classo

Release 1.0.11

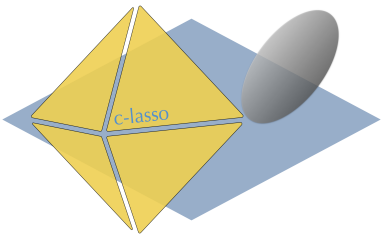
Leo Simpson

May 05, 2021

CONTENTS:

c-lasso is a Python package that enables sparse and robust linear regression and classification with linear equality constraints on the model parameters.

The package is available on *https://github.com/Leo-Simpson/c-lasso*.



CHAPTER

ONE

MATHEMATICAL DESCRIPTION

The forward model is assumed to be:

$$y = X\beta + \sigma\epsilon \qquad \text{subject to} \qquad C\beta = 0$$

Here, y and X are given outcome and predictor data. The vector y can be continuous (for regression) or binary (for classification). C is a general constraint matrix. The vector β comprises the unknown coefficients and σ an unknown scale.

The package handles several different estimators for inferring β and σ), including the constrained Lasso, the constrained scaled Lasso, and sparse Huber M-estimation with linear equality constraints. Several different algorithmic strategies, including path and proximal splitting algorithms, are implemented to solve the underlying convex optimization problems.

We also include two model selection strategies for determining the sparsity of the model parameters: k-fold cross-validation and stability selection.

This package is intended to fill the gap between popular python tools such as scikit-learn which CANNOT solve sparse constrained problems and general-purpose optimization solvers that do not scale well for the considered problems.

Below we show several use cases of the package, including an application of sparse *log-contrast* regression tasks for *compositional* microbiome data.

The code builds on results from several papers which can be found in the [References](#references). We also refer to the accompanying JOSS paper submission, also available on arXiv.

1.1 Regression and classification problems

The c-lasso package can solve six different types of estimation problems: four regression-type and two classification-type formulations.

1.1.1 [R1] Standard constrained Lasso regression

$$\min_{\beta \in \mathbb{R}^d} \|X\beta - y\|^2 + \lambda \|\beta\|_1 \qquad \text{subject to} \qquad C\beta = 0$$

This is the standard Lasso problem with linear equality constraints on the β vector. The objective function combines Least-Squares for model fitting with l1 penalty for sparsity.

1.1.2 [R2] Constrained sparse Huber regression

$$\min_{\beta \in \mathbb{R}^d} h_\rho(X\beta - y) + \lambda \left\|\beta\right\|_1 \qquad \text{subject to} \qquad C\beta = 0$$

This regression problem uses the Huber loss as objective function for robust model fitting with l1 and linear equality constraints on the β vector. The parameter $\rho = 1.345$.

1.1.3 [R3] Constrained scaled Lasso regression

$$\min_{\beta \in \mathbb{R}^d, \sigma \in \mathbb{R}_0} \frac{\left\|X\beta - y\right\|^2}{\sigma} + \frac{n}{2}\sigma + \lambda \left\|\beta\right\|_1 \qquad \text{subject to} \qquad C\beta = 0$$

This formulation is similar to [R1] but allows for joint estimation of the (constrained) β vector and the standard deviation σ in a concomitant fashion[4],[5] . This is the default problem formulation in c-lasso.

1.1.4 [R4] Constrained sparse Huber regression with concomitant scale estimation

$$\min_{\beta \in \mathbb{R}^d, \sigma \in \mathbb{R}_0} \left(h_\rho\left(\frac{X\beta - y}{\sigma}\right) + n \right) \sigma + \lambda \left\|\beta\right\|_1 \qquad \text{subject to} \qquad C\beta = 0$$

This formulation combines [R2] and [R3] to allow robust joint estimation of the (constrained) β vector and the scale σ in a concomitant fashion[?] ,[5] .

1.1.5 [C1] Constrained sparse classification with Square Hinge loss

$$\min_{\beta \in \mathbb{R}^d} \sum_{i=1}^{n} l(y_i x_i^\top \beta) + \lambda \left\|\beta\right\|_1 \qquad \text{subject to} \qquad C\beta = 0$$

where the x_i are the rows of X and l is defined as:

$$l(r) = \begin{cases} (1-r)^2 & if \quad r \leq 1 \\ 0 & if \quad r \geq 1 \end{cases}$$

This formulation is similar to [R1] but adapted for classification tasks using the Square Hinge loss with (constrained) sparse β vector estimation.

[4] P. L. Combettes and C. L. Müller, Perspective M-estimation via proximal decomposition, Electronic Journal of Statistics, 2020, Journal version

[5] P. L. Combettes and C. L. Müller, Regression models for compositional data: General log-contrast formulations, proximal optimization, and microbiome data applications, Statistics in Bioscience, 2020.

1.1.6 [C2] Constrained sparse classification with Huberized Square Hinge loss

$$\min_{\beta\in\mathbb{R}^d} \sum_{i=1}^{n} l_\rho(y_i x_i^\top \beta) + \lambda \|\beta\|_1 \qquad \text{subject to} \qquad C\beta = 0\,.$$

where the x_i are the rows of X and l_ρ is defined as:

$$l_\rho(r) = \begin{cases} (1-r)^2 & if \quad \rho \le r \le 1 \\ (1-\rho)(1+\rho-2r) & if \quad r \le \rho \\ 0 & if \quad r \ge 1 \end{cases}$$

This formulation is similar to [C1] but uses the Huberized Square Hinge loss for robust classification with (constrained) sparse β vector estimation[7].

1.2 Optimization schemes

The available problem formulations [R1-C2] require different algorithmic strategies for efficiently solving the underlying optimization problem. We have implemented four algorithms (with provable convergence guarantees) that vary in generality and are not necessarily applicable to all problems. For each problem type, c-lasso has a default algorithm setting that proved to be the fastest in our numerical experiments.

1.2.1 Path algorithms (Path-Alg)

This is the default algorithm for non-concomitant problems [R1,R3,C1,C2]. The algorithm uses the fact that the solution path along λ is piecewise- affine[1]. When Least-Squares is used as objective function, we derive a novel efficient procedure that allows us to also derive the solution for the concomitant problem [R2] along the path with little extra computational overhead.

1.2.2 Projected primal-dual splitting method (P-PDS)

This algorithm is derived from[2] and belongs to the class of proximal splitting algorithms. It extends the classical Forward-Backward (FB) (aka proximal gradient descent) algorithm to handle an additional linear equality constraint via projection. In the absence of a linear constraint, the method reduces to FB. This method can solve problem [R1]. For the Huber problem [R3], P-PDS can solve the mean-shift formulation of the problem[6].

7 S. Rosset and J. Zhu, Piecewise linear regularized solution paths, Ann. Stat., vol. 35, no. 3, pp. 1012–1030, 2007.

1 B. R. Gaines, J. Kim, and H. Zhou, Algorithms for Fitting the Constrained Lasso, J. Comput. Graph. Stat., vol. 27, no. 4, pp. 861–871, 2018.

2 L. Briceno-Arias and S.L. Rivera, A Projected Primal–Dual Method for Solving Constrained Monotone Inclusions, J. Optim. Theory Appl., vol. 180, Issue 3, March 2019.

6 A. Mishra and C. L. Müller, Robust regression with compositional covariates, arXiv, 2019.

1.2.3 Projection-free primal-dual splitting method (PF-PDS)

This algorithm is a special case of an algorithm proposed in[3] (Eq.4.5) and also belongs to the class of proximal splitting algorithms. The algorithm does not require projection operators which may be beneficial when C has a more complex structure. In the absence of a linear constraint, the method reduces to the Forward-Backward-Forward scheme. This method can solve problem [R1]. For the Huber problem [R3], PF-PDS can solve the mean-shift formulation of the problem[?].

1.2.4 Douglas-Rachford-type splitting method (DR)

This algorithm is the most general algorithm and can solve all regression problems [R1-R4]. It is based on Doulgas Rachford splitting in a higher-dimensional product space. It makes use of the proximity operators of the perspective of the LS objective (see[?] and[?]) The Huber problem with concomitant scale [R4] is reformulated as scaled Lasso problem with the mean shift[?] and thus solved in (n + d) dimensions.

1.3 References

[3] P. L. Combettes and J.C. Pesquet, Primal-Dual Splitting Algorithm for Solving Inclusions with Mixtures of Composite, Lipschitzian, and Parallel-Sum Type Monotone Operators, Set-Valued and Variational Analysis, vol. 20, pp. 307-330, 2012.

CHAPTER

TWO

GETTING STARTED

2.1 Installation

c-lasso is available on pip. You can install the package in the shell using

```
pip install c-lasso
```

To use the c-lasso package in Python, type

```
from classo import classo_problem
# one can add auxiliary functions as well such as random_data or csv_to_np
```

2.2 Dependencies

The *c-lasso* package depends on the following Python packages:

- `numpy`;
- `matplotlib`;
- `scipy`;
- `pandas`;
- `pytest` (for tests)

CHAPTER

THREE

EXAMPLES GALLERY

Below is a gallery of examples.

3.1 Basic example

Let's present what classo does when using its default parameters on synthetic data.

3.1.1 Import the package

```
import sys, os
from os.path import dirname, abspath

classo_dir = dirname(dirname(abspath("__file__")))
sys.path.append(classo_dir)
from classo import classo_problem, random_data
import numpy as np
```

3.1.2 Generate the data

This code snippet generates a problem instance with sparse ß in dimension d=100 (sparsity d_nonzero=5). The design matrix X comprises n=100 samples generated from an i.i.d standard normal distribution. The dimension of the constraint matrix C is d x k matrix. The noise level is =0.5. The input *zerosum=True* implies that C is the all-ones vector and Cß=0. The n-dimensional outcome vector y and the regression vector ß is then generated to satisfy the given constraints.

```
m, d, d_nonzero, k, sigma = 100, 200, 5, 1, 0.5
(X, C, y), sol = random_data(m, d, d_nonzero, k, sigma, zerosum=True, seed=1)
```

Remark : one can see the parameters that should be selected :

```
print(np.nonzero(sol))
```

Out:

```
(array([ 12, 157, 178, 181, 185]),)
```

3.1.3 Define the classo instance

Next we can define a default c-lasso problem instance with the generated data:

```
problem = classo_problem(X, y, C)
```

3.1.4 Check parameters

You can look at the generated problem instance by typing:

```
print(problem)
```

Out:

```
FORMULATION: R3

MODEL SELECTION COMPUTED:
     Stability selection

STABILITY SELECTION PARAMETERS:
     numerical_method : not specified
     method : first
     B = 50
     q = 10
     percent_nS = 0.5
     threshold = 0.7
     lamin = 0.01
     Nlam = 50
```

3.1.5 Solve optimization problems

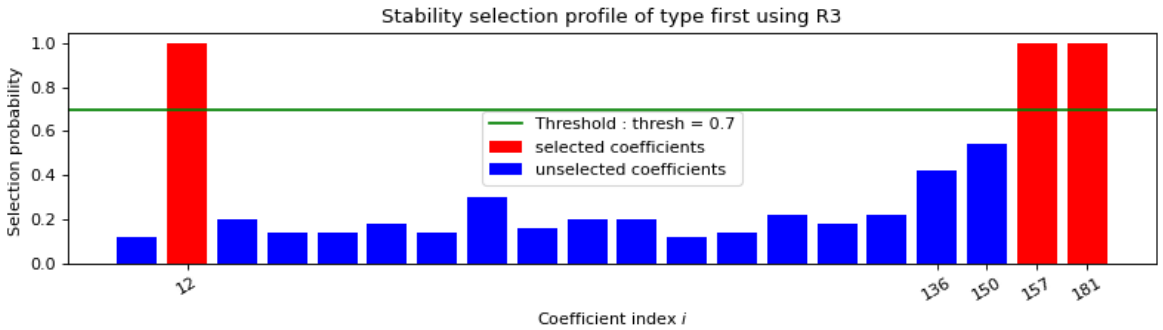
We only use stability selection as default model selection strategy. The command also allows you to inspect the computed stability profile for all variables at the theoretical

```
problem.solve()
```

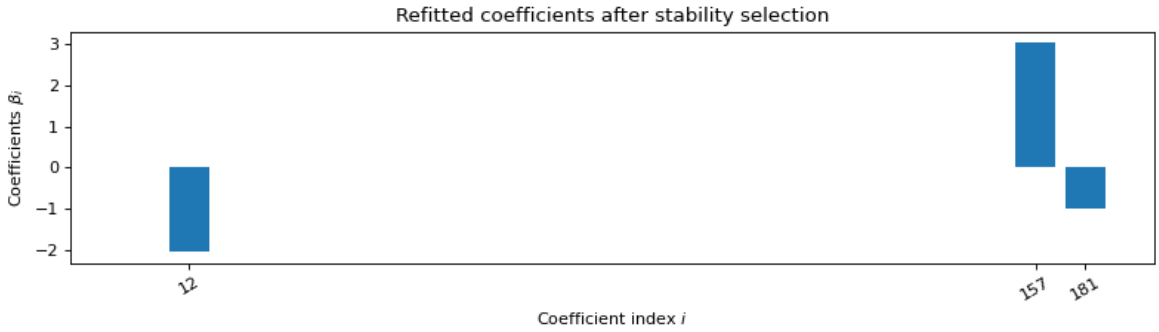
3.1.6 Visualisation

After completion, the results of the optimization and model selection routines can be visualized using

```
print(problem.solution)
```



-



Out:

```
STABILITY SELECTION :
  Selected variables :  12    157    181
  Running time :  0.809s
```

Total running time of the script: (0 minutes 1.643 seconds)

3.2 Advanced example

Let's present how one can specify different aspects of the problem formulation and model selection strategy on classo, using synthetic data.

3.2.1 Import the package

```
import sys, os
from os.path import join, dirname, abspath

classo_dir = dirname(dirname(abspath("__file__")))
sys.path.append(classo_dir)

from classo import classo_problem, random_data
import numpy as np
```

3.2.2 Generate the data

This code snippet generates a problem instance with sparse ß in dimension d=100 (sparsity d_nonzero=5). The design matrix X comprises n=100 samples generated from an i.i.d standard normal distribution. The dimension of the constraint matrix C is d x k matrix. The noise level is =0.5. The input *zerosum=True* implies that C is the all-ones vector and Cß=0. The n-dimensional outcome vector y and the regression vector ß is then generated to satisfy the given constraints. One can then see the parameters that should be selected.

```
m, d, d_nonzero, k, sigma = 100, 200, 5, 1, 0.5
(X, C, y), sol = random_data(
    m, d, d_nonzero, k, sigma, zerosum=True, seed=1, intercept=1.0
)
```

3.2.3 Create labels

This code snoppet creates labels that indicate where the solution ß should be nonzero.

```
labels = np.empty(d, dtype=str)
for i in range(d):
    if sol[i] == 0.0:
        labels[i] = "no_" + str(i)
    else:
        labels[i] = "yes_" + str(i)
```

3.2.4 Define the classo instance

Next we can define a default c-lasso problem instance with the generated data:

```
problem = classo_problem(X, y, C)
```

3.2.5 Change the parameters

Let's see some example of change in the parameters

```
problem.formulation.huber = True
problem.formulation.concomitant = False
problem.formulation.intercept = True
problem.model_selection.CV = True
problem.model_selection.LAMfixed = True
problem.model_selection.StabSelparameters.method = "max"
problem.model_selection.CVparameters.seed = 1
problem.model_selection.LAMfixedparameters.rescaled_lam = True
problem.model_selection.LAMfixedparameters.lam = 0.1
```

3.2.6 Check parameters

You can look at the generated problem instance by typing:

```
print(problem)
```

Out:

```
FORMULATION: R2

MODEL SELECTION COMPUTED:
     Cross Validation
     Stability selection
     Lambda fixed

LAMBDA FIXED PARAMETERS:
     numerical_method = not specified
     rescaled lam : True
     threshold : average of the absolute value of beta
     lam = 0.1

CROSS VALIDATION PARAMETERS:
```

(continues on next page)

(continued from previous page)

```
     numerical_method : not specified
     one-SE method : True
     Nsubset = 5
     lamin = 0.001
     Nlam = 80
     with log-scale

STABILITY SELECTION PARAMETERS:
     numerical_method : not specified
     method : max
     B = 50
     q = 10
     percent_nS = 0.5
     threshold = 0.7
     lamin = 0.01
     Nlam = 50
```

3.2.7 Solve optimization problems

We use stability selection as default model selection strategy.

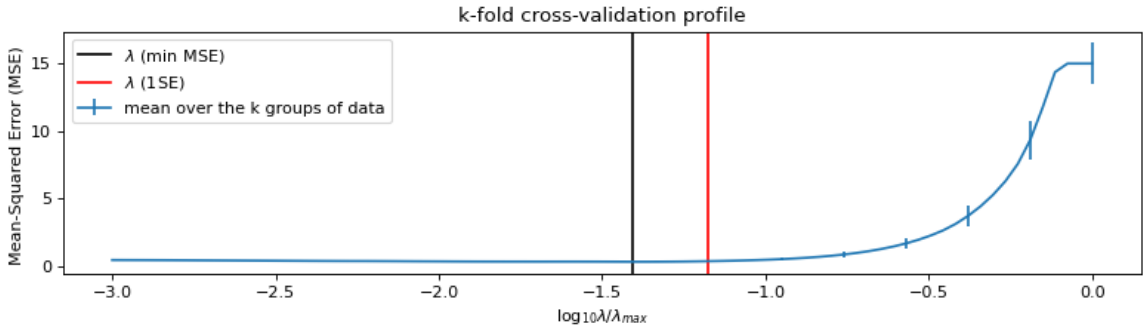
The command also allows you to inspect the computed stability profile for all variables at the theoretical . Two other model selections are computed here: computation of the solution for a fixed lambda; a path computation followed by a computation of the Approximation of the Leave-one Out error (ALO); a k-fold cross-validation.

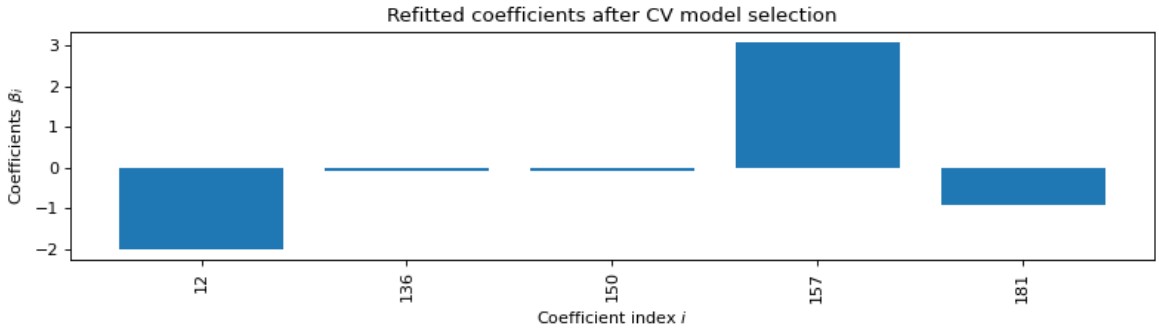
```
problem.solve()
```

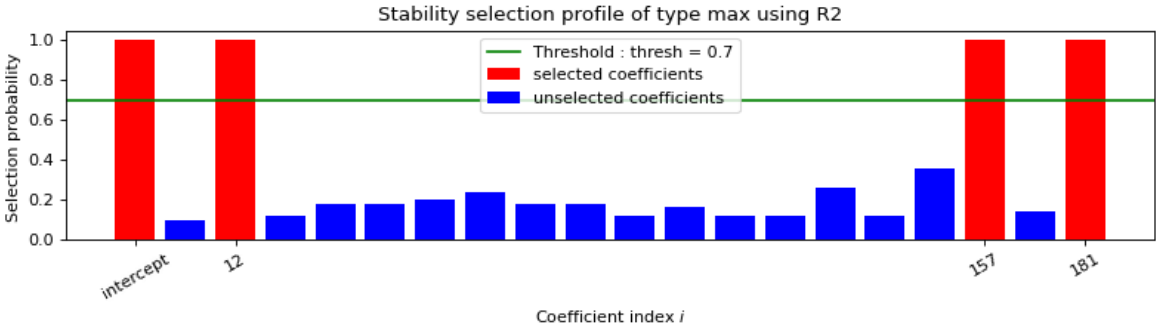
3.2.8 Visualisation

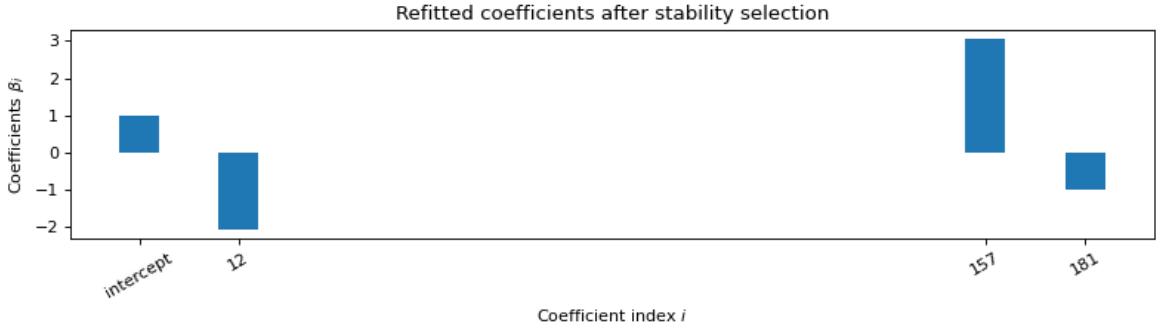
After completion, the results of the optimization and model selection routines can be visualized using

```
print(problem.solution)
```

- 

- 

- 

- 

Out:

```
LAMBDA FIXED :
  Selected variables :  intercept    12    157    181
  Running time :  0.041s

CROSS VALIDATION :
```

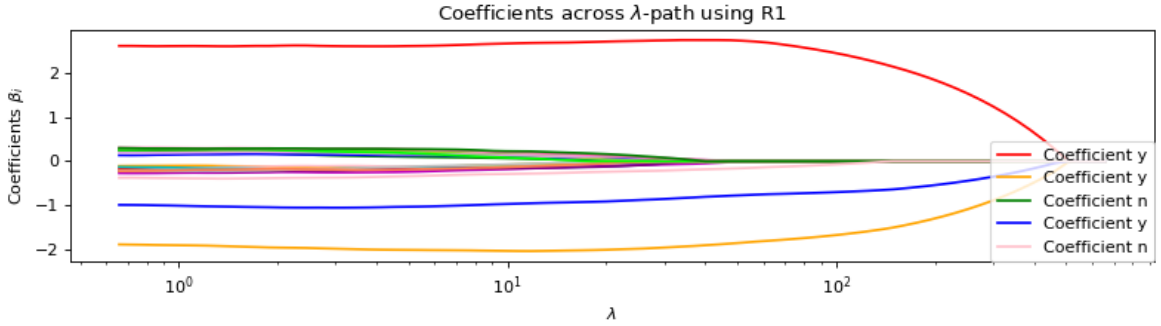
(continues on next page)

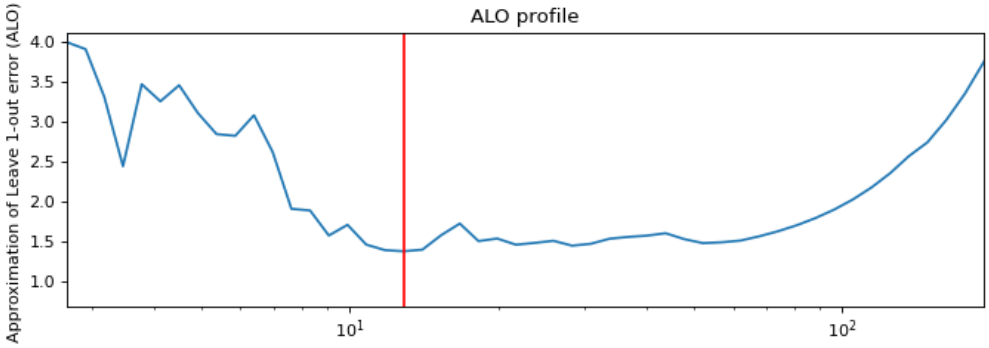
(continued from previous page)

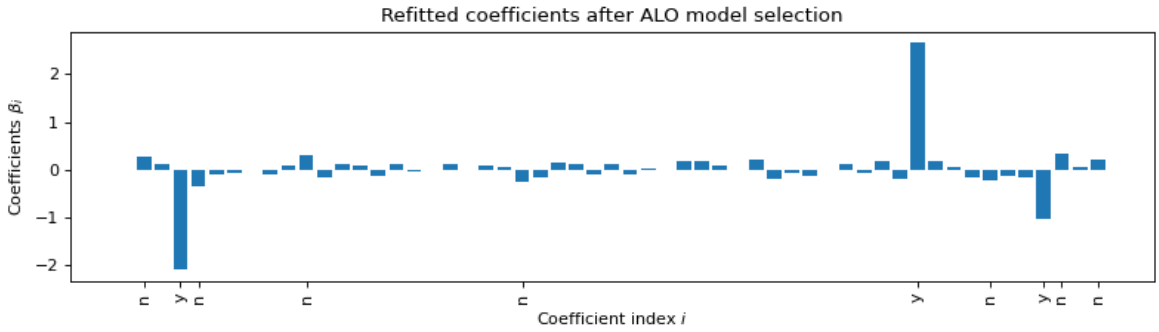
```
   Intercept : 1.0068530209340394
  Selected variables :  12    136    150    157    181
  Running time :  1.137s

STABILITY SELECTION :
  Selected variables :  intercept    12    157    181
  Running time :  4.533s
```

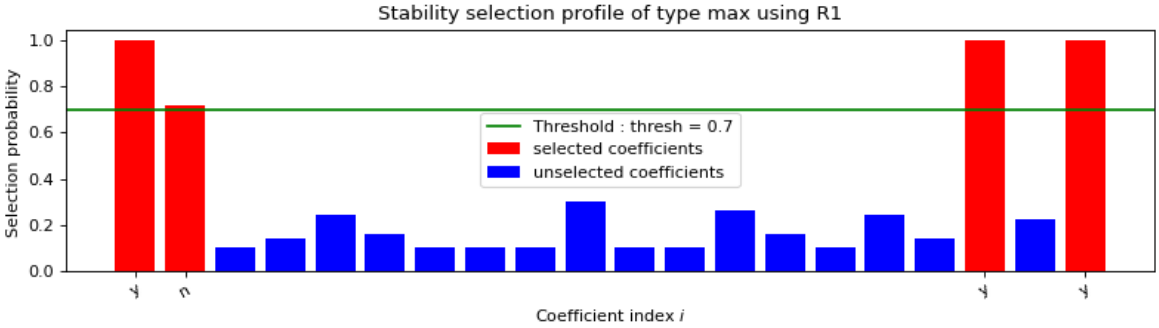
3.2.9 R1 formulation with ALO

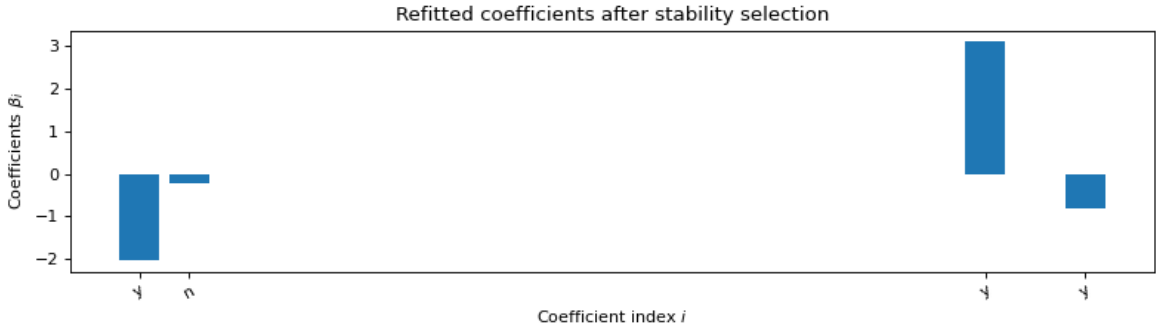
```
problem.data.label = labels
problem.formulation.intercept = False
problem.formulation.huber = False
problem.model_selection.ALO = True
problem.model_selection.CV = False
problem.model_selection.LAMfixed = False
problem.solve()
print(problem)
print(problem.solution)
```

- 

- 

- 

- 

- 

Out:

```
FORMULATION: R1

MODEL SELECTION COMPUTED:
     ALO
     Stability selection

ALO PARAMETERS:
     numerical_method : Path-Alg
     lamin = 0.001
     Nlam = 80


STABILITY SELECTION PARAMETERS:
     numerical_method : Path-Alg
     method : max
     B = 50
     q = 10
     percent_nS = 0.5
```

(continues on next page)

(continued from previous page)

```
    threshold = 0.7
    lamin = 0.01
    Nlam = 50


 ALO COMPUTATION :
   Selected variables :  n    n    n    y    n    n    n    n    n    n    n    n
↪n    n    n    n    n    n    n    n    n    n    n    n    n    n    n    n    n
↪ n    n    n    n    n    n    n    n    n    n    n    n    n    n    n    y    n
↪  n    n    n    n    n    y    n    n    n
   Running time :  0.245s

 STABILITY SELECTION :
   Selected variables :  y    n    y    y
   Running time :  3.203s
```

Total running time of the script: (0 minutes 11.030 seconds)

3.3 pH prediction using the 88 soils dataset

The next microbiome example considers a Soil dataset .

The data are generated thanks to a qiime2 workflow similar to a gneiss tutorial.

This workflow treat some files taken from gneiss GitHub.

The task is to predict pH concentration in the soil from microbial abundance data.

A similar analysis is also done in Tree-Aggregated Predictive Modeling of Microbiome Data. on another dataset

```
import sys, os
from os.path import join, dirname, abspath

classo_dir = dirname(dirname(abspath("__file__")))
sys.path.append(classo_dir)
from classo import classo_problem
import numpy as np
import pandas as pd
```

3.3.1 Load data

```
t = pd.read_csv("pH_data/qiime2/news/table.csv", index_col=0)
metadata = pd.read_table(
    "pH_data/qiime2/originals/88soils_modified_metadata.txt", index_col=0
)
y_uncent = metadata["ph"].values


X = t.values
label = t.columns


# second option to load the data
# import scipy.io as sio
```

(continues on next page)

(continued from previous page)

```
# pH = sio.loadmat("pH_data/matlab/pHData.mat")
# tax = sio.loadmat("pH_data/matlab/taxTablepHData.mat")["None"][0]
# X, y_uncent = pH["X"], pH["Y"].T[0]
# label = None

y = y_uncent - np.mean(y_uncent)  # Center Y
print(X.shape)
print(y.shape)
```

3.3.2 Set up c-lassso problem

```
problem = classo_problem(X, y, label=label)

problem.model_selection.StabSelparameters.method = "lam"
problem.model_selection.PATH = True
problem.model_selection.LAMfixed = True
problem.model_selection.PATHparameters.n_active = X.shape[1] + 1
```

3.3.3 Solve for R1

```
problem.formulation.concomitant = False
problem.solve()
print(problem, problem.solution)
```

3.3.4 Solve for R2

```
problem.formulation.huber = True
problem.solve()
print(problem, problem.solution)
```

3.3.5 Solve for R3

```
problem.formulation.concomitant = True
problem.formulation.huber = False
problem.solve()
print(problem, problem.solution)
```

3.3.6 Solve for R4

Remark : we reset the numerical method here, because it has been automatically set to ‘¨Path-Alg’ for previous computations, but for R4, “DR” is much better as explained in the documentation, R4 “Path-Alg” is a method for fixed lambda but is (paradoxically) bad to compute the lambda-path because of the absence of possible warm-start in this method

```
problem.model_selection.PATHparameters.numerical_method = "DR"
problem.formulation.huber = True
problem.solve()
print(problem, problem.solution)
```

Total running time of the script: (0 minutes 0.000 seconds)

3.4 BMI prediction using the COMBO dataset

We first consider the COMBO data set and show how to predict Body Mass Index (BMI) from microbial genus abundances and two non-compositional covariates using "filtered_data".

3.4.1 Import the package

```
import sys, os
from os.path import join, dirname, abspath

classo_dir = dirname(dirname(abspath("__file__")))
sys.path.append(classo_dir)
from classo import classo_problem, clr
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

3.4.2 Define how to read csv

```
def csv_to_np(file, begin=1, header=None):
    """Function to read a csv file and to create an ndarray with this

    Args:
        file (str): Name of csv file
        begin (int, optional): First colomn where it should read the matrix
        header (None or int, optional): Same parameter as in the function␣
↪:func:`pandas.read_csv`

    Returns:
        ndarray : matrix of the csv file
    """
    tab1 = pd.read_csv(file, header=header)
    return np.array(tab1)[:, begin:]
```

3.4.3 Load microbiome and covariate data X

```
data_dir = join(classo_dir, "examples/COMBO_data")
X0 = csv_to_np(join(data_dir, "complete_data/GeneraCounts.csv"), begin=0).
↪astype(float)
X_C = csv_to_np(join(data_dir, "CaloriData.csv"), begin=0).astype(float)
X_F = csv_to_np(join(data_dir, "FatData.csv"), begin=0).astype(float)
```

3.4.4 Load BMI measurements y

```
y = csv_to_np(join(data_dir, "BMI.csv"), begin=0).astype(float)[:, 0]
labels = csv_to_np(join(data_dir, "complete_data/GeneraPhylo.csv")).astype(str)[:, -1]
```

3.4.5 Normalize/transform data

```
y = y - np.mean(y)  # BMI data (n = 96)
X_C = X_C - np.mean(X_C, axis=0)  # Covariate data (Calorie)
X_F = X_F - np.mean(X_F, axis=0)  # Covariate data (Fat)
X0 = clr(X0, 1 / 2).T
```

3.4.6 Set up design matrix and zero-sum constraints for 45 genera

```
X = np.concatenate(
    (X0, X_C, X_F), axis=1
)  # Joint microbiome and covariate data and offset
label = np.concatenate([labels, np.array(["Calorie", "Fat"])])
C = np.ones((1, len(X[0])))
C[0, -1], C[0, -2] = 0.0, 0.0
```

3.4.7 Set up c-lassso problem

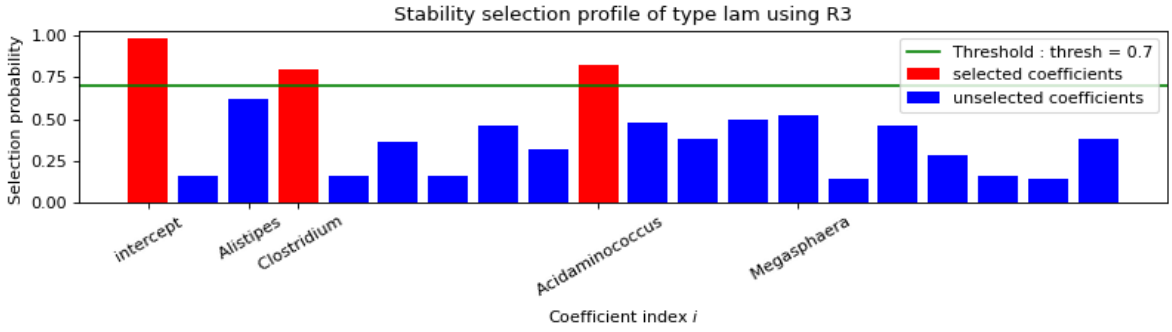
```
problem = classo_problem(X, y, C, label=label)
problem.formulation.intercept = True
```

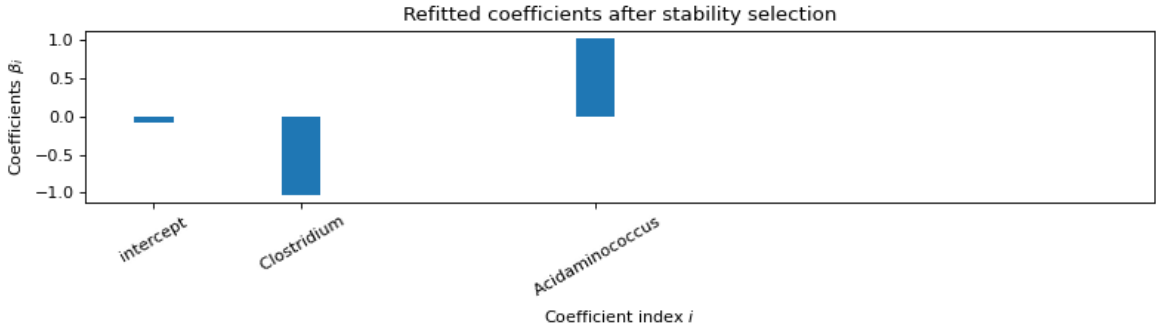
Use stability selection with theoretical lambda [Combettes & Müller, 2020b]

```
problem.model_selection.StabSelparameters.method = "lam"
problem.model_selection.StabSelparameters.threshold_label = 0.5
```

3.4.8 Use formulation R3

```
problem.formulation.concomitant = True

problem.solve()
print(problem)
print(problem.solution)
```

- 

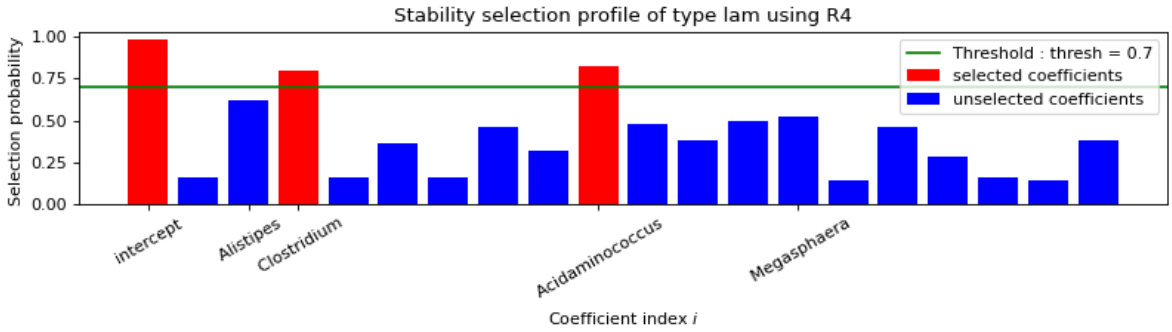
- 

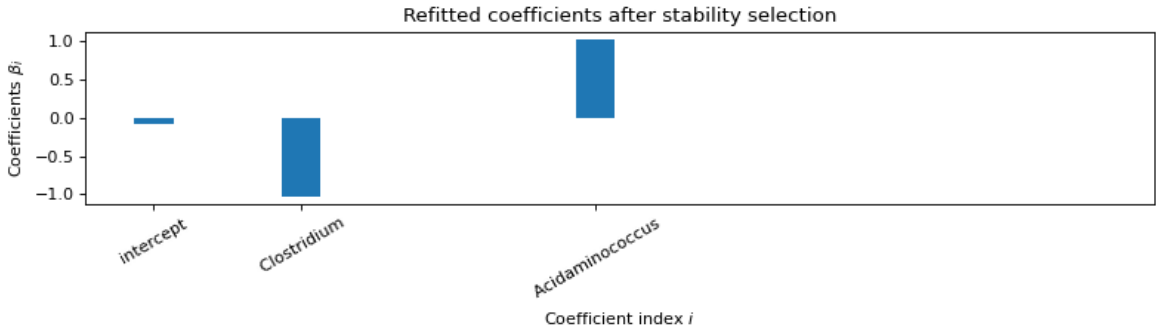
Out:

```
FORMULATION: R3

MODEL SELECTION COMPUTED:
     Stability selection

STABILITY SELECTION PARAMETERS:
     numerical_method : Path-Alg
     method : lam
     B = 50
     q = 10
     percent_nS = 0.5
     threshold = 0.7
     lam = theoretical
     theoretical_lam = 0.2818


 STABILITY SELECTION :
   Selected variables :  intercept     Clostridium     Acidaminococcus
   Running time :  0.478s
```

3.4.9 Use formulation R4

```
problem.data.label = label
problem.formulation.huber = True
problem.formulation.concomitant = True

problem.solve()
print(problem)
print(problem.solution)
```



- 

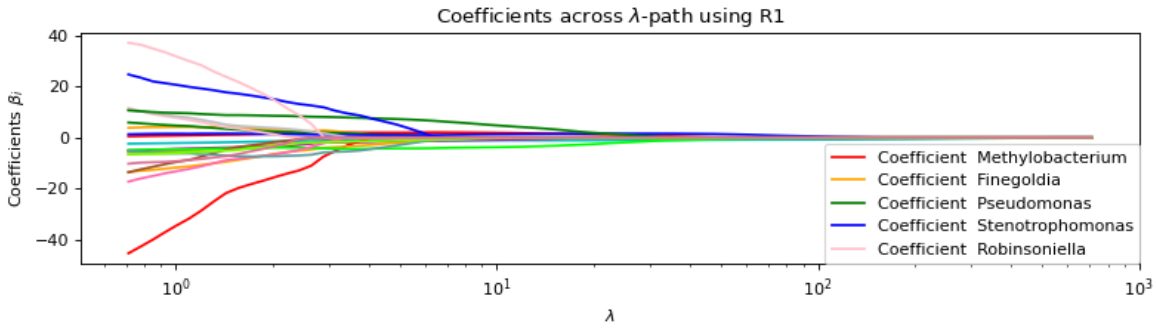
Out:

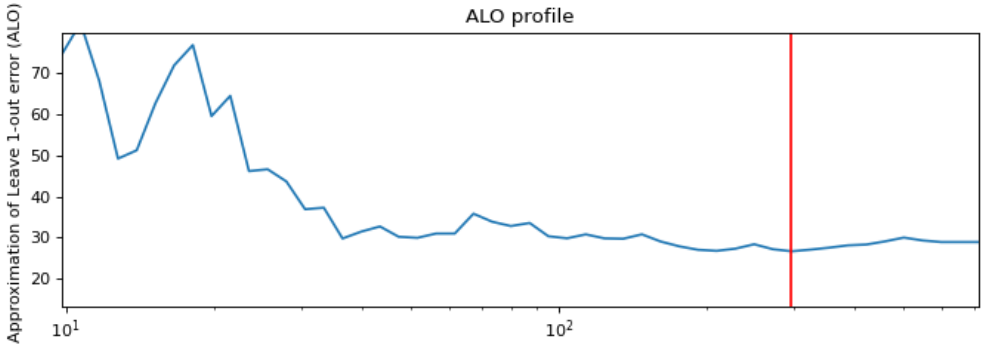
```
FORMULATION: R4

MODEL SELECTION COMPUTED:
     Stability selection

STABILITY SELECTION PARAMETERS:
     numerical_method : Path-Alg
     method : lam
     B = 50
     q = 10
     percent_nS = 0.5
     threshold = 0.7
     lam = theoretical
     theoretical_lam = 0.2818


 STABILITY SELECTION :
   Selected variables :  intercept     Clostridium     Acidaminococcus
   Running time :  0.698s
```

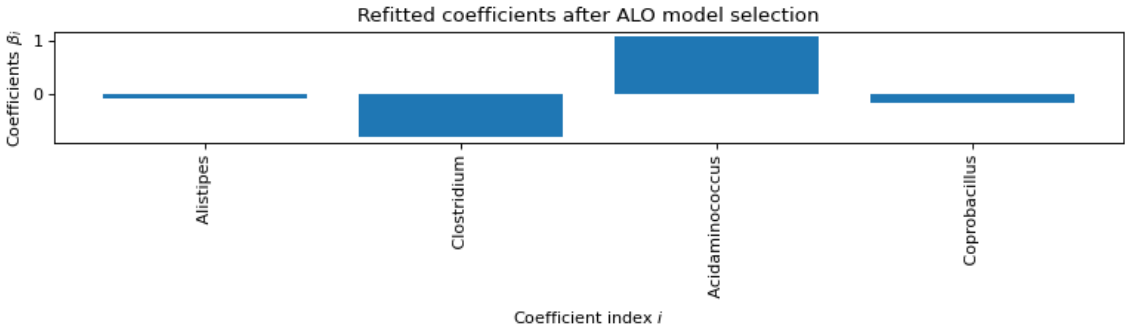
3.4.10 Use formulation R1 with ALO

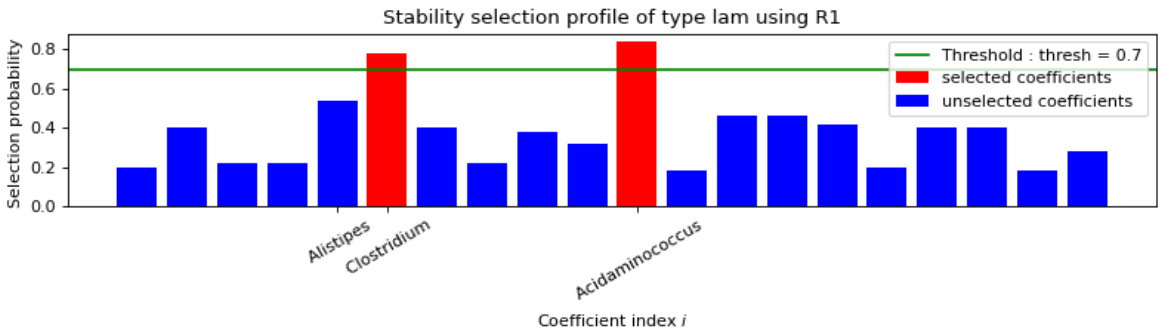
ALO is implemented only for R1 without intercept for now.

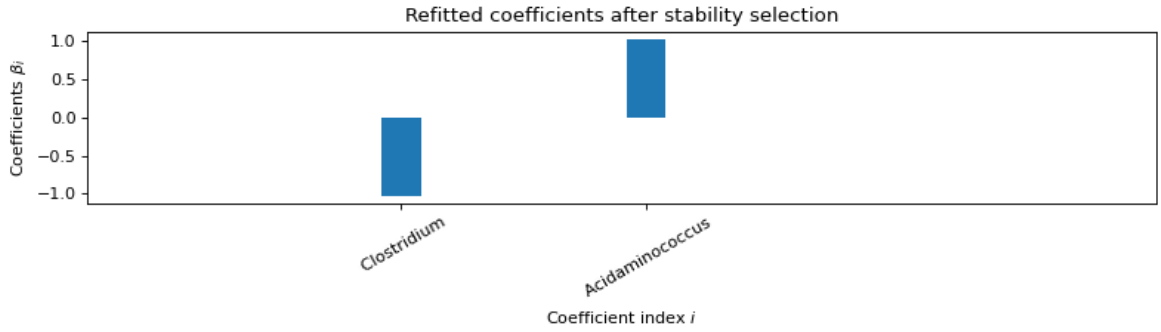
```
problem.data.label = label
problem.formulation.intercept = False
problem.formulation.huber = False
problem.formulation.concomitant = False
problem.model_selection.ALO = True

problem.solve()
print(problem)
print(problem.solution)
```

- 

- 

- 

- 

- 

Out:

```
FORMULATION: R1

MODEL SELECTION COMPUTED:
     ALO
     Stability selection

ALO PARAMETERS:
     numerical_method : Path-Alg
     lamin = 0.001
     Nlam = 80


STABILITY SELECTION PARAMETERS:
     numerical_method : Path-Alg
     method : lam
     B = 50
     q = 10
     percent_nS = 0.5
     threshold = 0.7
     lam = theoretical
     theoretical_lam = 0.2818


 ALO COMPUTATION :
   Selected variables :   Alistipes     Clostridium     Acidaminococcus
↪Coprobacillus
   Running time :  0.144s

 STABILITY SELECTION :
   Selected variables :   Clostridium     Acidaminococcus
   Running time :  0.334s
```

Total running time of the script: (0 minutes 3.469 seconds)

3.5 Ocean salinity prediction based on marin microbiome data

We repoduce an example of prediction of ocean salinity over ocean microbiome data that has been introduced in this article, where the R package trac (which uses c-lasso) has been used.

The data come originally from trac, then it is preprocessed in python in this notebook.

Bien, J., Yan, X., Simpson, L. and Müller, C. (2020). Tree-Aggregated Predictive Modeling of Microbiome Data :

"Integrative marine data collection efforts such as Tara Oceans (Sunagawa et al., 2020) or the Simons CMAP (https://simonscmap.com) provide the means to investigate ocean ecosystems on a global scale. Using Tara's environmental and microbial survey of ocean surface water (Sunagawa, 2015), we next illustrate how trac can be used to globally connect environmental covariates and the ocean microbiome. As an example, we learn a global predictive model of ocean salinity from n = 136 samples and p = 8916 miTAG OTUs (Logares et al., 2014). trac identifies four taxonomic aggregations, the kingdom bacteria and the phylum Bacteroidetes being negatively associated and the classes Alpha and Gammaproteobacteria being positively associated with marine salinity.

```
import sys, os
from os.path import join, dirname, abspath

classo_dir = dirname(dirname(abspath("__file__")))
sys.path.append(classo_dir)
from classo import classo_problem
import matplotlib.pyplot as plt
import numpy as np
```

3.5.1 Load data

```
data_dir = join(classo_dir, "examples/Tara")

data = np.load(join(data_dir, "tara.npz"))

x = data["x"]
label = data["label"]
y = data["y"]
tr = data["tr"]

A = np.load(join(data_dir, "A.npy"))
```

3.5.2 Preprocess: taxonomy aggregation

```
label_short = np.array([l.split("::")[-1] for l in label])

pseudo_count = 1
X = np.log(pseudo_count + x)
nleaves = np.sum(A, axis=0)
logGeom = X.dot(A) / nleaves
```

3.5.3 Cross validation and Path Computation

```
problem = classo_problem(logGeom[tr], y[tr], label=label_short)

problem.formulation.w = 1 / nleaves
problem.formulation.intercept = True
problem.formulation.concomitant = False

problem.model_selection.StabSel = False
problem.model_selection.PATH = True
problem.model_selection.CV = True
problem.model_selection.CVparameters.seed = (
    6  # one could change logscale, Nsubset, oneSE
)
print(problem)

problem.solve()
print(problem.solution)


selection = problem.solution.CV.selected_param[1:]  # exclude the intercept
print(label[selection])
```

Prediction plot

```
te = np.array([i for i in range(len(y)) if not i in tr])
alpha = problem.solution.CV.refit
yhat = logGeom[te].dot(alpha[1:]) + alpha[0]

M1, M2 = max(y[te]), min(y[te])
plt.plot(yhat, y[te], "bo", label="sample of the testing set")
plt.plot([M1, M2], [M1, M2], "k-", label="identity")
plt.xlabel("predictor yhat"), plt.ylabel("real y"), plt.legend()
plt.tight_layout()
```

3.5.4 Stability selection

```
problem = classo_problem(logGeom[tr], y[tr], label=label_short)

problem.formulation.w = 1 / nleaves
problem.formulation.intercept = True
problem.formulation.concomitant = False


problem.model_selection.PATH = False
problem.model_selection.CV = False
# can change q, B, nS, method, threshold etc in problem.model_selection.
↪StabSelparameters

problem.solve()

print(problem, problem.solution)
```

(continues on next page)

(continued from previous page)

```
selection = problem.solution.StabSel.selected_param[1:]  # exclude the intercept
print(label[selection])
```

Prediction plot

```
te = np.array([i for i in range(len(y)) if not i in tr])
alpha = problem.solution.StabSel.refit
yhat = logGeom[te].dot(alpha[1:]) + alpha[0]

M1, M2 = max(y[te]), min(y[te])
plt.plot(yhat, y[te], "bo", label="sample of the testing set")
plt.plot([M1, M2], [M1, M2], "k-", label="identity")
plt.xlabel("predictor yhat"), plt.ylabel("real y"), plt.legend()
plt.tight_layout()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.6 pH prediction using the Central Park soil dataset

The next microbiome example considers the [Central Park Soil dataset](./examples/CentralParkSoil) from [Ramirez et al.](https://royalsocietypublishing.org/doi/full/10.1098/rspb.2014.1988). The sample locations are shown in the Figure on the right.)

The task is to predict pH concentration in the soil from microbial abundance data.

This task is also done in Tree-Aggregated Predictive Modeling of Microbiome Data.

3.6.1 Import the package

```
import sys, os
from os.path import join, dirname, abspath

classo_dir = dirname(dirname(abspath("__file__")))
sys.path.append(classo_dir)

from classo import classo_problem
import matplotlib.pyplot as plt
import numpy as np
```

3.6.2 Load data

```
data_dir = join(classo_dir, "examples/CentralParkSoil")
data = np.load(join(data_dir, "cps.npz"))

x = data["x"]
label = data["label"]
y = data["y"]

A = np.load(join(data_dir, "A.npy"))
```

3.6.3 Preprocess: taxonomy aggregation

```
label_short = np.array([l.split("::")[-1] for l in label])

pseudo_count = 1
X = np.log(pseudo_count + x)
nleaves = np.sum(A, axis=0)
logGeom = X.dot(A) / nleaves

n, d = logGeom.shape

tr = np.random.permutation(n)[: int(0.8 * n)]
```

3.6.4 Cross validation and Path Computation

```
problem = classo_problem(logGeom[tr], y[tr], label=label_short)

problem.formulation.w = 1 / nleaves
problem.formulation.intercept = True
problem.formulation.concomitant = False

problem.model_selection.StabSel = False
problem.model_selection.PATH = True
problem.model_selection.CV = True
problem.model_selection.CVparameters.seed = (
    6  # one could change logscale, Nsubset, oneSE
)
print(problem)

problem.solve()
print(problem.solution)

selection = problem.solution.CV.selected_param[1:]  # exclude the intercept
print(label[selection])
```

Prediction plot

```
te = np.array([i for i in range(len(y)) if not i in tr])
alpha = problem.solution.CV.refit
yhat = logGeom[te].dot(alpha[1:]) + alpha[0]

M1, M2 = max(y[te]), min(y[te])
plt.plot(yhat, y[te], "bo", label="sample of the testing set")
plt.plot([M1, M2], [M1, M2], "k-", label="identity")
plt.xlabel("predictor yhat"), plt.ylabel("real y"), plt.legend()
plt.tight_layout()
```

3.6.5 Stability selection

```
problem = classo_problem(logGeom[tr], y[tr], label=label_short)

problem.formulation.w = 1 / nleaves
problem.formulation.intercept = True
problem.formulation.concomitant = False


problem.model_selection.PATH = False
problem.model_selection.CV = False
# can change q, B, nS, method, threshold etc in problem.model_selection.
↪StabSelparameters

problem.solve()

print(problem, problem.solution)

selection = problem.solution.StabSel.selected_param[1:]  # exclude the intercept
print(label[selection])
```

Prediction plot

```
te = np.array([i for i in range(len(y)) if not i in tr])
alpha = problem.solution.StabSel.refit
yhat = logGeom[te].dot(alpha[1:]) + alpha[0]

M1, M2 = max(y[te]), min(y[te])
plt.plot(yhat, y[te], "bo", label="sample of the testing set")
plt.plot([M1, M2], [M1, M2], "k-", label="identity")
plt.xlabel("predictor yhat"), plt.ylabel("real y"), plt.legend()
plt.tight_layout()
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER

FOUR

STRUCTURE OF PROBLEM INSTANCE

The package is organized as follow : There is a main class called *classo_problem*, that contains a lot of information about the problem, and once the problem is solved, it will also contains the solution.

Here is the global structure of the problem instance:

A *classo_problem* instance contains a *Data* instance, a *Formulation* instance, a *Model_selection* instance and a *Solution* instance.

A *Model_selection* instance contains the instances : *PATHparameters*, *CVparameters*, *StabSelparameters*, *LAMfixedparameters*.

A *Solution* instance, once is computed, contains the instances : *solution_PATH*, *solution_CV*, *solution_StabSel*, *solution_LAMfixed*.

Classes

classo_problem(X, y[, C, Tree, label])	Class that contains all the information about the problem.
classo_problem.solve()	Method that solves every model required in the attributes of the problem instance and update the attribute `solution` with the characteristics of the solution.
Data(X, y, C[, Tree, label])	Class that contains the data of the problem ie where matrices and labels are stored.
Formulation()	Class that contains the information about the formulation of the problem namely, the type of formulation (R1, R2, R3, R4, C1, C2) and its parameters like rho, the weigths and the presence of an intercept.
Model_selection([method])	Class that contains information about the model selections to perform.
PATHparameters([method])	Class that contains the parameters to compute the lasso-path.
CVparameters([method])	Class that contains the parameters to compute the cross-validation.
StabSelparameters([method])	Class that contains the parameters to compute the stability selection.
LAMfixedparameters([method])	Class that contains the parameters to compute the lasso for a fixed lambda.
Solution()	Class that contains characteristics of the solution of the model_selections that are computed Before using the method `solve()` , its componant are empty/null.

continues on next page

Table 1 – continued from previous page

solution_PATH(matrices, param, formulation, ...)	Class that contains characteristics of the lasso-path computed, which also contains representation method that plot the graphic of this lasso-path.
solution_ALO(matrices, param, formulation, ...)	Class that contains characteristics of the lasso-path computed, which also contains representation method that plot the graphic of this lasso-path.
solution_CV(matrices, param, formulation, ...)	Class that contains characteristics of the cross validation computed, which also contains a representation method that plot the selected parameters and the solution of the not-sparse problem on the selected variables set.
solution_CV.graphic([se_max, save, ...])	Method to plot the graphic showing mean squared error over along lambda path once cross validation is computed.
solution_StabSel(matrices, param, ...)	Class that contains characteristics of the stability selection computed, which also contains a representation method that plot the selected parameters, the solution of the not-sparse problem on the selected variables set, and the stability plot.
solution_LAMfixed(matrices, param, ...)	Class that contains characteristics of the lasso computed which also contains a representation method that plot this solution.

4.1 Class classo_problem

class classo.solver.**classo_problem**(*X*, *y*, *C=None*, *Tree=None*, *label=None*)

Class that contains all the information about the problem. It also has a representation method so one can print it.

Parameters

- **X** (*ndarray*) – Matrix representing the data of the problem.
- **y** (*ndarray*) – Vector representing the output of the problem.
- **C** (*str or ndarray, optional*) – Matrix of constraints to the problem. If it is 'zero-sum' then the corresponding attribute will be all-one matrix. Default value : 'zero-sum'
- **label** (*list, optional*) – list of the labels of each variable. If None, then label are just indices. Default value : None

data

object containing the data (matrices) of the problem. Namely : X, y, C and the labels.

Type *Data*

formulation

object containing the info about the formulation of the minimization problem we solve.

Type *Formulation*

model_selection

object containing the parameters we need to do variable selection.

Type *Model_selection*

solution

object giving caracteristics of the solution of the model_selection that is asked. Before using the method *solve()* , its componant are empty/null.

Type *Solution*

numerical_method
name of the numerical method that is used, it can be : 'Path-Alg' (path algorithm) , 'P-PDS' (Projected primal-dual splitting method) , 'PF-PDS' (Projection-free primal-dual splitting method) or 'DR' (Douglas-Rachford-type splitting method). Default value : 'not specified', which means that the function *choose_numerical_method()* will choose it accordingly to the formulation.

Type str

classo_problem.**solve**()
Method that solves every model required in the attributes of the problem instance and update the attribute *solution* with the characteristics of the solution.

classo.solver.**choose_numerical_method**(*method*, *model*, *formulation*, *StabSelmethod=None*, *lam=None*)
Annex function in order to choose the right numerical method, if the given one is invalid. In general, it will choose one of the possible optimization scheme for a given formulation. When several computation modes are possible, the rules are as follow :

If possible, always use "Path-Alg", except for fixed lambdas smaller than 0.05 and for R4 where Path-Alg does not compute the path (paradoxically).

Else, it uses "DR".

Parameters

- **method** (*str*) – input method that is possibly wrong and should be changed.
- **the method is valid for this formulation** (*If*) –
- **will not be changed.** (*it*) –
- **model** (*str*) – Computation mode. Can be "PATH", "StabSel", "CV" or "LAM".
- **formulation** (Formulation) – object containing the info about the formulation of the minimization problem we solve.
- **StabSelmethod** (*str, optional*) – if model is "StabSel", it can be "first" , "lam" or "max".
- **lam** (*float, optional*) – value of lam (fractional L1 penalty).

Returns : str : method that should be used. Can be "Path-Alg", "DR", "P-PDS" or "PF-PDS"

4.2 Class Data

class classo.solver.**Data**(*X*, *y*, *C*, *Tree=None*, *label=None*)
Class that contains the data of the problem ie where matrices and labels are stored.

Parameters

- **X** (*ndarray*) – Matrix representing the data of the problem.
- **y** (*ndarray*) – Vector representing the output of the problem.
- **C** (*str or array, optional*) – Matrix of constraints to the problem. If it is 'zero-sum' then the corresponding attribute will be all-one matrix.
- **label** (*list, optional*) – list of the labels of each variable. If None, then labels are juste the indices. Default value : None

- **Tree** (*skbio.TreeNode, optional*) – taxonomic tree, if not None, then the matrices X and C and the labels will be changed.

X
Matrix representing the data of the problem.

Type ndarray

y
Vector representing the output of the problem.

Type ndarray

C
Matrix of constraints to the problem. If it is 'zero-sum' then the corresponding attribute will be all-one matrix.

Type str or array, optional

label
list of the labels of each variable. If None, then labels are juste the indices.

Type list

tree
taxonomic tree.

Type skbio.TreeNode or None

4.3 Class Formulation

class classo.solver.**Formulation**
Class that contains the information about the formulation of the problem namely, the type of formulation (R1, R2, R3, R4, C1, C2) and its parameters like rho, the weigths and the presence of an intercept. The type of formulation is encoded with boolean huber concomitant and classification with the rule:

False False False = R1

True False False = R2

False True False = R3

True True False = R4

False False True = C1

True False True = C2

It also has a representation method so one can print it.

huber
True if the formulation of the problem should be robust. Default value : False

Type bool

concomitant
True if the formulation of the problem should be with an M-estimation of sigma. Default value : True

Type bool

classification
True if the formulation of the problem should be classification (if yes, then it will not be concomitant). Default value : False

Type bool

rho
Value of rho for R2 and R4 formulations. Default value : 1.345

Type float

scale_rho
If set to True, it will become rho * sqrt(mean(y**2)) while solving the problem so that it lives on the scale of y and also usefull so that we don't have the problem with the non strict convexity (i.e. at least one sample is on the quadratic mode of the huber loss function) as long as rho is higher than one. Default value : True

Type bool

rho_scaled
Actual rho after solving Default value : Not defined

Type float

rho_classification
value of rho for huberized hinge loss function for classification ie C2 (it has to be strictly smaller then 1). Default value : -1.

Type float

e
value of e in concomitant formulation. If 'n/2' then it becomes n/2 during the method `solve()`, same for 'n'. Default value : 'n' if huber formulation ; 'n/2' else

Type float or string

w
array of size d with the weights of the L1 penalization. This has to be positive. Default value : None (which makes it the 1,...,1 vector)

Type numpy ndarray

intercept
set to true if we should use an intercept. Default value : False

Type bool

4.4 Class Model_selection

class classo.solver.**Model_selection**(*method='not specified'*)
Class that contains information about the model selections to perform. It contains boolean that states which one will be computed. It also contains objects that contain parameters of each computation modes. It also has a representation method so one can print it.

PATH
True if path should be computed. Default value : False

Type bool

PATHparameters
object containing parameters to compute the lasso-path.

Type *PATHparameters*

ALO
True if path should be computed. Default value : False

Type bool

ALOparameters
object containing parameters to compute the ALO for c-lasso.

Type *ALOparameters*

CV
True if Cross Validation should be computed. Default value : False

Type bool

CVparameters
object containing parameters to compute the cross-validation.

Type *CVparameters*

StabSel
True if Stability Selection should be computed. Default value : True

Type boolean

StabSelparameters
object containing parameters to compute the stability selection.

Type *StabSelparameters*

LAMfixed
True if solution for a fixed lambda should be computed. Default value : False

Type boolean

LAMfixedparameters
object containing parameters to compute the lasso for a fixed lambda.

Type *LAMfixedparameters*

4.5 Classes used in Model_selection

class classo.solver.**PATHparameters**(*method='not specified'*)
Class that contains the parameters to compute the lasso-path. It also has a representation method so one can print it.

numerical_method
name of the numerical method that is used, it can be : 'Path-Alg' (path algorithm) , 'P-PDS' (Projected primal-dual splitting method), 'PF-PDS' (Projection-free primal-dual splitting method) or 'DR' (Douglas-Rachford-type splitting method). Default value : 'not specified', which means that the function *choose_numerical_method()* will choose it accordingly to the formulation

Type str

n_active
if it is higher than 0, then the algo stops computing the path when n_active variables are active.

Type int

Then the solution does not change from this point.
Default value : 0

lambdas
list of rescaled lambdas for computing lasso-path. Default value : None, which means line space between 1 and *lamin* and *Nlam* points, with logarithm scale or not depending on *logscale*.

Type numpy.ndarray

Nlam
number of points in the lambda-path if *lambdas* is still None (default). Default value : 80

Type int

lamin
lambda minimum if *lambdas* is still None (default). Default value : 1e-3

Type float

logscale
when *lambdas* is set to None (default), this parameters tells if it should be set with log scale or not. Default value : True

Type bool

plot_sigma
if True then the representation method of the solution will also plot the sigma-path if it is computed (formulation R3 or R4). Default value : True

Type bool

label
labels on each coefficient.

Type numpy.ndarray of str

class classo.solver.**ALOparameters**(*method='not specified'*)
Class that contains the parameters to compute the lasso-path, then the Approximation of Leave one-out error. It also has a representation method so one can print it.

numerical_method
name of the numerical method that is used, it can be : 'Path-Alg' (path algorithm) , 'P-PDS' (Projected primal-dual splitting method), 'PF-PDS' (Projection-free primal-dual splitting method) or 'DR' (Douglas-Rachford-type splitting method). Default value : 'not specified', which means that the function *choose_numerical_method()* will choose it accordingly to the formulation

Type str

n_active
if it is higher than 0, then the algo stops computing the path when n_active variables are active.

Type int

Then the solution does not change from this point.
Default value : 0

lambdas
list of rescaled lambdas for computing lasso-path. Default value : None, which means line space between 1 and *lamin* and *Nlam* points, with logarithm scale or not depending on *logscale*.

Type numpy.ndarray

Nlam
number of points in the lambda-path if *lambdas* is still None (default). Default value : 80

Type int

lamin
lambda minimum if *lambdas* is still None (default). Default value : 1e-3

Type float

logscale
when *lambdas* is set to None (default), this parameters tells if it should be set with log scale or not. Default value : True

Type bool

plot_sigma
if True then the representation method of the solution will also plot the sigma-path if it is computed (formulation R3 or R4). Default value : True

Type bool

label
labels on each coefficient.

Type numpy.ndarray of str

class classo.solver.**CVparameters**(*method='not specified'*)
Class that contains the parameters to compute the cross-validation. It also has a representation method so one can print it.

seed
Seed for random values, for an equal seed, the result will be the same. If set to False/None: pseudo-random seed. Default value : 0

Type bool or int, optional

numerical_method
name of the numerical method that is used, can be : 'Path-Alg' (path algorithm) , 'P-PDS' (Projected primal-dual splitting method), 'PF-PDS' (Projection-free primal-dual splitting method) or 'DR' (Douglas-Rachford-type splitting method). Default value : 'not specified', which means that the function *choose_numerical_method()* will choose it accordingly to the formulation.

Type str

lambdas
list of rescaled lambdas for computing lasso-path. Default value : None, which means line space between 1 and *lamin* and *Nlam* points, with logarithm scale or not depending on *logscale*.

Type numpy.ndarray

Nlam
number of points in the lambda-path if *lambdas* is still None (default). Default value : 80

Type int

lamin
lambda minimum if *lambdas* is still None (default). Default value : 1e-3

Type float

logscale
when *lambdas* is set to None (default), this parameters tells if it should be set with log scale or not. Default value : True

Type bool

oneSE
if set to True, the selected lambda is computed with method 'one-standard-error'. Default value : True

Type bool

Nsubset
number of subset in the cross validation method. Default value : 5

Type int

class classo.solver.**StabSelparameters**(*method='not specified'*)

Class that contains the parameters to compute the stability selection. It also has a representation method so one can print it.

seed

Seed for random values, for an equal seed, the result will be the same. If set to False/None: pseudo-random seed. Default value : 123

Type bool or int, optional

numerical_method

name of the numerical method that is used, can be : 'Path-Alg' (path algorithm) , 'P-PDS' (Projected primal-dual splitting method) , 'PF-PDS' (Projection-free primal-dual splitting method) or 'DR' (Douglas-Rachford-type splitting method). Default value : 'not specified', which means that the function *choose_numerical_method()* will choose it accordingly to the formulation.

Type str

lam

(only used if *method* = 'lam') lam for which the lasso should be computed. Default value : 'theoretical' which mean it will be equal to *theoretical_lam* once it is computed.

Type float or str

rescaled_lam

(only used if *method* = 'lam') False if lam = lambda, False if lam = lambda/lambdamax which is between 0 and 1. If False and lam = 'theoretical' , then it will take the value n*theoretical_lam. Default value : True

Type bool

theoretical_lam

(only used if *method* = 'lam') Theoretical lam. Default value : 0.0 (once it is not computed yet, it is computed thanks to the function *theoretical_lam()* used in *classo_problem.solve()*).

Type float

method

'first', 'lam' or 'max' depending on the type of stability selection we do. Default value : 'first'

Type str

B

number of subsample considered. Default value : 50

Type int

q

number of selected variable per subsample. Default value : 10

Type int

percent_nS

size of subsample relatively to the total amount of sample. Default value : 0.5

Type float

lamin

lamin when computing the lasso-path for method 'max'. Default value : 1e-2

Type float

hd
if set to True, then the 'max' will stop when it reaches n-k actives variables. Default value : False

Type bool

threshold
threshold for stability selection. Default value : 0.7

Type float

threshold_label
threshold to know when the label should be plot on the graph. Default value : 0.4

Type float

class classo.solver.**LAMfixedparameters**(*method='not specified'*)
Class that contains the parameters to compute the lasso for a fixed lambda. It also has a representation method so one can print it.

numerical_method
name of the numerical method that is used, can be : 'Path-Alg' (path algorithm) , 'P-PDS' (Projected primal-dual splitting method) , 'PF-PDS' (Projection-free primal-dual splitting method) or 'DR' (Douglas-Rachford-type splitting method). Default value : 'not specified', which means that the function *choose_numerical_method()* will choose it accordingly to the formulation

Type str

lam
lam for which the lasso should be computed. Default value : 'theoretical' which mean it will be equal to *theoretical_lam* once it is computed

Type float or str

rescaled_lam
False if lam = lambda, True if lam = lambda/lambdamax which is between 0 and 1. If False and lam = 'theoretical' , then it will takes the value n*theoretical_lam. Default value : True

Type bool

theoretical_lam
Theoretical lam. Default value : 0.0 (once it is not computed yet, it is computed thanks to the function *theoretical_lam()* used in *classo_problem.solve()*).

Type float

threshold
Threshold such that the parameters i selected or the ones such as the absolute value of beta[i] is greater than the threshold. If None, then it will be set to the average of the absolute value of beta. Default value : None

Type float

4.6 Class Solution

class classo.solver.**Solution**

Class that contains characteristics of the solution of the model_selections that are computed Before using the method `solve()` , its componant are empty/null. It also has a representation method so one can print it.

PATH

Solution components of the model PATH.

Type *solution_PATH*

CV

Solution components of the model CV.

Type *solution_CV*

StabelSel

Solution components of the model StabSel.

Type *solution_StabSel*

LAMfixed

Solution components of the model LAMfixed.

Type *solution_LAMfixed*

4.7 Classes used in Solution

class classo.solver.**solution_PATH**(*matrices*, *param*, *formulation*, *numerical_method*, *label*)

Class that contains characteristics of the lasso-path computed, which also contains representation method that plot the graphic of this lasso-path.

BETAS

array of size Npath x d with the solution beta for each lambda on each row.

Type numpy.ndarray

SIGMAS

array of size Npath with the solution sigma for each lambda when the formulation of the problem is R2 or R4.

Type numpy.ndarray

LAMBDAS

array of size Npath with the lambdas (real lambdas, not divided by lambda_max) for which the solution is computed.

Type numpy.ndarray

logscale

whether or not the path should be plotted with a logscale.

Type bool

method

name of the numerical method that has been used. It can be 'Path-Alg', 'P-PDS' , 'PF-PDS' or 'DR'.

Type str

save
if it is a str, then it gives the name of the file where the graphics has been/will be saved (after using print(solution)).

Type bool or str

formulation
object containing the info about the formulation of the minimization problem we solve.

Type *Formulation*

time
running time of this action.

Type float

class classo.solver.**solution_ALO**(*matrices*, *param*, *formulation*, *numerical_method*, *label*)
Class that contains characteristics of the lasso-path computed, which also contains representation method that plot the graphic of this lasso-path.

BETAS
array of size Npath x d with the solution beta for each lambda on each row.

Type numpy.ndarray

SIGMAS
array of size Npath with the solution sigma for each lambda when the formulation of the problem is R2 or R4.

Type numpy.ndarray

LAMBDAS
array of size Npath with the lambdas (real lambdas, not divided by lambda_max) for which the solution is computed.

Type numpy.ndarray

logscale
whether or not the path should be plotted with a logscale.

Type bool

method
name of the numerical method that has been used. It can be 'Path-Alg', 'P-PDS' , 'PF-PDS' or 'DR'.

Type str

save1, save2
if a string is given, the corresponding graph will be saved with the given name of the file. save1 is for the path plot ; save2 for ALO plot ; and save3 for refit beta-solution.

Type bool or string

formulation
object containing the info about the formulation of the minimization problem we solve.

Type *Formulation*

time
running time of this action.

Type float

class classo.solver.**solution_CV**(*matrices*, *param*, *formulation*, *numerical_method*, *label*)

Class that contains characteristics of the cross validation computed, which also contains a representation method that plot the selected parameters and the solution of the not-sparse problem on the selected variables set.

xGraph
array of size Nlam of the lambdas / lambda_max.

Type numpy.ndarray

yGraph
array of size Nlam of the average validation residual (over the K subsets).

Type numpy.ndarray

standard_error
array of size Nlam of the standard error of the validation residual (over the K subsets).

Type numpy.ndarray

logscale
whether or not the path should be plotted with a logscale.

Type bool

index_min
index on xGraph of the selected lambda without 1-standard-error method.

Type int

index_1SE
index on xGraph of the selected lambda with 1-standard-error method.

Type int

lambda_min
selected lambda without 1-standard-error method.

Type float

lambda_oneSE
selected lambda with 1-standard-error method.

Type float

beta
solution beta of classo at lambda_oneSE/lambda_min depending on *CVparameters.oneSE*.

Type numpy.ndarray

sigma
solution sigma of classo at lambda_oneSE when formulation is 'R2' or 'R4'.

Type float

selected_param
boolean arrays of size d with True when the variable is selected.

Type numpy.ndarray

refit
solution beta after solving unsparse problem over the set of selected variables.

Type numpy.ndarray

save1, save2
if a string is given, the corresponding graph will be saved with the given name of the file. save1 is for CV curve ; and save2 for refit beta-solution.

Type bool or string

formulation
object containing the info about the formulation of the minimization problem we solve.

Type *Formulation*

time
running time of this action.

Type float

solution_CV.**graphic**(*se_max=None*, *save=None*, *logscale=True*, *errorevery=5*)
Method to plot the graphic showing mean squared error over along lambda path once cross validation is computed.

Parameters

- **se_max** (*float*) – float thanks to which the graphic will not show the lambdas from which MSE(lambda)> min(MSE) + se_max * Standard_error(lambda_min). this parameter is useful to plot a graph that zooms in the interesting part. Default value : None
- **logScale** (*bool*) – input that tells to plot the mean square error as a function of lambda, or log10(lambda) Default value : True
- **errorevery** (*int*) – parameter input of matplotlib.pyplot.errorbar that gives the frequency of the error bars appearence. Default value : 5
- **save** (*string*) – path to the file where the figure should be saved. If None, then the figure will not be saved. Default value : None

class classo.solver.**solution_StabSel**(*matrices*, *param*, *formulation*, *numerical_method*, *label*)
Class that contains characteristics of the stability selection computed, which also contains a representation method that plot the selected parameters, the solution of the not-sparse problem on the selected variables set, and the stability plot.

distribution
d array of stability ratios.

Type array

lambdas_path
for 'first' method : Nlam array of the lambdas used. Other cases : 'not used'.

Type array or string

distribution_path
for 'first' method : Nlam x d array with stability ratios as a function of lambda.

Type array or string

threshold
threshold for StabSel, ie for a variable i, stability ratio that is needed to get selected.

Type float

save1, save2
if a string is given, the corresponding graph will be saved with the given name of the file. save1 is for stability plot ; and save2 for refit beta-solution.

Type bool or string

selected_param
boolean arrays of size d with True when the variable is selected.

Type numpy.ndarray

to_label
boolean arrays of size d with True when the name of the variable should be seen on the graph.

Type numpy.ndarray

refit
solution beta after solving unsparse problem over the set of selected variables.

Type numpy.ndarray

formulation
object containing the info about the formulation of the minimization problem we solve.

Type *Formulation*

time
running time of this action.

Type float

class classo.solver.**solution_LAMfixed**(*matrices*, *param*, *formulation*, *numerical_method*, *label*)

Class that contains characteristics of the lasso computed which also contains a representation method that plot this solution.

lambdamax
lambda maximum for which the solution is non-null.

Type float

rescaled_lam
if True, the problem had been computed for lambda*lambdamax (so lambda should be between 0 and 1).

Type bool

lamb
lambda for which the problem is solved.

Type float

beta
solution beta of classo.

Type numpy.ndarray

sigma
solution sigma of classo when formulation is 'R2' or 'R4'.

Type float

selected_param
boolean arrays of size d with True when the variable is selected (which is the case when the i-th component solution of the classo is non-null).

Type numpy.ndarray

refit
solution beta after solving unsparse problem over the set of selected variables.

Type numpy.ndarray

formulation
object containing the info about the formulation of the minimization problem we solve.

Type *Formulation*

time
running time of this action.

Type float

CHAPTER

FIVE

MISCELLANEOUS FUNCTIONS

Functions

random_data(n, d, d_nonzero, k, sigma[, ...])	Generation of random matrices as data such that y = X.sol + sigma.
clr(array[, coef])	Centered-Log-Ratio transformation
theoretical_lam(n, d)	Theoretical lambda as a function of the dimensions of the problem

CHAPTER

SIX

MORE DETAILS

classo.misc_functions.**random_data**(*n*, *d*, *d_nonzero*, *k*, *sigma*, *zerosum=False*, *seed=False*, *classification=False*, *exp=False*, *A=None*, *lb_beta=3*, *ub_beta=10*, *intercept=None*)

Generation of random matrices as data such that y = X.sol + sigma. noise

The data X is generated as a normal matrix. The vector sol is generated randomly with a random support of size d_nonzero, and componants are projected random intergers between -10 and 10 on the kernel of C restricted to the support. The vector y is then generated with X.dot(sol)+ sigma*noise, with noise a normal vector.

Parameters

- **n** (*int*) – Number of samples, dimension of y.
- **d** (*int*) – Number of variables, dimension of sol.
- **d_nonzero** (*int*) – Number of non null componant of sol.
- **k** (*int*) – Number of constraints, number of rows of C.
- **sigma** (*float*) – size of the noise.
- **zerosum** (*bool, optional*) – If True, then C is the all-one matrix with 1 row, independently of k.
- **seed** (*bool or int, optional*) – Seed for random values, for an equal seed, the result will be the same. If set to False: pseudo-random vectors
- **classification** (*bool, optional*) – if True, then it returns sign(y) instead of y.
- **A** (*numpy.ndarray*) – matrix corresponding to a taxa tree, if it is given, then the problem should be y = X.A.g + eps , C.A.g = 0.

Returns tuple of three ndarray that corresponds to the data : (X,C,y). ndarray : array corresponding to sol which is the real solution of the problem y = Xbeta + noise s.t. beta sparse and Cbeta = 0.

Return type tuple

classo.misc_functions.**clr**(*array*, *coef=0.5*)

Centered-Log-Ratio transformation

Set all non positive entry to a constant coef. Then compute the log of each component. Then substract the mean of each column.

Parameters

- **array** (*ndarray*) – matrix nxd
- **coef** (*float, optional*) – Value to replace the zero values

Returns clr transformed matrix nxd

Return type ndarray

classo.misc_functions.**theoretical_lam**(*n*, *d*)

Theoretical lambda as a function of the dimensions of the problem

This function returns (with $\phi = erf$) :

$4/\sqrt{n}\phi^{-1}(1-2x)$ such that $x = 4/d(\phi^{-1}(1-2x)4 + \phi^{-1}(1-2x)^2)$

Which is the same (thanks to formula : $norm^{-1}(1-t) = \sqrt{2}\phi^{-1}(1-2t)$) as :

$\sqrt{2/n} * norm^{-1}(1-k/p)$ such that $k = norm^{-1}(1-k/p)^4 + 2norm^{-1}(1-k/p)^2$

Parameters

- **n** (*int*) – number of sample
- **d** (*int*) – number of variables

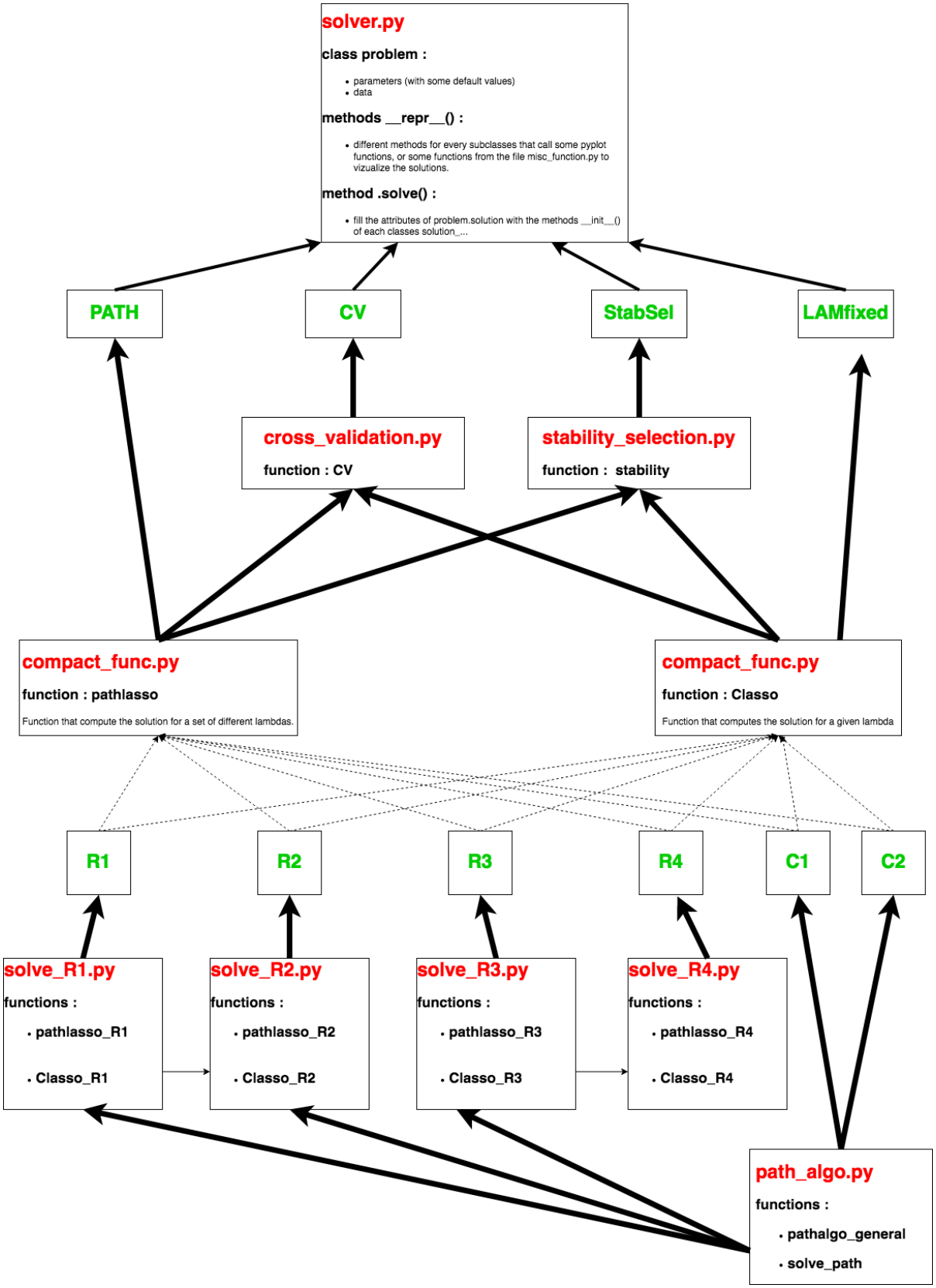
Returns theoretical lambda

Return type float

CHAPTER

SEVEN

STRUCTURE OF THE CODE


solver.py
class problem :
parameters (with some default values)
data
methods __repr__() :
different methods for every subclasses that call some pyplot functions, or some functions from the file misc_function.py to vizualize the solutions.
method .solve() :
fill the attributes of problem.solution with the methods __init__() of each classes solution_...
PATH
CV
StabSel
LAMfixed
cross_validation.py
function : CV
stability_selection.py
function : stability
compact_func.py
function : pathlasso
Function that compute the solution for a set of different lambdas.
compact_func.py
function : Classo
Function that computes the solution for a given lambda
R1
R2
R3
R4
C1
C2
solve_R1.py
functions :
pathlasso_R1
Classo_R1
solve_R2.py
functions :
pathlasso_R2
Classo_R2
solve_R3.py
functions :
pathlasso_R3
Classo_R3
solve_R4.py
functions :
pathlasso_R4
Classo_R4
path_algo.py
functions :
pathalgo_general
solve_path

CHAPTER

EIGHT

LICENSE

CHAPTER

NINE

CONTRIBUTING TO C-LASSO

`c-lasso` is a package that always can be improved. Any feedback can help a lot to fix some bug and to add possible new functionality.

One can contribute either by reporting an error, requesting a new feature or adding a new feature.

9.1 Reporting errors

Any errors or general problems can be reported on GitHub's Issue tracker

The quickest way resolve a problem is to go through the following steps:

- Have I tested this on the latest GitHub (`master`) version? To see which version you use, you can run on python :

```
>>> import classo
>>> classo.__version__
```

- Have I provided a sample code block which reproduces the error? Have I tested the code block?

While more information can help, the most important step is to report the problem, and any missing information can be provided over the course of the discussion.

9.2 Feature requests

We recommend opening an issue on issue on GitHub to discuss potential changes.

When preparing a feature request, consider providing the following information:

- What problem is this feature trying to solve?
- Is it solvable using Python intrinsics? How is it currently handled in similar modules?
- Can you provide an example code block demonstrating the feature?
- Does this feature require any new dependencies ?

9.3 Adding a feature

One can also contribute with a new feature or with fixing a bug.

Feature should be sent as pull requests via GitHub, specifically to the `master` branch, which acts as the main development branch.

Fixes and features are very welcome to `c-lasso`, and are greatly encouraged.

If you are concerned that a project may not be suitable or may conflict with ongoing work, then feel free to submit a feature request.

When preparing a pull request, one should make sure that the code changes:

- Pass existing tests, this can be done by running within the root directory:

```
$ pip install --upgrade pytest
$ pytest
```

- Includes a test case. See the files in `c-lasso/tests` for examples
- Includes some example of use cases. See the files in `c-lasso/examples` for examples
- Depends on standard library. Any features requiring an external dependency should only be enabled when the dependenc is available.
- Be properly documented. c-lasso's documentation (including docstring in code) uses ReStructuredText format, see Sphinx documentation to learn more about editing them. The code follows the NumPy docstring standard. To ensure that documentation is rendered correctly, the best bet is to follow the existing examples for function docstrings. If you want to test the documentation locally, you will need to run the following command lines within the `c-lasso/docs` directory :

```
$ pip install --upgrade sphinx
$ make html
```

9.4 Seeking for support ?

If the above ways of interacting with `c-lasso` does not fit your request, you may contact directly one of the autors.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

M

N

O

P

Q

R

S

T

W

X

Y