linvpy Documentation

Release 0.0.1

Guillaume Beaud, Marta Martinez-Camara

June 07, 2016

Contents

LinvPy is a Python package designed for solving linear inverse problems of the form $y = Ax + n$, where y is a vector of measured values, A a known matrix, x an unknown input vector and n is noise. The goal is to find x, or at least the best possible estimation; if the matrix A is invertible, the solution is easy to find by multiplying by the inverse, if not, we need to use regression techniques such as least squares method to find x.

The first motivation for this project is that Marta Martinez-Camara, PhD student in Communications Systems at EPFL (Switzerland) designed some new algorithms for solving linear inverse problems. LinvPy is a Python implementation of these algorithms, which may not be available anywhere else than here. LinvPy also contains several other known and available techniques such as least squares regression, regularization functions, or M-estimators.

CHAPTER 1

Get it

LinvPy is available from PyPi. If you have pip already installed, simply run :

```
$ sudo pip install linvpy
```

If you don't have pip installed, run :

```
$ sudo easy_install pip
$ sudo pip install linvpy
```

To upgrade linvpy to the latest version :

```
$ sudo pip install --upgrade linvpy
```

CHAPTER 2

Documentation

linvpy.**least_squares**(*matrix_a*, *vector_y*)

This function computes the estimate $\hat{x}$ given by the least squares method $\hat{\mathbf{x}} = \arg\min_x \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2$. This is the simplest algorithm to solve a linear inverse problem of the form $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{n}$, where $\mathbf{y}$ (vector) and $\mathbf{A}$ (matrix) are known and $\mathbf{x}$ (vector)

and $\mathbf{n}$ (vector) are unknown.

Parameters

- **matrix_a** – (np.matrix) matrix $\mathbf{A}$
- **vector_y** – (array) vector $\mathbf{y}$

Return vector_x (array) estimate $\hat{\mathbf{x}}$ given by least squares

Example : compute the least squares solution of a system $\mathbf{y} = \mathbf{A}\mathbf{x}$

```
import numpy as np
import linvpy as lp

A = np.matrix([[1,3],[3,4],[4,5]])
y = [-6,1,-2]

# Returns x_hat, the least squares solution of y = Ax
lp.least_squares(A,y)

# [ 3.86666667 -3.18666667]
```

linvpy.**tikhonov_regularization**(*matrix_a*, *vector_y*, *lambda_parameter=0*)

The standard approach to solve the problem $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{n}$ explained above is to use the ordinary least squares method. However if your matrix $\mathbf{A}$ is a fat matrix (it has more columns than rows) or it has a large condition number, then you should use a regularization to your problem in order to get a meaningful estimation of $\mathbf{x}$.

The Tikhonov regularization is a tradeoff between the least squares solution and the minimization of the L2-norm of the output x (L2-norm = sum of squared values of the vector x), $\hat{\mathbf{x}} = \arg\min_x \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2 + \lambda\|\mathbf{x}\|_2^2$

The parameter lambda tells how close to the least squares solution the output $\mathbf{x}$ will be; a large lambda will make $\mathbf{x}$ close to $\|\mathbf{x}\|_2^2 = 0$, while a small lambda will approach the least squares solution (Running the function with lambda=0 will behave like the ordinary least_squares() method).

The Tikhonov solution has an analytic solution and it is given by $\hat{\mathbf{x}} = (\mathbf{A^T}\mathbf{A} + \lambda^2\mathbf{I})^{-1}\mathbf{A}^T\mathbf{y}$, where $\mathbf{I}$ is the identity matrix.

Raises a ValueError if lambda < 0.

Parameters

- **matrix_a** – (np.matrix) matrix A in $\mathbf{y} = \mathbf{Ax} + \mathbf{n}$
- **vector_y** – (array) vector y in $\mathbf{y} = \mathbf{Ax} + \mathbf{n}$
- **lambda** – (int) lambda non-negative parameter to regulate the trade off.

Return vector_x (array) Tikhonov estimate $\hat{\mathbf{x}}$

Raises **ValueError** – raises an exception if lambda_parameter < 0

Example : compute the solution of a system $\mathbf{y} = \mathbf{Ax}$ (knowing $\mathbf{y}, \mathbf{A}$) which is a trade off between the least squares solution and the minimization of x's L2-norm. The greater lambda, the smaller the norm of the given solution. We take a matrix $\mathbf{A}$ which is ill-conditionned.

```
import numpy as np
import linvpy as lp

A = np.matrix([[7142.80730214, 6050.32000196],
               [6734.4239248, 5703.48709251],
               [4663.22591408, 3949.23319264]])

y = [0.83175086, 0.60012918, 0.89405644]

# Returns x_hat, the tradeoff solution of y = Ax
print lp.tikhonov_regularization(A, y, 50)

# [8.25871731e-05   4.39467106e-05]
```

linvpy.**rho_huber**(*input*, *clipping=1.345*)

The regular huber loss function; the "rho" version.

$$\rho(x) = \begin{cases} \frac{1}{2}x^2 & \text{if |x| <=}clipping, \\ clipping(|x| - \frac{1}{2}clipping) & \text{otherwise.} \end{cases}$$

This function is quadratic for small inputs, and linear for large inputs.

Parameters

- **input** – (float) x
- **clipping** – (optional)(float) clipping parameter. Default value is optimal for normalized distributions.

Return float $\rho(x)$

Example : run huber loss on a vector

```
import linvpy as lp

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

loss = [lp.rho_huber(e, 4) for e in x]

# [0.5, 2.0, 4.5, 8.0, 12, 16, 20, 24, 28]
```

linvpy.**psi_huber**(*input*, *clipping=1.345*)

Derivative of the Huber loss function; the "psi" version. Used in the weight function of the M-estimator.

$$\psi(x) = \begin{cases} x & \text{if |x| <=}clipping, \\ clipping \cdot sign(x) & \text{otherwise.} \end{cases}$$

Parameters

- **input** – (float) x
- **clipping** – (optional)(float) clipping parameter. Default value is optimal for normalized distributions.

Return float $\psi(x)$

Example : run psi_huber derivative on a vector

```
import linvpy as lp

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

derivative = [lp.psi_huber(e, 4) for e in x]

# [1, 2, 3, 4, 4, 4, 4, 4, 4]
```

linvpy.**rho_bisquare**(*input*, *clipping=4.685*)

The regular bisquare loss (or Tukey's loss), "rho" version.

$$\rho(x) = \begin{cases} (c^2/6)(1-(1-(x/c)^2)^3) & \text{if } |x| \leq 0, \\ c^2/6 & \text{if } |x| > 0. \end{cases}$$

Parameters

- **input** – (float) x
- **clipping** – (optional)(float) clipping parameter. Default value is optimal for normalized distributions.

Returns (float) result $\rho(x)$ of bisquare function

Example : run bisquare loss on a vector

```
import linvpy as lp

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

result = [lp.rho_bisquare(e, 4) for e in x]

# [0.4694104166666663, 1.5416666666666665, 2.443359375, 2.6666666666666665, 2.6666666666666665,
```

linvpy.**psi_bisquare**(*input*, *clipping=4.685*)

The derivative of bisquare loss (or Tukey's loss), "psi" version.

$$\psi(x) = \begin{cases} x((1-(x/c)^2)^2) & \text{if } |x| \leq 0, \\ 0 & \text{if } |x| > 0. \end{cases}$$

Parameters

- **input** – (float) x
- **clipping** – (optional)(float) clipping parameter. Default value is optimal for normalized distributions.

Returns (float) $\psi(x)$

Example : run psi_bisquare on a vector

```
import linvpy as lp

x = [1,2,3,4,5,6,7,8,9]
```

```
    result = [lp.psi_bisquare(e, 4) for e in x]

    # [0.87890625, 1.125, 0.57421875, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

linvpy.**rho_cauchy**(*input*, *clipping=2.3849*)

Cauchy loss function; the "rho" version.

$\rho(x) = (c^2/2)log(1 + (x/c)^2)$

Parameters

- **input** – (float) x
- **clipping** – (optional)(float) clipping parameter. Default value is optimal for normalized distributions.

Return float $\rho(x)$

Example : run Cauchy loss on a vector

```
    import linvpy as lp

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

    result = [lp.rho_cauchy(e, 4) for e in x]

    # [0.4849969745314787, 1.7851484105136781, 3.5702968210273562, 5.545177444479562, 7.527866755716
```

linvpy.**psi_cauchy**(*input*, *clipping=2.3849*)

Derivative of Cauchy loss function; the "psi" version.

$\psi(x) = \frac{x}{1+(x/c)^2}$

Parameters

- **input** – (float) x
- **clipping** – (optional)(float) clipping parameter. Default value is optimal for normalized distributions.

Return float result of the Cauchy's derivative function

Example : run psi_cauchy on a vector

```
    import linvpy as lp

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

    result = [lp.psi_cauchy(e, 4) for e in x]

    # [0.9411764705882353, 1.6, 1.92, 2.0, 1.951219512195122, 1.8461538461538463, 1.7230769230769232
```

linvpy.**rho_optimal**(*input*, *clipping=3.27*)

The so-called optimal loss function is given by $\rho(x) = \begin{cases} 1.38(x/c)^2 & \text{if } |x/c| \\ 0.55 - 2.69(x/c)^2 + 10.76(x/c)^4 - 11.66(x/c)^6 + 4.04(x/c)^8 & \text{if } 2/3 \\ 1 & \text{if } |x/c| \end{cases}$

Parameters

- **input** – (float) x
- **clipping** – (optional)(float) clipping parameter. Default value is optimal for normalized distributions.

Return float $\rho(x)$

linvpy.**psi_optimal**(*input*, *clipping=3.27*)

The derivative of the optimal 'rho' function is given by $\psi(x) = \begin{cases} 2 \cdot 1.38x/c^2 & \text{if } |x/c| \leq 2/3, \\ 2 \cdot 2.69x/c^2 + 4 \cdot 10.76x^3/c^4 - 6 \cdot 11.66x^5/c^6 + 8 \cdot 4.04x^7/c^8 & \text{if } 2/3 < |x/c| \leq 1, \\ 0 & \text{if } |x/c| > 1. \end{cases}$

Parameters

- **input** – (float) x
- **clipping** – (optional)(float) clipping parameter. Default value is optimal for normalized distributions.

Return float $\psi(x)$

linvpy.**weights**(*input*, *loss_function*, *clipping=None*, *nmeasurements=None*)

Returns an array of : $\begin{cases} \frac{loss\_function(x_i)}{x_i} & \text{if } x_i \neq 0, \\ 0 & \text{otherwise.} \end{cases}$

Weights function designed to be used with loss functions like rho_huber, psi_huber, rho_cauchy... Note that the loss_function passed in argument must support two inputs.

Parameters

- **input** – (array or float) vector or float to be processed, x_i's
- **loss_function** – (loss_function) f(x) in f(x)/x.
- **clipping** – (optional) clipping parameter of the huber loss function.

Return array or float element-wise result of f(x)/x if x!=0, 0 otherwise

Example : run the weight function with the psi_huber with default clipping or with another function like rho_cauchy and another clipping.

```
import linvpy as lp

x = [1,2,3,4,5]

# psi_huber, default clipping
lp.weights(x, lp.psi_huber)

# [1.0, 0.67249999999999999, 0.44833333333333331, 0.33624999999999999, 0.26900000000000002]

# rho_cauchy, clipping=2.5
lp.weights(x, lp.rho_cauchy, 2.5)

# [0.46381251599460444, 0.7729628778689174, 0.9291646242761568, 0.9920004256749526, 1.0058986952
```

linvpy.**irls**(*matrix_a*, *vector_y*, *loss_function*, *clipping=None*, *scale=None*, *lamb=0*, *initial_x=None*, *regularization=<function tikhonov_regularization>*, *kind=None*, *b=0.5*, *tolerance=1e-05*, *max_iterations=100*)

The method of iteratively reweighted least squares (IRLS) minimizes iteratively the function:

$\boldsymbol{x}^{(t+1)} =$

$_{\boldsymbol{x}} arg\,min\, \boldsymbol{W}(\boldsymbol{x}^{(t)}) ||\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}||_2^2.$

The IRLS is used, among other things, to compute the M-estimate and the tau-estimate.

Parameters

- **matrix_a** – (np.matrix) matrix A in y - Ax
- **vector_y** – (array) vector y in y - Ax
- **loss_function** – the loss function to be used in the M estimator
- **clipping** – clipping parameter for the loss function

Return array vector x solution of IRLS

linvpy.**basictau**(*a*, *y*, *loss_function*, *clipping*, *ninitialx*, *maxiter=100*, *nbest=1*, *initialx=None*, *b=0.5*, *regularization=<function tikhonov_regularization>*, *lamb=0*)

This routine minimizes the objective function associated with the tau-estimator. For more information on the tau estimator see http://arxiv.org/abs/1606.00812

This function is hard to minimize because it is non-convex. This means that it has several local minima. Depending on the initial x that we use for our minimization, we will end up in a different local minimum.

In this algorithm we take the 'brute force' approach: let's try many different initial solutions, and let's pick the minimum with smallest value. The output of basictau are the best nbest minima (we will need them later)

Parameters

- **a** – matrix A in y - Ax
- **y** – vector y in y - Ax
- **loss_function** – type of the rho function we are using
- **clipping** – clipping parameters. In this case we need two, because the rho function for the tau is composed two rho functions.
- **ninitialx** – how many different solutions do we want to use to find the global minimum (this function is not convex!) if ninitialx=0, means the user introduced a predefined initial solution
- **maxiter** – maximum number of iteration for the irls algorithm
- **nbest** – we return the best nbest solutions. This will be necessary for the fast algorithm
- **initialx** – the user can define here the initial x he wants
- **b** – this is a parameter to estimate the scale

Return xhat contains the best nmin estimations of x

Return mintauscale value of the objective function when x = xhat

linvpy.**fasttau**(*y*, *a*, *loss_function*, *clipping*, *ninitialx*, *regularization=<function tikhonov_regularization>*, *nmin=5*, *initialiter=5*, *lamb=0*)

linvpy.**array_loss**(*values*, *loss_function*, *clipping=None*)

CHAPTER 3

Contribute

If you want to contribute to this project, feel free to fork our GitHub main repository repository : https://github.com/GuillaumeBeaud/linvpy. Then, submit a 'pull request'. Please follow this workflow, step by step:

1. Fork the project repository: click on the 'Fork' button near the top of the page. This creates a copy of the repository in your GitHub account.
2. Clone this copy of the repository in your local disk.
3. Create a branch to develop the new feature :

```
$ git checkout -b new_feature
```

   Work in this branch, never in the master branch.

4. To upload your changes to the repository :

```
$ git add modified_files
$ git commit -m "what did you implement in this commit"
$ git push origin new_feature
```

When your are done, go to the webpage of the main repository, and click 'Pull request' to send your changes for review.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

l

A

B

F

I

L

P

R

T

W