fitterpp

Release 0.0.2

Joseph L Hellerstein

Dec 18, 2022

CONTENTS:

fitterpp (pronounced “fitter plus plus”) fits a parameterized function to data. Some key features are:

1. Simplicity. Users only provide a parameterized function, data, and descriptions of parameters.
2. Sophistication. Provides for running several fitting algorithm in succession, and starting from multiple initial values.
3. Statistics. Provides information on the runtime and quality of parameter fits.

CHAPTER

ONE

CORE CONCEPTS

Many times we want to fit a *parameterized* function to data. For example, suppose that we have an array of data `y[n]` that we want to fit as a linear function of the variables `x[n]`, where the n-th element of each array. That is, we want to find the slope `a` and the y-intercept `b` such that `a*x[n] + b` is as close as possible to `y[n]`. We define "as close as possible" to mean that the sum of the squared difference between `y[n]` and `a*x[n] + b` is as small as possible.

Fitting is the process of finding parameters `a` and `b` that make the **fitting function** as close as possible to the observational data. Thus, to perform fitting, we must specify:

- the fitting function;
- the parameters of the function that are to be adjusted;
- observational data;

CHAPTER

TWO

FITTING BASICS

This section describes the basic usage of `fitterpp`.

To install the package use:

```
pip install fitterpp
```

To use the package in your code, include the following statement at the top of your python module:

```
import fitterpp as fpp
```

To do fitting, you must first write a parameterized function. For example, consider the following function for a parabola the has two parameters: * where the parabola is centered on the x-axis * a multiplier for how quickly the y-value increases

```
def calcParabola(center=None, mult=None, xvalues=range(20), is_dataframe=True):
    estimates = np.array([mult*(n - center)**2 for n in xvalues])
    if is_dataframe:
        result = pd.DataFrame({"row_key": xvalues, "y": estimates})
        result = result.set_index("row_key")
    else:
        result = np.array([estimates])
        result = np.reshape(result, (len(estimates), 1))
    return result
```

Note that all arguments to `calcParabola` are specified using keywords. `Fitterpp` requires both an array and `DataFrame` output for efficiency reasons and to make the user function self-describing. The keyword argument `is_dataframe` specifies whether to return a `numpy` array or a `DataFrame`. The array should contain only the data values. The `DataFrame` must:

- contain columns names that match some of the names in data provided to `Fitterpp`;
- have an index with values that have a non-null intersection with index values in the data provided to `Fitterpp`.

You will also need to describe the parameters to be fitted. In our example, these are `center` and `mult`. You use lmfit.Parameters to describe these parameters, as shown below.

```
parameters = lmfit.Parameters()
parameters.add("center", value=0, min=0, max=100)
parameters.add("mult", value=0, min=0, max=100)
```

Last, you must provide data that is used to fit the parameters. The data should be a `pandas` `DataFrame` that has some (or all) of the columns present in the output of the function to be fit.

```
print(data_df)
row_key   y
```

(continues on next page)

(continued from previous page)

```
0       203.602263
1       168.826647
2       129.106718
3       106.392522
4        76.568092
5        53.599780
6        32.178451
7        27.475269
8        17.673933
9         5.571118
10        9.088864
11        4.040736
12       10.043712
13       20.858908
14       32.427186
15       53.417786
16       80.242909
17      104.973683
18      132.189584
19      169.439043
```

and outputs a list (or list-like) of floats that are the difference between what the function computed for these parameter values and observational data.

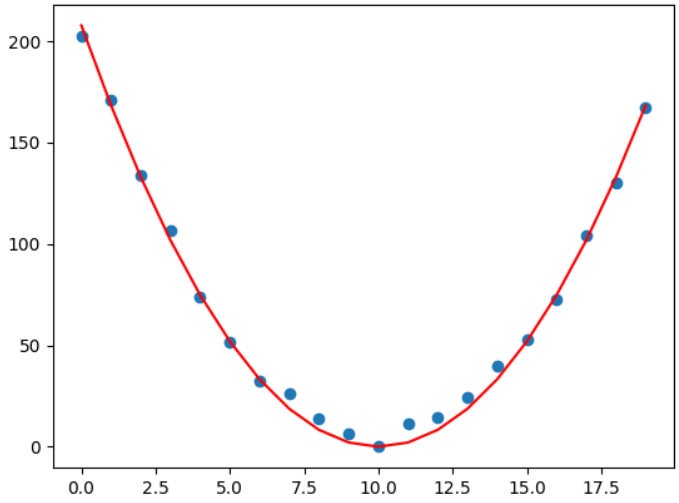
To do a fit, use:

```
fitter = fpp.Fitterpp(calcParabola, parameters, data_df , methods=methods)
fitter.execute()
```

To see the resulting fits:a

```
>print(fitter.final_params.valuesdict())
{'center': 9.991226336877833, 'mult': 2.072849009501976}
```

The figure below displays the parabola (red line plot) for the above fitted parameter values along with the fitting data (blue scatter plot).


200
150
100
50
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

CHAPTER

THREE

ADANCED FITTING

This section describes how to use advanced features of `fitterpp`. The section assumes that you have read the basic tutorial. Specifically, you should be familiar with the following the `calcParabola` function we used as an example of fitting

```
def calcParabola(center=None, mult=None, xvalues=XVALUES):
    estimates = np.array([mult*(n - center)**2 + for n in xvalues])
    return pd.DataFrame({"x": xvalues, "y": estimates})
```

and the following script that fits the observational data `data_df` to the parameters `center` and `mult` in `calcParabola`.

```
# Import the required libraries
import lmfit
import fitterpp as fpp
# Construct the parameter objects
parameters = lmfit.Parameters()
parameters.add("center", value=0, min=0, max=100)
parameters.add("mult", value=0, min=0, max=100)
# Run the fitting algorithm
fitter = fpp.Fitterpp(calcParabola, parameters, data_df)
fitter.execute()
# Display the fittted values
print(fitter.final_params.valuesdict())
```

A first consideration in more advanced fitting is to have more control over the way in which `fitterpp` searches for parameter values. This is accompished by making use of an optional keyword parameter in the constructor, `fpp.Fitterpp`. You can specify any algorithm that is used by `lmfit.minmize`. To simplify common usage, `fitterpp` provides global constants for the "leastsq" and "differential_evolution" algorithms.

```
# Spsecify the methods used for fitting
methods = fpp.Fitterpp.mkFitterMethod(
      method_names=fpp.METHOD_DIFFERENTIAL_EVOLUTION,
      method_kwargs={fpp.MAX_NFEV: 1000})
fitter = fpp.Fitterpp(calcParabola, parameters, data_df,
      methods=methods)
```

A second consideration in more advanced fitting is the tradeoff between the followin: * quality of the fit and * runtime of the fitting codes.

Runtime of the fitting codes is typically measured in seconds. We quantify the quality of the fit by the sum of squares of the residuals or **RSSQ**. If there is a perfect match between the observational data and the estimates produced by fitted parameters, then RSSQ is 0. Larger values of RSSQ indicate a fit that with lower quality. Many times a fitting problem involves trade-offs between quality and runtime.

You can get a basic understanding of the quality of the fit from the fitter report. Building on the example in the basic tutorial, As before, we are fitting the parameters of `calcParabola`. To use this feature, first perform a fit and then use

```
print(fitter.report())
```

This produces the output below:

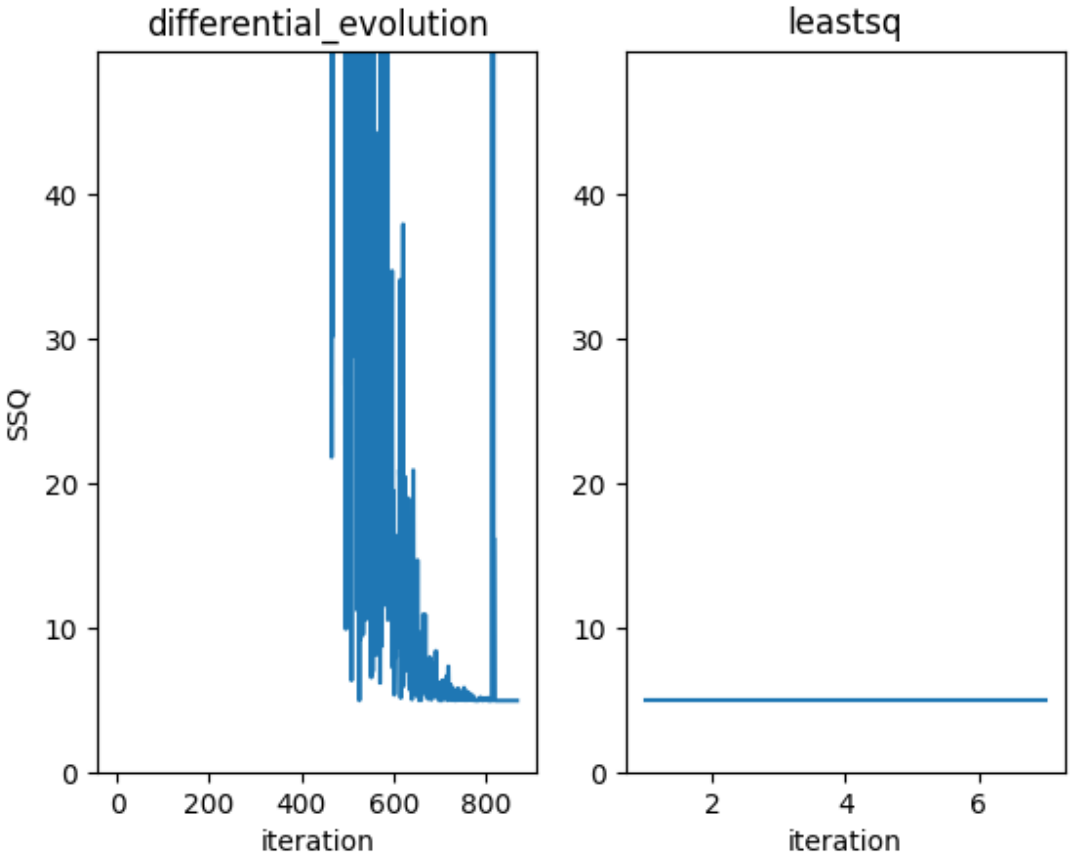
```
[[Variables]]
    mult:  2.0102880244080366
    center:  10.000857728276536
[[Fit Statistics]]
    # fitting method   = differential_evolution
    # function evals   = 921
    # data points      = 20
    # variables        = 2
    chi-square         = 2.54104032
    reduced chi-square = 0.14116891
    Akaike info crit   = -37.2631740
    Bayesian info crit = -35.2717095
```

Of most interest here is the number of function evalutions–921. This provides some insight into the extent of the search for fitting values.

The quality plot indicates how RSSQ changes across iterations. Sometimes, a large fraction of iterations do not result in reductions in RSSQ. To generate the quality plot use:

```
fitter.plotQuality()
```

which produces the plot

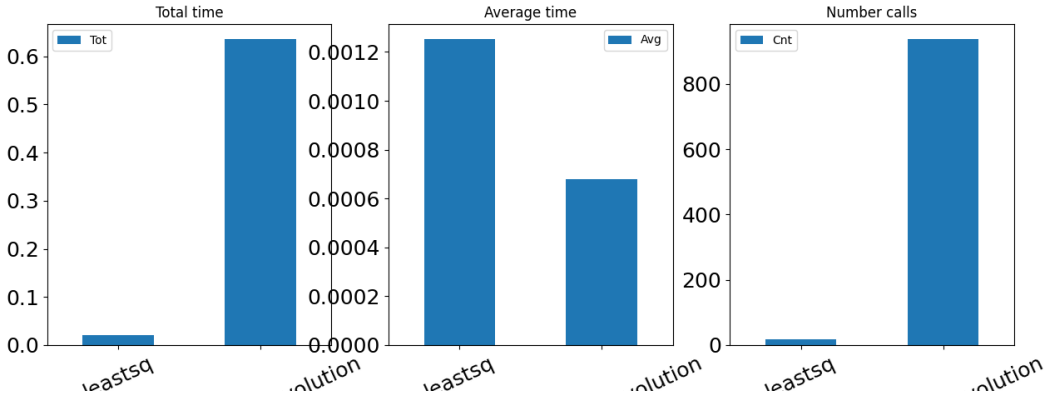


Note that the y-axis of the plot is scaled to show RSSQ values within ten times the minimal RSSQ. From this plot, we observe that (a) differential evolution rquires about 800 iterations before RSSQ is reduced substantially; and (b) gradient descent ("leastsq") provides little reduction in RSSQ.

Finally, the performance plot provides insight into the causes of long runtimes. To generate the performance plot use:

```
fitter.plotPerformance()
```

which produces the plot below.



From this we conclude that the time required to fit the parameters is large due to the large number of iterations of

differential evolution.

CHAPTER

FOUR

METHODS

class fitterpp.**Fitterpp**(*user_function*, *initial_params*, *data_df*, *method_names=None*, *max_fev=1000*, *num_latincube=None*, *latincube_idx=None*, *logger=None*, *is_collect=False*)

Implements an interface to parameter fitting methods that provides additional capabilities and bug fixes. The class also handles an oddity with lmfit that the final parameters returned may not be the best.

If latincube_idx is not None, then use a precomputed latin cube position.

4.1 Usage

fitter = fitterpp(calcResiduals, params, [cn.METHOD_LEASTSQ]) fitter.fit()

static Fitterpp.**mkFitterppMethod**(*method_names=None*, *method_kwargs=None*, *max_fev=1000*)

Constructs an FitterppMethod Parameters ———- method_names: list-str/str method_kwargs: list-dict/dict

4.2 Returns

list-FitterppMethod

Fitterpp.**report**()

Reports the result of an optimization.

4.3 Returns

str

Fitterpp.**plotQuality**(*is_plot=True*)

Plots the quality results

4.4 Parameters

is_plot: bool (plot the output)

4.5 Returns

dict
key: method name value: list-float (residual sum of squares)

`Fitterpp.plotPerformance(is_plot=True)`
Plots the statistics for running the objective function.

4.6 Parameters

is_plot: bool (plot the output)

4.7 Returns

pd.DataFrame

Columns
tot: total_times cnt: counts avg: averages

index: method

INDEX