multilstsq Documentation

Release 1.0.0

Laurent Fasnacht

May 08, 2018

Contents

Least squares fitting is a underlying method for numerous applications, the most common one being linear regression. It consists in finding the parameters vector $\beta°$ which minimizes $||\epsilon||_2$ in the equation $y = X\beta + \epsilon$, where X is the design matrix, y the observation vector, and ϵ the error vector.

Since it is a fundamental algorithm, a number of Python 3 implementations exist, with different feature sets and performance, such as: `numpy.linalg.lstsq`, `scipy.stats.linregress`, `sklearn.linear_model.LinearRegression` and `statsmodel.OLS`.

However, the current available libraries are not designed to work on a large quantity of simultaneous problems, for example solving a least square problem for each pixel of an image. Iterating over a large number of small problems is inefficient. Moreover, when doing linear regression, it is often tedious to build the design matrix `X`.

The goal of `multilstsq` is to work on arrays of problems, with good performance, low memory requirements, reliability and flexibility. It also provides a way to automate the construction of the relevant structures (mostly the design matrix), using a model given as a string. It however does not strive to be a complete statistical library such as what would be provided by `statsmodel` or the language `R`.

To reach these goals, `multilstsq` uses the following techniques:

- It is possible to compute $\beta° = (XX)^{-1}Xy$ incrementally, due to the linearity of `XX` and `Xy`, by providing data in chunks.
- Inverting `XX` is done by explicit formulas when the dimension is small. This has the advantage of being vector operations which can be applied simultaneously on all problems.
- Masked data are handled as lines of zeros in the design matrix and the observation, which in fact have no effect. This allows adding different amount of data in different subproblems.
- For regression, an expression evaluator is implemented, which converts the input model from the user (for example `b0+b1*x0`) into the complex expression needed to build the design matrix from the vector `X` provided by the user. In that example, it is: `np.concatenate([np.ones(o_dim)[(..., np.newaxis)], ((X)[..., :, 0])[(..., np.newaxis)]])`. This expression evaluator also may be useful for other purposes in other libraries.

CHAPTER 1

Example use

```
import numpy as np
from multilstsq import MultiRegression

x1 = np.array([1.47, 1.50, 1.52, 1.55, 1.57, 1.60, 1.63, 1.65, 1.68, 1.70, 1.73, 1.75,
↪ 1.78, 1.80, 1.83])
y1 = np.array([52.21, 53.12, 54.48, 55.84, 57.20, 58.57, 59.93, 61.29, 63.11, 64.47,␣
↪66.28, 68.10, 69.92, 72.19, 74.46])

x2 = np.arange(10)
y2 = np.arange(10)

X = np.ma.masked_all((2, max(len(x1), len(x2)), 1))
y = np.ma.masked_all((2, max(len(x1), len(x2)), 1))

X[0, :len(x1), 0] = x1
X[1, :len(x2), 0] = x2

y[0, :len(y1), 0] = y1
y[1, :len(y2), 0] = y2

mr = MultiRegression((2,), 'b0 + b1*x0 + b2*(x0**2)')
mr.add_data(X,y)
mr.switch_to_variance()
mr.add_data(X,y)

print(mr.beta)
#Identify parameter names in the parameter vector
print(mr.beta_names)

#Get the covariance matrix for the first problem
print(mr.variance[0])

#Get the expression to predict for the first problem
expr = mr.get_expr_for_idx((0,))
```

(continues on next page)

(continued from previous page)

```
#Evaluate at x=1.79
print(expr(1.79))
```

The nice thing about this module is that it is possible to change the model by only changing the line instanciating the MultiRegression object. For example, for a quadratic regression:

```
mr = MultiRegression((2,), 'b0 + b1*x0 + b2*(x0**2)')
```

CHAPTER 2

Documentation contents

2.1 Installation

`multilstsq` can be installed either using `pip`, or from source.

Since `multilstsq` requires `Python 3`, you should ensure that you're using the correct `pip` executable. On some distributions, the executable is named `pip3`.

It is possible to check the `Python` version of `pip` using:

```
pip -V
```

Similarly, you may need to use `py.test-3` instead of `py.test`.

2.1.1 Installing using `pip`

To install using `pip`, simply run:

```
pip install multilstsq
```

2.1.2 Manual installation from source

To install manually, run the following:

```
git clone https://github.com/UniNE-CHYN/multilstsq
cd multilstsq
pip install .
```

To contribute, use `pip install -e` instead of `pip install`. This sets a link to the source folder in the python installation, instead of copying the files.

It is advisable to run the tests to ensure that everything is working. This can be done by running the following command in the `multilstsq` directory:

```
py.test
```

2.2 Tutorial

This page contains some information about how to start if you want to use `multilstsq` in your project.

Three different use cases are described in this documentation:

1. You want to solve `y = X`β in the least squares sense, `y` and `X` are already available.
2. You want to solve `y = f(x, `β`)` in the least squares sense (β is the unknown), where `f` is linear in β, but not necessarily in `x`. This is the linear regression case. Using `Multiregression`, it is possible to test multiple different `f`, without changing anything else, so it's great for evaluating multiple models.
3. You only want to use the expression evaluator to add flexibility to some algorithm of your creation. This case is described in its own section: *Using ExprEvaluator*.

For the first two cases, we use the following terminology: `X` is the *design matrix*, `y` the *observation vector*, and β the *parameter vector*.

Since `multilstsq` is usually used to solve multiple problems simultaneously, we define as *problem dimension* the size of the n-dimensional grid of problems.

The number of problem dimension doesn't have much influence on the way the computation are done, but it makes sense to use dimensions that corresponds to the problem. For instance, if a characteristics is required for each element of a 20×20 grid, it makes sense to define the problem dimension as `(20, 20)`, even though it would be possible to map the problem to a `(400, )` or a `(10, 10, 4)` problem dimension.

2.2.1 Solving a least squares problem

First, lets see how to solve a least squares problem.

To keep the example small, let consider `problem_dimension = (2, 3)`, and the model to be `y = `β_0` + `β_1`x`.

We have two parameters, and the design matrix consists in rows `[1 x]`.

Let's see how to do such a regression using first only the class *`multilstsq.MultiLstSq`*. This class requires us to build the explicit design matrix, and does a textbook least square solving on it. First, we need to create the object:

```
from multilstsq import MultiLstSq
import numpy as np

# First argument is the problem dimensions
# Second argument is the number of parameters
mls = MultiLstSq((2, 3), 2)
```

Now, we need to construct the data to add. We can add one or multiple "observations" at the time. For a single problem, the matrix `X` would be of dimensions `(n_observations, 2)` (since we have two parameters), while the observation vector should be a column vector of shape: `(n_observations, 1)`. However, since we are solving multiple simultaneous least squares problems, we need to add the data for all the problems simultaneously. Therefore in our case the matrix `X` is of dimensions `(2, 3, n_observations, 2)` (we added the problem dimensions) and the observation vector `(2, 3, n_observations, 1)`.

```
# 4 observations
X = np.ma.masked_all((2, 3, 4, 2))
y = np.ma.masked_all((2, 3, 4, 1))
```

(continues on next page)

(continued from previous page)

```
# This is for β₀
X[:, :, :, 0] = 1

# Values of x, as coefficients of β₁
# The first two dims are the problem index
# The third is the observation index
X[0, 0, :, 1] = [1, 2, 3, 4]
X[0, 1, :, 1] = [1, 2, 1, 2]
X[0, 2, :, 1] = [1, 1, 1, 1]
X[1, 0, :, 1] = [-5, -6, 13, 43]
X[1, 1, :3, 1] = [-1, 0, 1]  # only 3 observations
X[1, 2, :2, 1] = [4, 8]  # only 2 observations

# Observations
y[0, 0, :, 0] = [1, 2, 3, 4]
y[0, 1, :, 0] = [1.1, 2, 0.9, 2.1]
y[0, 2, :, 0] = [3, 4, 5, 6]
y[1, 0, :, 0] = [-5.9, -5.2, 11.9, 42.1]
y[1, 1, :3, 0] = [1, 2, 3]  # only 3 observations
y[1, 2, :2, 0] = [4.5, 5]  # only 2 observations

#Add the data to the regression
mls.add_data(X, y)
```

It is possible to run *multilstsq.MultiLstSq.add_data()* multiple times, to "stream" the data to the solver, for example one observation at a time. It may seem complicated but usually it's quite easy to convert the data matrix into the design matrix. We will see later an even simpler method to do so.

Once we added all the data, we can get β:

```
print(mls.beta.shape) # (2, 3, 2, 1)

# Since it is a big matrix, lets get the value for the first problem
print(mls.beta[0, 0]) # We get a column vector [0, 1], as expected

print(mls.beta[0, 2]) # This problem cannot be solved, we get [nan, nan]
```

It may be required to have variance information. We need to switch to variance mode, and then re-add all the data:

```
mls.switch_to_variance()

# We should add exactly the same data as in the previous step.
# It doesn't matter if the number of calls is the same.
mls.add_data(X, y)
```

Once all the data is added, it is possible to get the covariance matrix for each problem. The covariance matrix is of shape `(n_parameters, n_parameters)` for each subproblem.

```
# We can get the covariance matrix
print(mls.variance.shape)  # (2, 3, 2, 2)

print(mls.variance[0, 0])  # all zero, as it is a perfect fit

print(mls.variance[1, 0])  # [[0.299, -0.006], [-0.006, 0.001]]
```

(continues on next page)

(continued from previous page)

```
print(mls.variance[1, 2])  # all masked: no variance can be computed (only two points␣
↪for the line)
print(mls.variance[0, 2])  # all masked: no variance can be computed (not possible to␣
↪do the fit)
```

It is also possible to get some additional information, which enables additional processing (like implementing tests):

```
print(mls.n_observations)
# [[4 4 4]
#  [4 3 2]]

print(mls.n_parameters)
# 2

print(mls.sigma_2)
# [[0.00 0.01 -- ]
#  [0.90 0.00 -- ]]

print(mls.rss)
# [[0.00 0.03 -- ]
#  [1.81 0.00 0.0]]
```

2.2.2 Solving a regression problem

The direct approach shown in the section above works well, but building the design matrix may be impractical when complex model are used, especially if the model changes. In the following example, for readability we will use `()` as the problem size, this means we only do a single regression. Working with a bigger problem size is exacly similar to the previous section.

Let's say we have the following data:

```
import numpy as np
x0 = np.array([-0.44, -0.52, -0.65, 0.89, -1.15, 1.07, -0.1, 1.05, 1.5, 0.23, -0.87,␣
↪1.77, -0.42, 0.43, 1.58, -0.2, -1.69, -1.92, 1.18, -1.18])
x1 = np.array([1.36, -0.69, -0.96, -0.27, 0.34, -0.02, -0.63, -0.66, 0.96, -0.21, 0.
↪01, -0.06, -1.3, 1.05, 1.08, -1.74, -0.87, 0.72, 0.7, 1.67])
y = np.array([-0.68, -1.74, -2.24, 2.65, -3.23, 3.29, -0.45, 2.97, 4.96, 0.71, -2.51,␣
↪5.35, -1.68, 1.81, 5.25, -1.2, -5.35, -5.41, 3.91, -2.79])
```

We suspect that this data follows a multilinear relationship according to the model $y = \beta_0 + \beta_1 x_0 + \beta_2 x_1$. The classical approach would be to create a design matrix with rows $[1, x_0, x_1]$, but we can let the class *multilstsq.MultiRegression* do the work for us:

```
from multilstsq import MultiRegression
mr = MultiRegression((), 'b0 + b1 * x0 + b2 * x1')

#Note that X is a matrix of two columns x_0, x_1
X = np.array([x0, x1]).T
Y = np.array([y]).T

mr.add_data(X,Y)
mr.switch_to_variance()
mr.add_data(X,Y)

mr.rss  # 0.002193175857390197
```

(continues on next page)

(continued from previous page)

```
# etc.
```

Now, let say we suspect that the model is not what we expected, we can compare models easily:

```
models = [
  'b0 + b1 * x0 + b2 * x1',
  'b0 + b1 * x0 + b2 * x1 + b3 * (x1**2)',
  'b0 + b1 * x0 + b2 * x1 + b3 * (x0**2)',
  'b0 + b1 * x0 + b2 * x1 + b3 * (x0**2) + b4 * (x1**2)',
]

for model in models:
  mr = MultiRegression((), model)
  mr.add_data(X,Y)
  mr.switch_to_variance()
  mr.add_data(X,Y)

  print('{:0.05f}'.format(mr.rss), model)

# We obtain the following output:
# 0.00278 b0 + b1 * x0 + b2 * x1
# 0.00270 b0 + b1 * x0 + b2 * x1 + b3 * (x1**2)
# 0.00016 b0 + b1 * x0 + b2 * x1 + b3 * (x0**2)
# 0.00014 b0 + b1 * x0 + b2 * x1 + b3 * (x0**2) + b4 * (x1**2)
```

We can see that the best model is likely to be $y = \beta_0 + \beta_1 x_0 + \beta_2 x_1 + \beta_2 {x_0}^2$. Using this kind of technique it is very simple to make step-wise regression.

The model string can be any valid Python expression, but requires it to be linear in `b`'s. Each variable `b0`, `b1`, `b2`... corresponds to a parameter, while `x0`, `x1`, `x2`... corresponds to columns of the matrix `X`.

2.3 Using `ExprEvaluator`

This page describes how to use *`multilstsq.ExprEvaluator`*, which is a tool to parse and *evaluate* (convert to a Python object) expressions provided as strings.

At first, an expression is parsed:

```
from multilstsq import ExprEvaluator
e = ExprEvaluator('a*(2+c*b)')
```

This expression has 3 *variables*: `a`, `b` and `c`. We can get this information using:

```
e.variables # returns {'a', 'b', 'c'}
```

As long as these variables are not substituted, it is not possible to evaluate the expression. We can substitute these variables and evaluate the results by doing:

```
e(a=3, b=5, c=1) # returns 21, as expected.
```

However, this is not the typical use case. The goal of *`multilstsq.ExprEvaluator`* is to enable **partial** substitution, for example substituting `b` and `c`, but not `a`:

```
e2 = e.substitute(constants={'b':5, 'c':1})
e2(a=3) # Will also return 21, as previously
```

We can see that `e2` has two *constants* (defined variables) and one variable. We can also get the expression as a string for debugging:

```
e2.variables # returns {'a'}
e2.constants # returns {'b', 'c'}
str(e2)      # returns '(a) * ((2) + ((c) * (b)))'
             # The parentheses are added to ensure correct resolution order
```

By looking at the expression, we can observe that second operand of the multiplication is fully known, and therefore that it could be *reduced*:

```
e3 = e2.reduce()
str(e3)      # returns '(a) * (__ExprEvaluator_0)'
e3.variables # returns {'a'}
e3.constants # returns {'__ExprEvaluator_0'}
```

We can observe that the expression has indeed be reduced. In the present case, it would have made sense to reduce it to `(a) * 7`, but instead a (reserved) constant `__ExprEvaluator_0` has been created. This makes sense the result of the parenthesis could have been a complex object, like a numpy array.

If the expression is evaluated multiple times, it is usually a large performance improvement to substitute everything possible before evaluating it.

Note that any python object can be substituted:

```
import numpy as np
e4 = e3.substitute(constants = {'a': np.array([1, 2, 3])})

e4.eval() # returns array([ 7, 14, 21])
e4()      # returns the same thing
e4.reduce().eval() # still the same thing
```

Complex expression can also be used in the evaluation string. Here's another example:

```
import numpy as np
e = ExprEvaluator('np.linalg.inv(A).dot(y)')
v1 = np.array([[1, 4], [2, -1]])
v2 = np.array([9, 0])
e(A=v1,y=v2) # returns array([1., 2.])
```

By default, *`multilstsq.ExprEvaluator`* uses the modules defined at the calling stack frame. This makes `np` work in the expression in the example above. This behavious can be disabled if needed:

```
import numpy as np
e = ExprEvaluator('np.linalg.inv(A).dot(y)', enable_caller_modules=False)
v1 = np.array([[1, 4], [2, -1]])
v2 = np.array([9, 0])
e(A=v1,y=v2) # Will fail, np is not defined

e(A=v1,y=v2, np=np) # returns array([1., 2.])
```

Additionally, it is also possible to substitute a variable in the expression by something else:

```
import numpy as np
e = ExprEvaluator('np.linalg.inv(A).dot(y)')
e2 = e.substitute(expressions={'A':'A.T'})

str(e2) # '(((((np).linalg).inv)(A)).dot)(y)'

v1 = np.array([[1, 4], [2, -1]])
v2 = np.array([9, 0])
e(A=v1,y=v2) # returns array([1., 4.])
```

This functionnality is used extensively by the *multilstsq.MultiRegression* class.

Warning: Using *multilstsq.ExprEvaluator* to evaluate expressions from untrusted sources may lead to security vulnerabilities.

2.4 API Documentation

This is the API documentation of the **public** API of multilstsq. These are the most commonly used functions when using multilstsq.

2.4.1 Least squares

class multilstsq.**MultiLstSq**(*problem_dimensions*, *n_parameters*, *internal_dtype=<class 'float'>*)

Multiple least squares problems.

$y = X\beta + \epsilon$

This is a class which can exist in three modes:

- `MODE_REGRESSION`: add data to the least square matrices to obtain the mean
- `MODE_VARIANCE`: compute the variance
- `MODE_READONLY`: the object is frozen.

The mode can only be switched "forward" (it is not possible to move from `MODE_VARIANCE` to `MODE_REGRESSION` for example.)

__init__(*problem_dimensions*, *n_parameters*, *internal_dtype=<class 'float'>*)

Create a MultiLstSq object, which is originally in `MODE_REGRESSION`.

Parameters

- **problem_dimensions** – Tuple, size of the problem array, it can be `()` (0-dimensional), for a single least square problem, or for example `(800, 600)` for 800x600 times the regression problem
- **n_parameters** – Number of parameters of the least squares problem.
- **internal_dtype** – `numpy.dtype` data type of the matrices

__weakref__

list of weak references to the object (if defined)

add_data(*X*, *y*, *w=None*)

Add data to the object (depending on the mode, for either mean or variance computation)

beta
The linear coefficients that minimize the least squares criterion.

evaluate_at(*X*)
Evaluate Xβ

n_observations
Number of observations n.

n_parameters
Number of parameters

rss
Residual sum of squares

sigma_2
Scaling parameter for the covariance matrix.

switch_to_read_only()
Switch to read-only mode.

switch_to_variance()
Switch to variance computation mode.

variance
Returns the variance/covariance matrix.

2.4.2 Regression

class multilstsq.**MultiRegression**(*problem_dimensions*, *model_str*, *internal_dtype=<class 'float'>*)

__init__(*problem_dimensions*, *model_str*, *internal_dtype=<class 'float'>*)
Create a MultiLstSq object, which is originally in `MODE_REGRESSION`.

Parameters

- **problem_dimensions** – Tuple, size of the problem array, it can be `()` (0-dimensional), for a single least square problem, or for example `(800, 600)` for 800x600 times the regression problem
- **n_parameters** – Number of parameters of the least squares problem.
- **internal_dtype** – `numpy.dtype` data type of the matrices

add_data(*X*, *y*, *w=None*)
Add data to the object (depending on the mode, for either mean or variance computation)

2.4.3 Expression evaluator

class multilstsq.**ExprEvaluator**(*expr*, *constants=None*, *enable_caller_modules=True*)

__call__(**args*, ***kwargs*)
Substitute the arguments in the expression, and then evaluate it and return the resulting Python object.

The positional arguments are `zip()`'ed with the `variable_list` of *multilstsq.ExprEvaluator.enable_call()*, which the named arguments are directly substituted.

Returns The Python object of the evaluated expression.

Raises

- **RuntimeError** – if positional arguments are used, but *multilstsq.ExprEvaluator.enable_call()* was not called.
- **ValueError** – if there are undefined variables.

__init__(*expr*, *constants=None*, *enable_caller_modules=True*)
Initialize an ExprEvaluator object

Parameters

- **expr** – Expression string, should be a valid Python 3 expression
- **constant** – Dictionary of constants `{'constantname': value}`.
- **enable_caller_modules** – Boolean, if `True` modules of the first frame in the stack outside of multilstsq will be included in the context of the expression

__repr__()
Return repr(self).

__str__()

Returns a string corresponding to the expression

__weakref__
list of weak references to the object (if defined)

constants

Returns Set of constants names (values which have a defined substitution) in the current expression

enable_call(*variable_list*)
Enables calling the expression, as a simplification for calling *multilstsq.ExprEvaluator.substitute()* followed by *multilstsq.ExprEvaluator.eval()*.

Parameters **variable_list** – List of variables names which correspond to the arguments which will be used in *multilstsq.ExprEvaluator.__call__()*

eval()

Returns The python object which results of the expression

Raises **ValueError** – if at least one variable is not defined

reduce()

Returns A copy of the current expression, where all known part of the syntax tree are simplified.

For example, reducing `a*(b+c)` with known `b` and `c`, will result in `a*__ExprEvaluator_0`, where `__ExprEvaluator_0` is defined as a constant.

substitute(*expressions=None*, *constants=None*)
Substitute expressions or constants in the current expression.

Parameters

- **expressions** – dictionary of expression to substitute. The keys can be of `str`, `ast.AST` or *multilstsq.ExprEvaluator*.
- **constants** – dictionary of constants to substitute `{'constantname': value}`

Returns New *multilstsq.ExprEvaluator* object with the substitution done.

variables

Returns Set of variables names (values which don't have a defined substitution) in the current expression

2.5 Contributing guide

`multilstsq` is a free software, and all contributions are welcome, whether they are bug reports, source code, or documentation.

2.5.1 Reporting bugs

To report bugs, open an issue in the issue tracker.

Ideally, a bug report should contain at least the following information:

- a minimum code example to trigger the bug
- the expected result
- the result obtained.

2.5.2 Quick guide to contributing code or documentation

To contribute, you'll need Sphinx to build the documentation, and pytest to run the tests.

If for some reason you are not able to run the following steps, simply open an issue with your proposed change.

1. Fork the multilstsq on GitHub.
2. Clone your fork to your local machine:

```
git clone https://github.com/<your username>/multilstsq.git
cd multilstsq
pip install -e .
```

3. Create a branch for your changes:

```
git checkout -b <branch-name>
```

4. Make your changes.
5. If you're writing code, you should write some tests, ensure that all the tests pass and that the code coverage is good. This can be done using:

```
py.test --cov=multilstsq --pep8
```

6. You should also check that the documentation compiles and that the result look good. The documentation can be seen by opening a browser in *doc/html*. You can (re)build it using the following command line (make sure that there is no warnings):

```
sphinx-build doc/source doc/html
```

7. Commit your changes and push to your fork on GitHub:

```
git add .
git commit -m "<description-of-changes>"
git push origin <name-for-changes>
```

8. Submit a pull request.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index