symfit Documentation

Release 0.2

tBuLi

December 06, 2015

Contents

Contents:

CHAPTER 1

Introduction

Existing fitting modules are not very pythonic in their API and can be difficult for humans to use. This project aims to marry the power of `scipy.optimize` with the readability of `SymPy` to create a highly readable and easy to use fitting package which works for projects of any scale.

`symfit` makes it extremely easy to provide guesses for your parameters and to bound them to a certain range:

```
a = Parameter(1.0, min=0.0, max=5.0)
```

To define models to fit to:

```
x = Variable()
A = Parameter()
sig = Parameter(1.0, min=0.0, max=5.0)
x0 = Parameter(1.0, min=0.0)
# Gaussian distrubution
model = exp(-(x - x0)**2/(2 * sig**2))
```

And finally, to execute the fit:

```
fit = Fit(model, xdata, ydata)
fit_result = fit.execute()
```

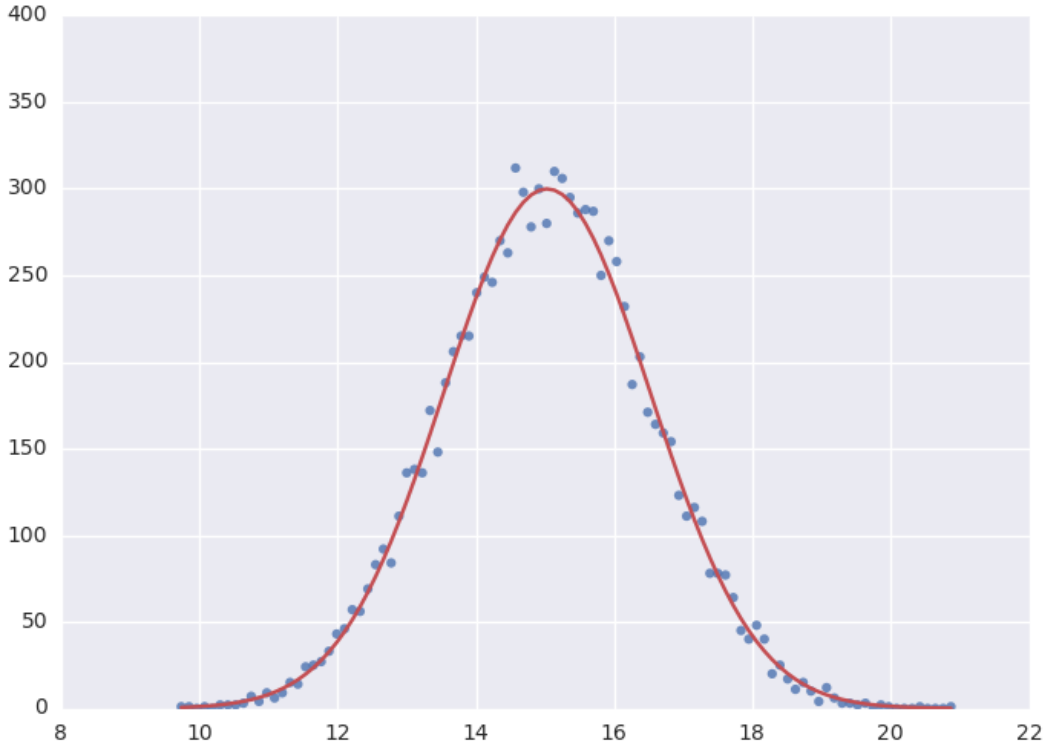
And to evaluate the model using the best fit parameters:

```
y = model(x=xdata, **fit_result.params)
```

For the full code to this or other examples, check the example library here: example-library.

1.1 Technical Reasons

On a more technical note, this symbolic approach turns out to have great technical advantages over using scipy directly. In order to fit, the algorithm needs the Jacobian: a matrix containing the derivatives of your model in it's parameters. Because of the symbolic nature of `symfit`, this is determined for you on the fly, saving you the trouble of having to determine the derivatives yourself. Furthermore, having this Jacobian allows good estimation of the errors in your parameters, something `scipy` does not always succeed in.


400
350
300
250
200
150
100
50
0
8
10
12
14
16
18
20
22

CHAPTER 2

Installation

If you are using pip, you can simply run

```
pip install symfit
```

from your terminal. If you are using linux and do not use pip, you can download the source from https://github.com/tBuLi/symfit and install manually.

Are you not on linux and you do not use pip? That's your own mess.

2.1 Dependencies

```
pip install sympy
pip install numpy
pip install scipy
```

CHAPTER 3

Quick Start

If you simply want the most important parts about symfit, you came to the right place.

3.1 Single Variable Problem

```
from symfit.api import Parameter, Variable, exp, Fit

A = Parameter(100, min=0)
b = Parameter()
x = Variable()
model = A * exp(x * b)

xdata = # your 1D xdata. This is a quick start guide, so I'm assuming you know how to get it.
ydata = # 1D ydata

fit = Fit(model, xdata, ydata)
fit_result = fit.execute()

# Plot the fit.
# The model *has* to be called by keyword arguments to prevent any ambiguity
y = model(x=xdata, **fit_result.params)
plt.plot(xdata, y)
plt.show()
```

3.2 symfit.api exposes sympy.api

`symfit.api` exposes the sympy api as well, so mathematical expressions such as `exp`, `sin` and `pi` are importable from `symfit.api` as well. For more, read the sympy docs.

3.3 Initial Guess

For fitting to work as desired you should always give a good initial guess for a parameter. The `Parameter` object can therefore be initiated with the following keywords:

- `value` the initial guess value.
- `min` Minimal value for the parameter.

- `max` Maximal value for the parameter.
- `fixed` Fix the value of the parameter during the fitting to `value`.

In the example above, we might change our `Parameter`'s to the following after looking at a plot of the data:

```
a = Parameter(value=4, min=3, max=6)
```

3.4 Multivariable Problem

Let M be the number of variables in your model, and N the number of data point in xdata. Symfit assumes xdata to be of shape $N \times M$ or even $N_1 \times \ldots N_i \times M$ dimensional, as long as either the first or last axis of the array is of the same length as the number of variables in your model. Currently it is assumed that the function is not vector valued, meaning that for every datapoint in xdata, only a single y value is returned. Vector valued functions are on my ToDo list.

```
from symfit.api import Parameter, Variable, Fit

a = Parameter()
b = Parameter()
x = Variable()
y = Variable()
model = a * x**2 + b * y**2

xdata = # your NxM data.
ydata = # ydata

fit = Fit(model, xdata, ydata)
fit_result = fit.execute()

# Plot the fit.
z = model(x=xdata[:, 0] y=xdata[:, 1], **fit_result.params)
plt.plot(xdata, z)
plt.show()
```

CHAPTER 4

Tutorial

4.1 Simple Example

The example below shows how easy it is to define a model that we could fit to.

```
from symfit.api import Parameter, Variable

a = Parameter()
b = Parameter()
x = Variable()
model = a * x + b
```

Lets fit this model to some generated data.

```
from symfit.api import Fit
import numpy as np

xdata = np.linspace(0, 100, 100) # From 0 to 100 in 100 steps
a_vec = np.random.normal(15.0, scale=2.0, size=(100,))
b_vec = np.random.normal(100.0, scale=2.0, size=(100,))
ydata = a_vec * xdata + b_vec # Point scattered around the line 5 * x + 105

fit = Fit(model, xdata, ydata)
fit_result = fit.execute()
```

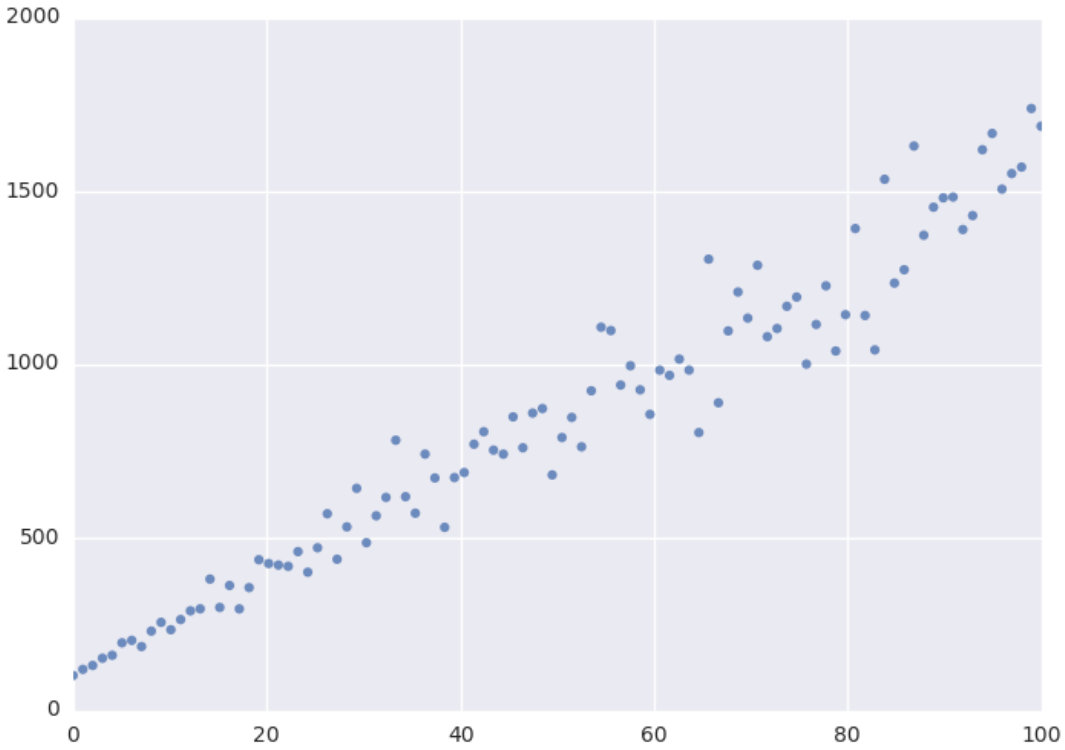
Printing `fit_result` will give a full report on the values for every parameter, including the uncertainty, and quality of the fit.

4.2 Initial Guess

For fitting to work as desired you should always give a good initial guess for a parameter. The `Parameter` object can therefore be initiated with the following keywords:

- `value` the initial guess value.
- `min` Minimal value for the parameter.
- `max` Maximal value for the parameter.
- `fixed` Fix the value of the parameter during the fitting to `value`.

In the example above, we might change our `Parameter`'s to the following after looking at a plot of the data:


2000
1500
1000
500
0
0
20
40
60
80
100

```
a = Parameter(value=4, min=3, max=6)
```

4.3 Accessing the Results

A call to `Fit.execute()` returns a `FitResults` instance. This object holds all information about the fit. The fitting process does not modify the `Parameter` objects. In this example, `a.value` will still be `4.0` and not the value we obtain after fitting. To get the value of fit paramaters we can do:

```
>>> print(fit_result.params.a)
>>> 14.66946...
>>> print(fit_result.params.a_stdev)
>>> 0.3367571...
>>> print(fit_result.params.b)
>>> 104.6558...
>>> print(fit_result.params.b_stdev)
>>> 19.49172...
>>> print(fit_result.r_squared)
>>> 0.950890866472
```

For more FitResults, see the API docs. (Under construction.)

4.4 Evaluating the Model

With these parameters, we could now evaluate the model with these parameters so we can make a plot of it. In order to do this, we simply call the model with these values:

```
import matplotlib.pyplot as plt

y = model(x=xdata, a=fit_result.params.a, b=fit_result.params.b)
plt.plot(xdata, y)
plt.show()
```

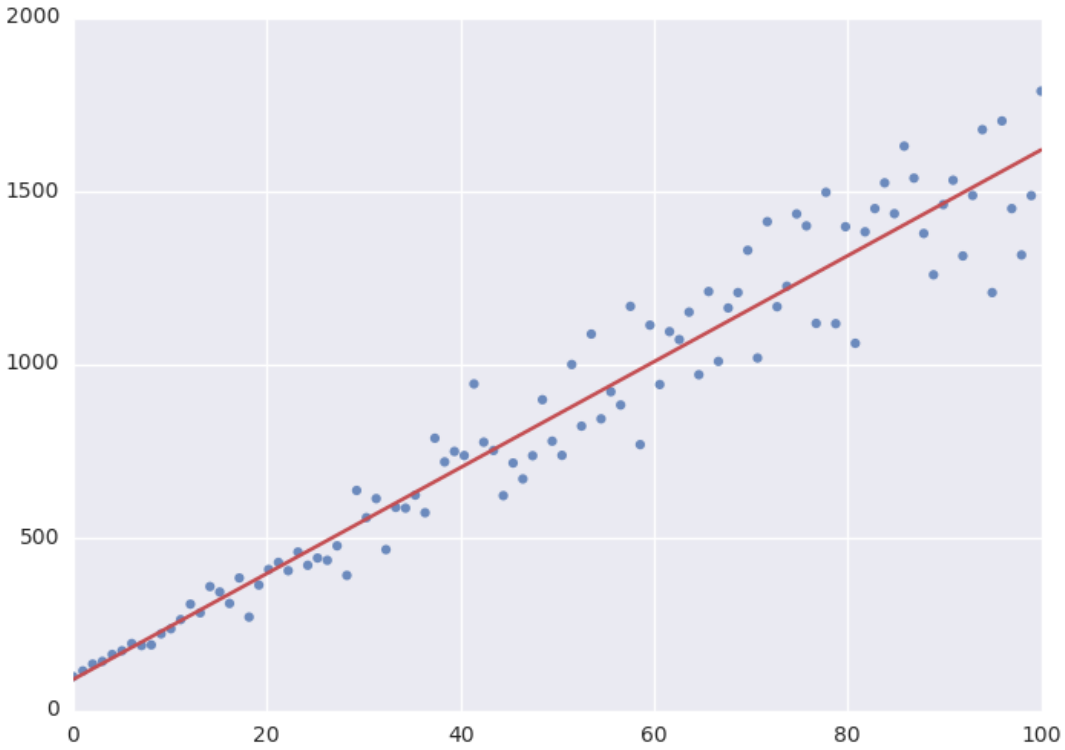
The model *has* to be called by keyword arguments to prevent any ambiguity. So the following does not work:

```
y = model(xdata, fit_result.params.a, fit_result.params.b)
```

To make life easier, there is a nice shorthand notation to immediately use a fit result:

```
y = model(x=xdata, **fit_result.params)
```

This unpacks the .params object as a dict. For more info view ParameterDict.


2000
1500
1000
500
0
0
20
40
60
80
100

CHAPTER 5

Fitting Types

5.1 Fit (LeastSquares)

The default fitting object does least-squares fitting:

```
from symfit.api import Parameter, Variable, Fit
import numpy as np

# Define a model to fit to.
a = Parameter()
b = Parameter()
x = Variable()
model = a * x + b

# Generate some data
xdata = np.linspace(0, 100, 100) # From 0 to 100 in 100 steps
a_vec = np.random.normal(15.0, scale=2.0, size=(100,))
b_vec = np.random.normal(100.0, scale=2.0, size=(100,))
ydata = a_vec * xdata + b_vec # Point scattered around the line 5 * x + 105

fit = Fit(model, xdata, ydata)
fit_result = fit.execute()
```

`Fit` currently simply wraps `LeastSquares`. This might be changed in the future to it determining which fit would work best for the current data and then just trying the best option.

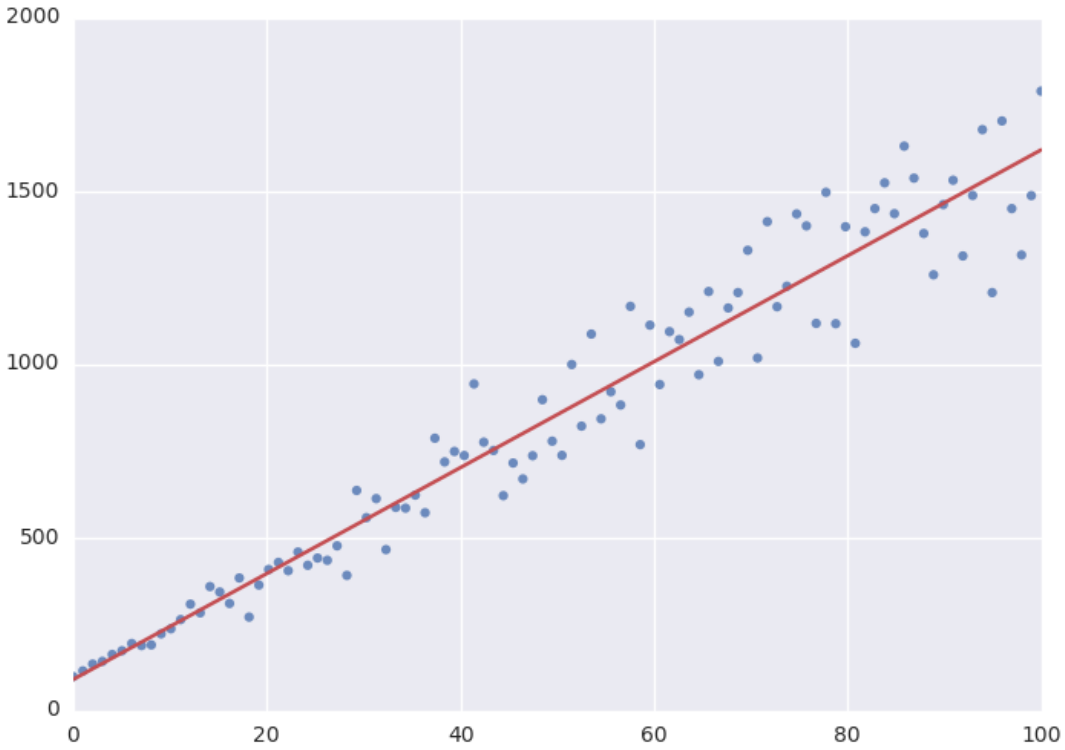
5.2 Likelihood

Given a dataset and a model, what values should the model's parameters have to make the observed data most likely? This is the principle of maximum likelihood and the question the Likelihood object can answer for you.

Example:

```
from symfit.api import Parameter, Variable, Likelihood, exp
import numpy as np

# Define the model for an exponential distribution (numpy style)
beta = Parameter()
x = Variable()
model = (1 / beta) * exp(-x / beta)
```


2000
1500
1000
500
0
0
20
40
60
80
100

```
# Draw 100 samples from an exponential distribution with beta=5.5
data = np.random.exponential(5.5, 100)

# Do the fitting!
fit = Likelihood(model, data)
fit_result = fit.execute()
```

Off-course `fit_result` is a normal `FitResults` object. Because `scipy.optimize.minimize` is used to do the actual work, bounds on parameters, and even constraints are supported. For more information on this subject, check out `symfit`'s `Minimize`.

5.3 Minimize/Maximize

Minimize or Maximize a model subject to bounds and/or constraints. It is a wrapper to `scipy.optimize.minimize`. As an example I present an example from the `scipy` docs.

Suppose we want to maximize the following function:

$$f(x, y) = 2xy + 2x - x^2 - 2y^2$$

Subject to the following constraits:

$$x^3 - y = 0$$

$$y - 1 >= 0$$

In SciPy code the following lines are needed:

```
def func(x, sign=1.0):
    """ Objective function """
    return sign*(2*x[0]*x[1] + 2*x[0] - x[0]**2 - 2*x[1]**2)

def func_deriv(x, sign=1.0):
    """ Derivative of objective function """
    dfdx0 = sign*(-2*x[0] + 2*x[1] + 2)
    dfdx1 = sign*(2*x[0] - 4*x[1])
    return np.array([ dfdx0, dfdx1 ])

cons = ({'type': 'eq',
         'fun' : lambda x: np.array([x[0]**3 - x[1]]),
         'jac' : lambda x: np.array([3.0*(x[0]**2.0), -1.0])},
        {'type': 'ineq',
         'fun' : lambda x: np.array([x[1] - 1]),
         'jac' : lambda x: np.array([0.0, 1.0])})

res = minimize(func, [-1.0,1.0], args=(-1.0,), jac=func_deriv,
               constraints=cons, method='SLSQP', options={'disp': True})
```

Takes a couple of readthroughs to make sense, doesn't it? Let's do the same problem in `symfit`:

```
x = Parameter()
y = Parameter()
model = 2*x*y + 2*x - x**2 -2*y**2
constraints = [
  x**3 - y == 0,
  y - 1 >= 0,
```

```
]

fit = Maximize(model, constraints=constraints)
fit_result = fit.execute()
```

Done! symfit will determine all derivatives automatically, no need for you to think about it.

Warning: You might have noticed that `x` and `y` are `Parameter`'s in the above problem, which may stike you as weird. However, it makes perfect sence because in this problem they are parameters to be optimised, not variables. Furthermore, this way of defining it is consistent with the treatment of `Variable`'s and `Parameter`'s in `symfit`. Be aware of this when using these objects, as the whole process won't work otherwise.

CHAPTER 6

Dependencies and Credits

Always pay credit where credit's due. `symfit` uses the following projects to make it's sexy interface possible:

- leastsqbound-scipy is used to bound parameters to a given domain.
- seaborn was used to make the beautifully styled plots in the example code. All you have to do to sexify your matplotlib plot's is import seaborn, even if you don't use it's special plotting facilities, so I highly recommend it.
- numpy and scipy are off-course used to do efficient data manipulation.
- sympy is used for the manipulation of the symbolic expressions that give this project it's high readability.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search