  Classification labels (usually contains only 1s and -1s).

- \texttt{predictions} : array_like, shape (N,)
  
  Classification labels predicted.

- \texttt{error_weights} : array_like, shape (N,), optional (default is None)
  
  Classification error weights. If \texttt{None} the default weights are calculated removing from each value in \texttt{labels} their mean value.
Returns

error : float

Classification error calculated.

Examples

```python
>>> l1l2py.tools.balanced_classification_error(labels=[1, 1, 1], predictions=[-1, -1, -1])
0.0
>>> l1l2py.tools.balanced_classification_error(labels=[-1, 1, 1], predictions=[-1, 1, 1])
0.0
>>> l1l2py.tools.balanced_classification_error(labels=[-1, 1, 1], predictions=[1, -1, -1])
0.88888888888888895
>>> l1l2py.tools.balanced_classification_error(labels=[-1, 1, 1], predictions=[1, 1, 1])
0.44444444444444442
>>> l1l2py.tools.balanced_classification_error(labels=[-1, 1, 1], predictions=[1, 1, -1],
...    error_weights=[1, 1, 1])
0.33333333333333331
```

Note

The balanced classification error is calculated using this formula:

\[
error = \frac{\sum_{i=1}^{N} w_i \cdot f(l_i, p_i)}{N}
\]

where \(f(l_i, p_i)\) is as defined above.

With the default weights the error function becomes:

\[
error = \frac{\sum_{i=1}^{N} |l_i - \text{labels}| \cdot f(l_i, p_i)}{N}
\]

**Warning**: If `labels` contains only values belonging to one class, the function returns always 0.0 because \(l_i - \text{labels} = 0\), than \(w_i = 0\) for each \(i\).

### 4.4 Cross Validation utilities

l1l2py.tools.kfold_splits(labels, k, rseed=0)

k-fold cross validation splits.

Given a list of labels, the function produces a list of \(k\) splits. Each split is a pair of tuples containing the indexes of the training set and the indexes of the test set.

**Parameters**

- **labels**: array_like, shape (N,)
  Data labels.
- **k**: int, greater than 0
  Number of splits.
- **rseed**: int, optional (default is 0)
  Random seed.

**Returns**

- **splits**: list of \(k\) tuples
  Each tuple contains two lists with the training set and test set indexes.
Raises

ValueError:

If \( k \) is less than 2 or greater than \( N \).

Examples

```python
>>> labels = range(10)
>>> l1l2py.tools.kfold_splits(labels, 2)
[([7, 1, 3, 6, 8], [9, 4, 0, 5, 2]), ([9, 4, 0, 5, 2], [7, 1, 3, 6, 8])]
>>> l1l2py.tools.kfold_splits(labels, 1)
Traceback (most recent call last):
...
ValueError: 'k' must be greater than one and smaller or equal than the number of samples
```

l1l2py.tools.stratified_kfold_splits (labels, k, rseed=0)

Stratified k-fold cross validation splits.

This function is a variation of kfold_splits, which returns stratified splits. The divisions are made by preserving the percentage of samples for each class, assuming that the problem is binary.

Parameters

labels : array_like, shape (N,)
    Data labels (usually contains only 1s and -1s).

k : int, greater than 0
    Number of splits.

rseed : int, optional (default is 0)
    Random seed.

Returns

splits : list of \( k \) tuples
    Each tuple contains two lists with the training set and test set indexes.

Raises

ValueError:

If `labels` contains more than two classes labels.

ValueError:

If \( k \) is less than 2 or greater than number of positive or negative samples in `labels`.

Examples

```python
>>> labels = range(10)
>>> l1l2py.tools.stratified_kfold_splits(labels, 2)
Traceback (most recent call last):
...
ValueError: 'labels' must contains only two class labels
>>> labels = [1, 1, 1, 1, 1, 1, -1, -1, -1, -1]
>>> l1l2py.tools.stratified_kfold_splits(labels, 2)
[([8, 9, 5, 2, 1], [7, 6, 3, 0, 4]), ([7, 6, 3, 0, 4], [8, 9, 5, 2, 1])]
>>> l1l2py.tools.stratified_kfold_splits(labels, 1)
Traceback (most recent call last):
...
ValueError: 'k' must be greater than one and smaller or equal than number of positive and negative samples in 'labels'.
```

Note

Running this functions more times with the same value of the parameter rseed gives always the same result, in order to allow repeatable experiments. Note, moreover, that each of this functions sets the random seed equal to None, to restore a random seed for the following use of the random module (see random.seed).
References


INDEX

B
balanced_classification_error() (in module l1l2py.tools), 27

C
center() (in module l1l2py.tools), 24
classification_error() (in module l1l2py.tools), 26
correlated_dataset() (in module dataset_generation), 8

G
geometric_range() (in module l1l2py.tools), 23

K
cvk_errors() (in module plot), 7
kfold_splits() (in module l1l2py.tools), 28

L
l1_bound() (in module l1l2py.algorithms), 19
l1l2_path() (in module l1l2py.algorithms), 20
l1l2_regularization() (in module l1l2py.algorithms), 18
linear_range() (in module l1l2py.tools), 23

M
minimal_model() (in module l1l2py), 11
model_selection() (in module l1l2py), 14

N
nested_models() (in module l1l2py), 13

R
regression_error() (in module l1l2py.tools), 26
ridge_regression() (in module l1l2py.algorithms), 18

S
standardize() (in module l1l2py.tools), 25
stratified_kfold_splits() (in module l1l2py.tools), 29