« Go back

Three examples of nonlinear least-squares fitting in Python with SciPy

by Elias Hernandis • 05 April 2020

Least-squares fitting is a well-known statistical technique to estimate parameters in mathematical models. It concerns solving the optimisation problem of finding the minimum of the function

$$F(\theta) = \sum_{i=1}^{N} \rho(f_i(\theta)^2),$$

where

- $\theta = (\theta_1, \ldots, \theta_r)$ is a collection of parameters which we want to estimate,
- N is the number of available data points,
- ρ is a loss function to reduce the influence of outliers, and
- $f_i(\theta)$ is the i-th component of the vector of residuals.

Given a model function $m(t; \theta)$ and some data points $D = \{(t_i, d_i) \mid i = 1, \ldots, N\}$, one normally defines the vector of residuals as the difference between the model prediction and the data, that is:

$$f_i(\theta) = m(t_i; \theta) - d_i.$$

If the model is linear, i.e. $\theta = (m, n)$ and $m(t; m, n) = mt + n$, then the previous approach is reduced to the even more well known linear least-squares fitting problem, in the sense that there exists an explicit formula for finding the optimal value for the parameters $\hat{\theta}$. In this report we consider more involved problems where the model may be nonlinear and finding the optimal value $\hat{\theta}$ must be done with iterative algorithms. We shall not go into the theoretical details of the algorithms, but rather explore the implementation of the `least_squares` function available in the `scipy.optimize` module of the SciPy Python package. In particular, we give examples of how to handle multi-dimensional and multi-variate functions so that they adhere to the `least_squares` interface.

First example: a scalar function

The first example we will consider is a simple logistic function

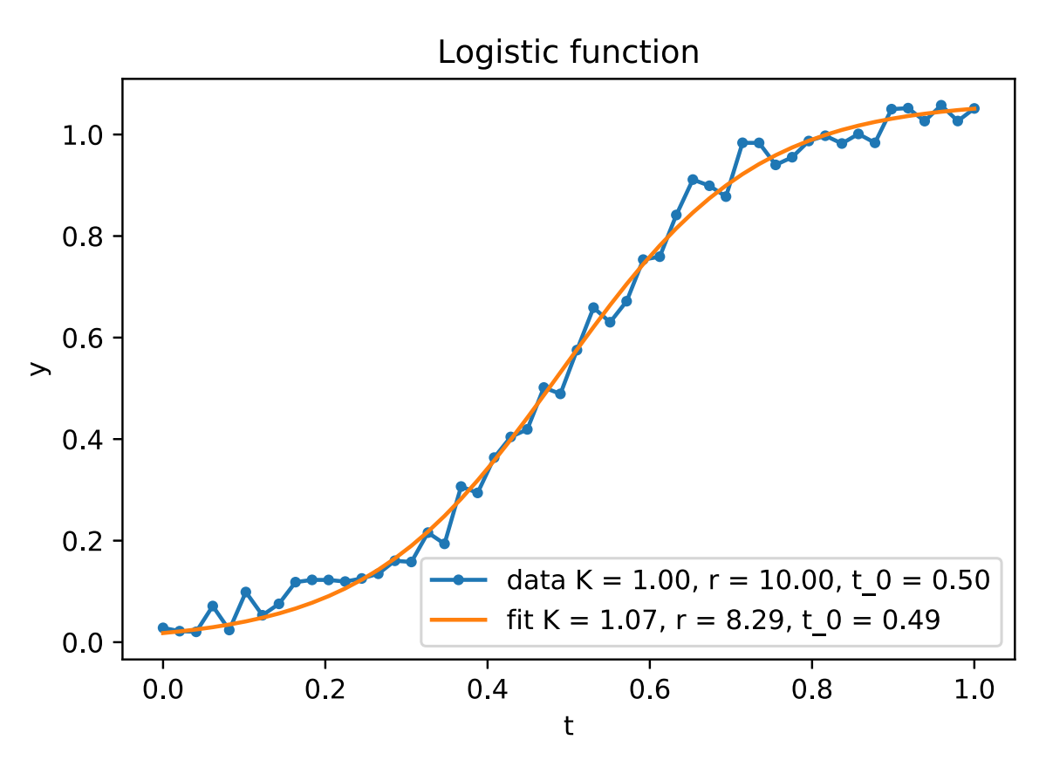
$$y(t) = \frac{K}{1 + e^{-r(t-t_0)}}.$$

The three parameters in the function are:

- K, the supremum of y (think of this as a maximum that is achieved when $t = \infty$),
- r, the logistic growth rate, or sharpness of the curve, and
- t_0 the value at which the midpoint $K/2$ is achieved.

Optimising the parameters $\theta = (K, r, t_0)$ is straightforward in this case, since `least_squares` allows us to input the model without any special changes. To do this we first define the model as the Python function $y(theta, t)$ and then generate some data in `ys` by evaluating the model over a sample `ts` of the independent variable t. When generating the test data, we are passing an array of numbers `ts` directly to the model. We are able to do this because we defined the model `y` using NumPy's `array` object which implements standard element-wise operations between arrays. We add some uniformly distributed noise, make an initial guess for the parameters `theta0` and define the vector of residues `fun`. The vector of residues must be a function of the optimization variables θ so that `least_squares` can evaluate the fitness of its guesses.

```
def y(theta, t):
    return theta[0] / (1 + np.exp(- theta[1] * (t - theta[2])))

ts = np.linspace(0, 1)
K = 1; r = 10; t0 = 0.5; noise = 0.1
ys = y([K, r, t0], ts) + noise * np.random.rand(ts.shape[0])

def fun(theta):
    return y(theta, ts) - ys

theta0 = [1,2,3]
res1 = least_squares(fun, theta0)
```



We see that the estimated parameters are indeed very close to those of the data.

In some cases, we may want to only optimise some parameters while leaving others fixed. This can be done by removing them from `theta` and hard-coding them into the model.

Second example: a parametrised circumference

The motivation behind this example is that sometimes we may have data that is described by two dependent quantities which we hypothesize to be a function of a single independent variable. The simplest example I could think of was that of a parametrised circumference:

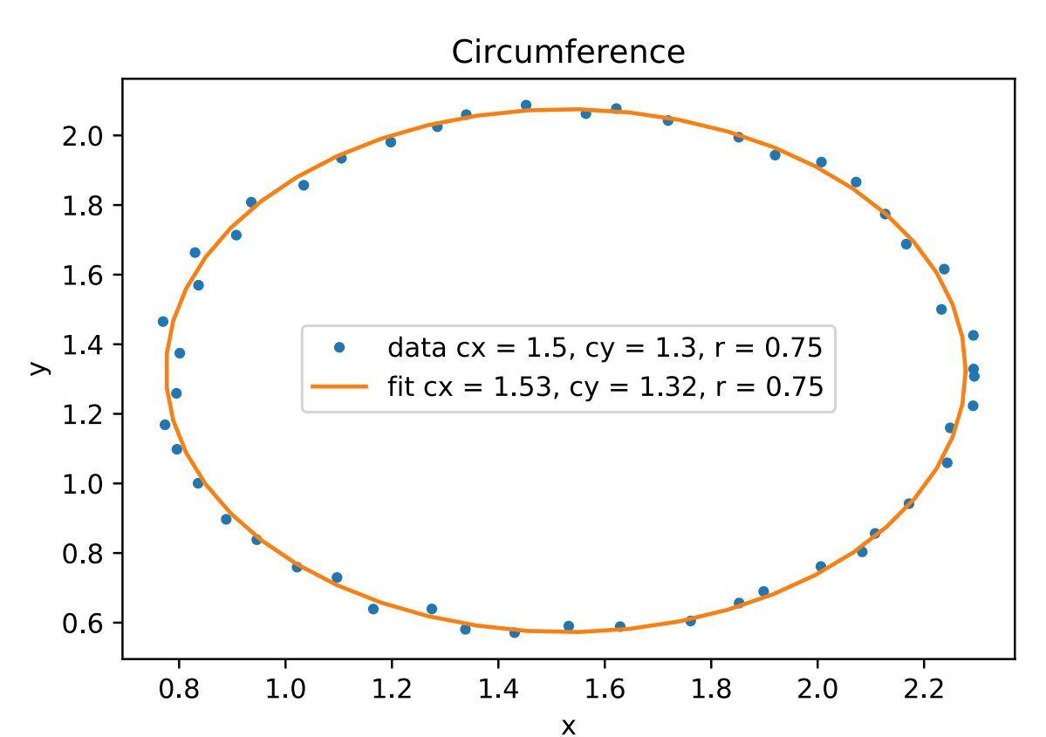
$$s(t) = (c_x + r\cos(t), c_y + r\sin(t)).$$

Here, the parameters are

- the centre $C = (c_x, c_y)$, and
- the radius r.

Like before, we define the model `s` and generate some random data. Only this time, the model outputs two dependent variables $s(t) = (x(t), y(t))$. As expected, this requires us to define the model differently so that it returns two numbers instead of one as well as adding noise to both outputs. When it comes to defining the vector of residuals, we must take care to match the shape expected by `least_squares`. Per the documentation, we must provide a vector of N elements which `least_squares` will square before inputing the result into the loss function ρ. However, we want to compute the square of the distance between the model prediction and the test data. We would normally do this by calculating $(m_x - d_x)^2 + (m_y - d_y)^2$. But if we did this the result would get squared again by `least_squares`. The solution is to return a vector of residues of size $2N$ where components 1 through N correspond to the differences $m_x - d_x$ and components $N+1$ through $2N$ correspond to the differences $m_y - d_y$. This is easily achieved by taking the difference between the prediction and the data as usual and then *flattening* the two arrays into a longer one. We are able to do this because `least_squares` never really sees the raw data and the cost functions presented at the beginning of the report is just there to provided a mathematical background. Also, this approach generalises easily to higher-dimensional model outputs.

```
def s(theta, t):
    x = theta[0] + theta[2] * np.cos(t)
    y = theta[1] + theta[2] * np.sin(t)
    return np.array([x, y])

ts = np.linspace(0, 2 * np.pi)
cx = 1.5; cy = 1.3; r = 0.75; noise = 0.05
ss = s([cx, cy, r], ts)
ss[0] += noise * np.random.rand(ts.shape[0])
ss[1] += noise * np.random.rand(ts.shape[0])

def fun(theta):
    return (s(theta, ts) - ss).flatten()

theta0 = [0,0,0]
res2 = least_squares(fun, theta0)
```



Third example: an elliptic paraboloid

Lastly we focus on an elliptic paraboloid since it can be parametrised as a function of two independent variables with

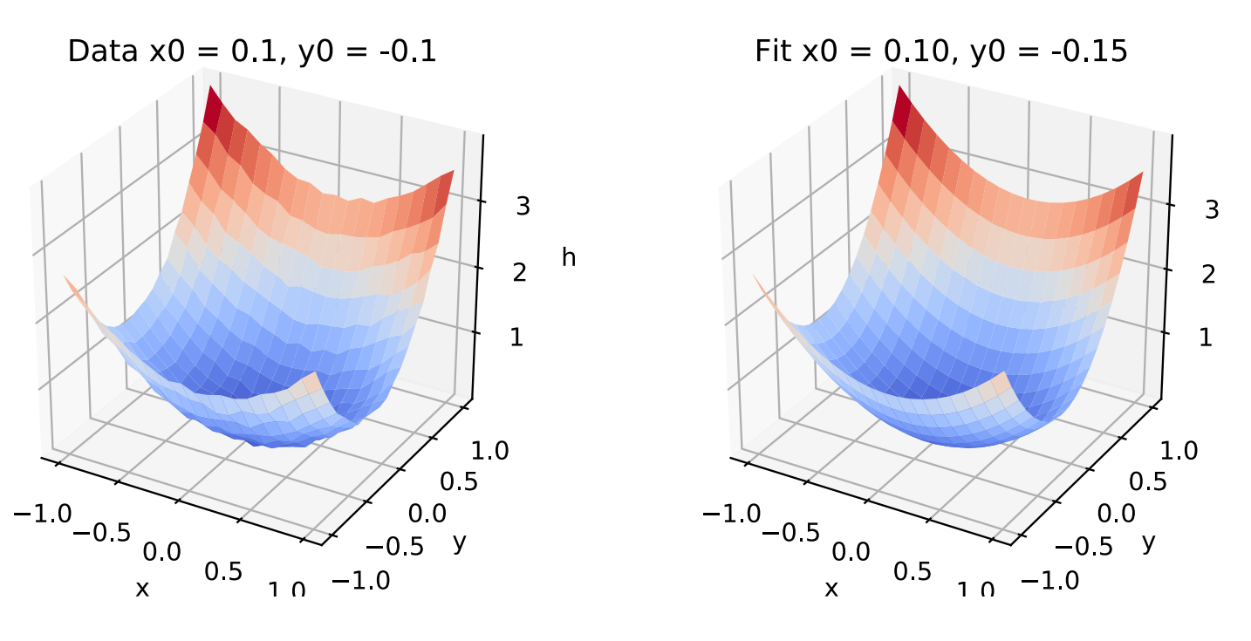
$$h(x, y) = a(x - x_0)^2 + b(y - y_0)^2.$$

Here the parameters are

- a and b, which determine the eccentricity of the ellipses that the level sets of the paraboloid ($a = b = 1$ give circumferences), and
- (x_0, y_0), which determine the projection of the vertical axis of the paraboloid onto the xy-plane.

In this instance we must also be careful with how we sample the domain of the independent variable. We choose to sample the square $[-1, 1] \times [-1, 1]$ with a 20×20 mesh grid, i.e. we evaluate the model at points $(-1 + 0.1k, -1 + 0.1k)$ for $k = 0, \ldots, 20$. This sounds more complicated than it really is, since NumPy provides a method `numpy.meshgrid` that does this for us. As before, we add noise, taking care to do so for each of the $400 = 20 \times 20$ data points we generated. For the model definition, we do not need to do anything special since NumPy also implements binary element-wise operations between the components of a mesh grid.

```
def h(theta, x, y):
    return theta[2] * (x - theta[0])**2 + theta[3] * (y - theta[1])*

xs = np.linspace(-1, 1, 20)
ys = np.linspace(-1, 1, 20)
gridx, gridy = np.meshgrid(xs, ys)
x0 = 0.1; y0 = -0.15; a = 1; b = 2; noise = 0.1
hs = h([x0, y0, a, b], gridx, gridy)
hs += noise * np.random.default_rng().random(hs.shape)

def fun(theta):
    return (h(theta, gridx, gridy) - hs).flatten()

theta0 = [0, 0, 1, 2]
res3 = least_squares(fun, theta0)
```



Goodness of fit and parameter distribution estimation

Once we have parameter estimates for our model, one question we may ask ourselves is how well does the model fit the data. Since we are doing least-square estimation, one of the easiest errors to calculate is the mean squared error [MSE]. In the context of the previous question, i.e. how well do these parameter estimates fit the data, this error is just the mean of the squares of the residuals

$$\mathrm{MSE} = \frac{1}{N} \sum_{i=1}^{N} f_i(\hat{\theta}).$$

Therefore it is a function of both the training data set and the parameters themselves. In the previous three cases the MSE can be calculated easily with Python, by using the result object returned by `least_squares`:

```
mse1 = (res1.fun ** 2).mean()
mse2 = (res2.fun ** 2).mean()
mse3 = (res3.fun ** 2).mean()
```

```
MSE in the logistic equation (ex1):         0.000806
MSE in the parametrised circumference (ex2): 0.000226
MSE in the elliptic paraboloid (ex3):        0.001717
```

Now suppose we are using the parameter estimates and model for prediction. For instance, think of the first logistic growth model as the cumulative number of cases during an epidemic. Let's say we are in the middle of the epidemic and only have data available for the first couple of weeks but with to determine when the increase of new cases will start to decline (at t_0) or what the total number of cases will be (approximately K). We estimate the parameters with the available data and make estimates. Then the epidemic finally stops and we wish to evaluate how well our model did.

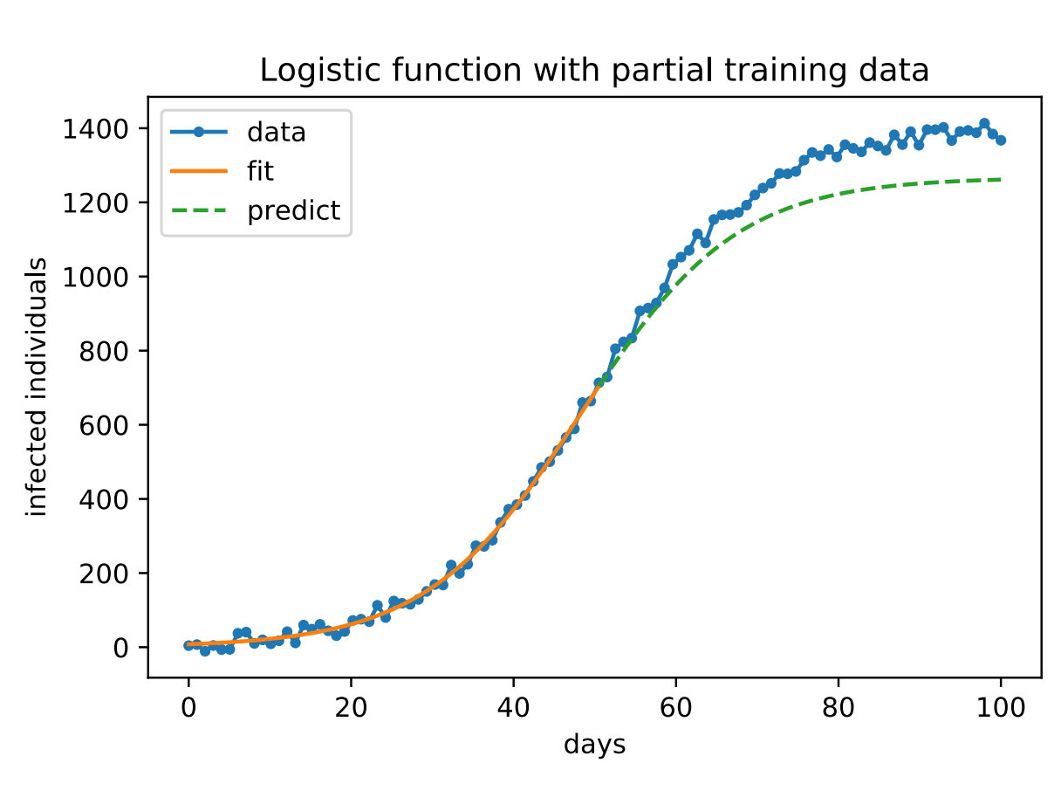
In this context the MSE is called the mean square prediction error [MSPE] and is defined as

$$\mathrm{MSPE} = \frac{1}{Q} \sum_{i=N+1}^{N+Q} (d_i - m(t_i; \hat{\theta}))^2,$$

where we suppose we have made Q predictions using the model at the values of the independent variable $t_{N+1}, \ldots, t_{N+Q}$ using the estimated parameters $\hat{\theta}$.

```
# generate data again
ts = np.linspace(0, 100, 100)
K = 1400; r = 0.1; t0 = 50; noise = 50
ys = y([K, r, t0], ts) + noise * (np.random.rand(ts.shape[0]) - 0.5)

# only this time we only use the first 50% of the data
train_limit = 50 # out of 100 datapoints
def fun(theta):
    return y(theta, ts[:train_limit]) - ys[:train_limit]

# run the parameter estimation again
theta0 = [1000, 0.1, 30]
res4 = least_squares(fun, theta0)

# predict the values for the rest of the epidemic
predict = y(res4.x, ts[train_limit:])
mspe = ((ys[train_limit:] - predict) ** 2).mean()
```

```
K = 1266.9713573318118, r = 0.10463262364967481, t_0 = 48.37440994315726
MSPE for a prediction with 50.0% of the data: 9660.411804269846
```



The previous simulation is extremely sensitive to the amount of training data. In particular, if the training dataset ends much before t_0 the model can be horribly wrong to the point where the prediction can be that the epidemic will still be in the initial exponential phase by day 100. In any case, this issue is beyond the scope of this report as it has more to do with the high sensitivity to small changes in parameters that characterises exponential models.

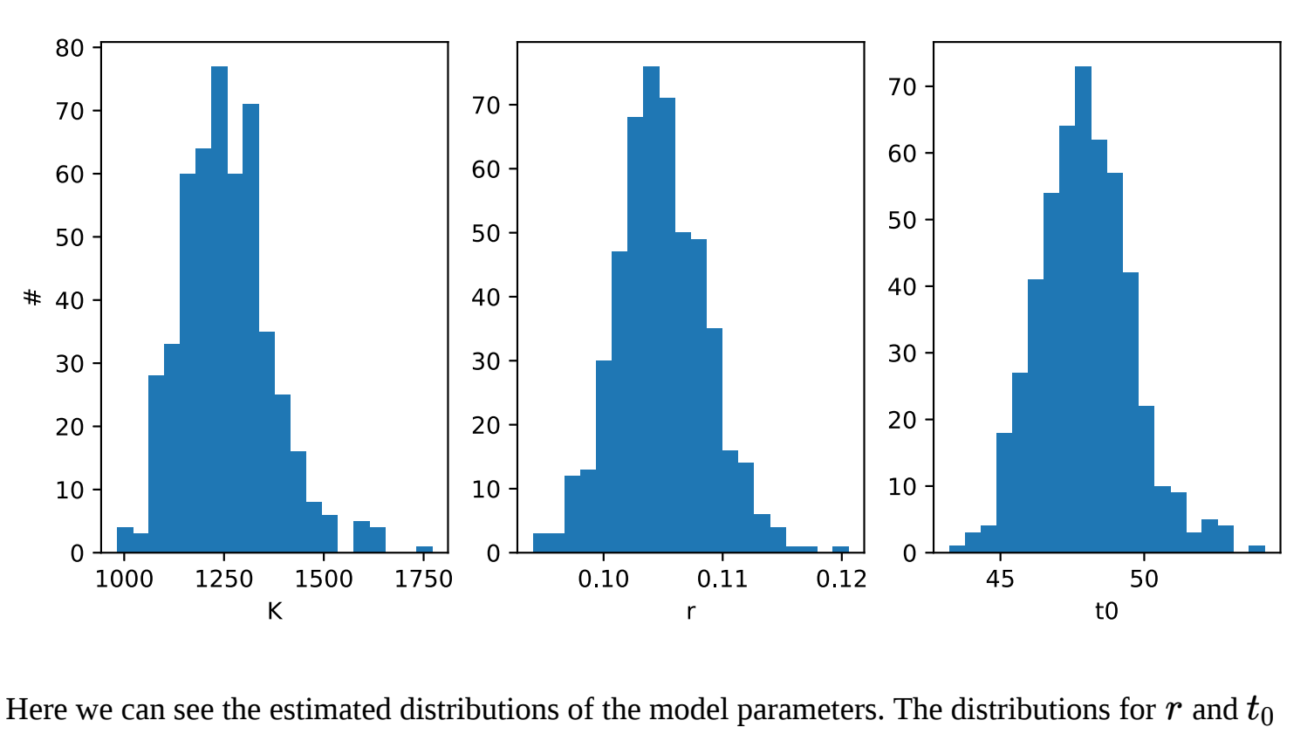
However, the idea of measuring the accuracy of a prediction is in the right direction. Another question we might ask ourselves is how can we get error estimates for our predictions before being able to validate them against real data. In other words, can we predict how wrong we will be in addition to predicting how many infected cases there will be? For general models and general techniques of parameter estimation, there are countless answers to this question. In what follows we focus on a very particular approach that, of course, has its flaws.

Error estimates via residual resampling

One common technique for quantifying errors in parameter estimation is the use of confidence intervals. If the underlying distribution of the data is known, it can sometimes be used to derive confidence intervals via explicit formulas. When the underlying distribution is either unknown or too complex to treat analytically, one can try to estimate the distribution of the data itself. One possible way of doing this is to instead estimate the distribution of the residuals and generate new samples from the fitted values.

More formally, one does NLS fitting and retains the fit values $\hat{y}_i$ and the residuals $f_i(\hat{\theta})$ in addition to the estimated parameters $\hat{\theta}$. Recall that because of the way we have defined f, the relation between these three is $f_i(\hat{\theta}) = \hat{\varepsilon}_i = \hat{y}_i - y_i$. Now we generate new synthetic data, $y_i^* = \hat{y}_i + \hat{\varepsilon}_j$ where $\hat{\varepsilon}_j$ is randomly drawn from the collection of residues $\hat{\varepsilon}_1, \ldots, \hat{\varepsilon}_N$. We use that synthetic data to refit the model and retain interesting quantities such as the values of the parameter estimates. We repeat this process many times to estimate the distribution of the interesting quantities we picked. We are now in a position to estimate the confidence intervals for the parameters.

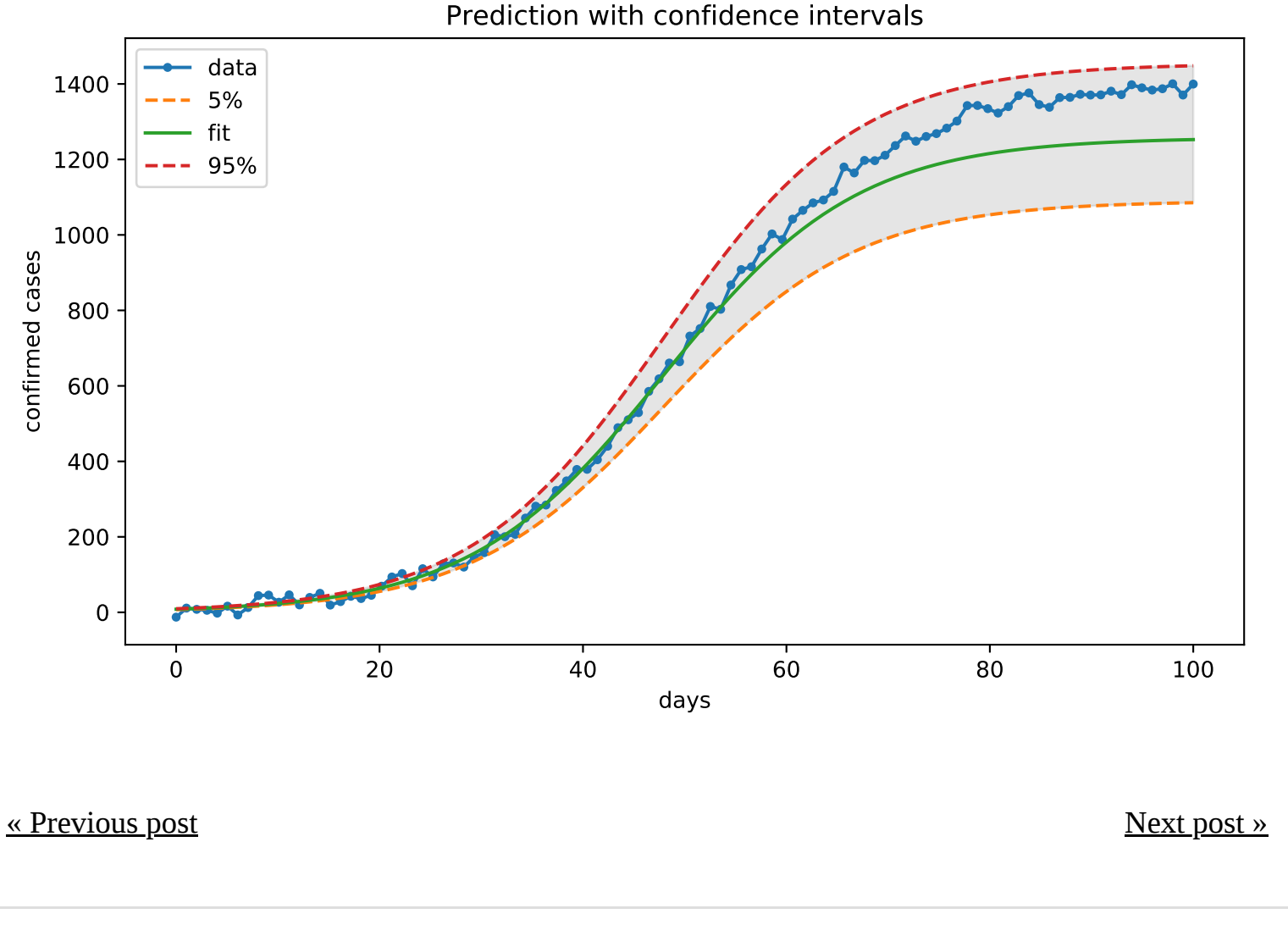
```
# real data
ts = np.linspace(0, 100, 100)
K = 1400; r = 0.1; t0 = 50; noise = 50
ys = y([K, r, t0], ts) + noise * (np.random.rand(ts.shape[0]) - 0.5)

# again, we only use the first 50% of the data
train_limit = 50 # out of 100 datapoints
def fun(theta):
    return y(theta, ts[:train_limit]) - ys[:train_limit]

# run the parameter estimation
theta0 = [1000, 0.1, 30]
res5 = least_squares(fun, theta0)

# use the residuals to estimate the error distribution
eps = res5.fun

# residual resampling
M = 500 # number of resamples
theta_est = []
for _ in range(M):
    # generate synthetic sample
    fit_ys = y(res5.x, ts[:train_limit])
    np.random.default_rng().shuffle(eps)
    synthetic_ys = fit_ys + eps

    # fit the model again
    res = least_squares(lambda theta: y(theta, ts[:train_limit]) - s
    theta_est.append(res.x)

Ks, rs, t0s = np.array(theta_est).transpose()
```



Here we can see the estimated distributions of the model parameters. The distributions for r and t_0 are more or less centred while the distribution for K presents a huge variance and is quite skewed. We choose to do the following, fix the parameters $\hat{r} = \bar{r}$ and $\hat{t}_0 = \bar{t}_0$ and plot the predictions for the model with them and the values for the 95% confidence interval for K which we estimate from the distribution.

```
K05, K95 = np.quantile(Ks, [0.05, 0.95])
K = Ks.mean()
r = rs.mean()
t0 = t0s.mean()

y05 = y([K05, r, t0], ts)
y95 = y([K95, r, t0], ts)
print(f'K = {K:.0f} ({K05:.0f}, {K95:.0f}), r = {r:.3f}, t0 = {t0:.0

fig = plt.figure(figsize = (9, 5))
ax = fig.add_subplot(111)
ax.plot(ts, ys, '.-', label = 'data')
ax.plot(ts, y05, '--', label = '5%')
ax.plot(ts, y([K, r, t0], ts), label = 'fit')
ax.plot(ts, y95, '--', label = '95%')
ax.fill_between(ts, y05, y95, alpha = 0.2, color = 'gray')
ax.set_xlabel('days')
ax.set_ylabel('confirmed cases')
ax.set_title('Prediction with confidence intervals')
ax.legend()

fig.show()
```

```
K = 1258 (1090, 1454), r = 0.105, t0 = 48
```



« Previous post
Next post »