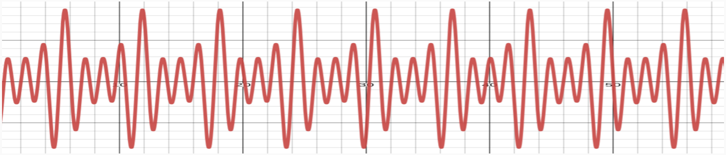
Nonlinear Regression



Prof. Fabrício Olivetti de França

Federal University of ABC

05 February, 2024

Nonlinear Regression

Definition

We now turn our attention to a more generic regression model

$$y^{(i)} = f(x^{(i)}; \theta) + \epsilon^{(i)}$$

As before, $x^{(i)}$ is the vector with the regressor variables, ϵ is a normally distributed disturbance, and f is the expectation function of a parametereized family of functions specified by the parameters θ.

Definition

We use θ for this function to differentiate from the linear models where the parameters were labeled β.

Definition

A **nonlinear regression** model is characterized by the function $f(x;\theta)$ where at least one of the derivatives of the expectation function wrt the parameters depends on at least one of the parameters.

Example

For example, the models:

$$f(x;\theta) = 60 + 70e^{-\theta x}$$
$$f(x;\theta) = \frac{\theta_1 x}{\theta_2 + x}$$

are both nonlinear.

Example

All of the partial derivatives depend on at least one parameter:

$$\begin{aligned}
\frac{\partial f(x;\theta)}{\partial \theta} &= -70xe^{-\theta x} \\
\frac{\partial f(x;\theta)}{\partial \theta_1} &= \frac{x}{\theta_2 + x} \\
\frac{\partial f(x;\theta)}{\partial \theta_2} &= \frac{-\theta_1 x}{(\theta_2 + x)^2}
\end{aligned}$$

Nonlinear models

Fitting nonlinear models requires nonlinear optimization methods that do not guarantee convergence to the global optima.

We can resort to linear approximations to fit the parameters.

Transformably Linear Models

Notice that our second example can be changed into a linear model:

$$
\begin{aligned}
f(x;\theta) &= \frac{\theta_1 x}{\theta_2 + x} \\
\frac{1}{f(x;\theta)} &= \frac{\theta_2 + x}{\theta_1 x} \\
\frac{1}{f(x;\theta)} &= \frac{\theta_2}{\theta_1 x} + \frac{x}{\theta_1 x} \\
\frac{1}{f(x;\theta)} &= \frac{1}{\theta_1} + \frac{\theta_2}{\theta_1 x} \\
g(u;\beta) &= \beta_1 + \beta_2 u \\
u &= \frac{1}{x}
\end{aligned}
$$

This is called **transformably linear models**.

Transformably Linear Models

Notice, though, that when we transform f to a linear model, we are also transforming the disturbance term.

This can change the distribution of the disturbance, making the assumptions invalid.

This linearization can be helpful to find initial guesses for the parameters values, but the fitting should be performed with a nonlinear optimization method.

Conditionally Linear Parameters

Some parameters of a nonlinear model may be **conditionally linear**. For example, the derivative wrt θ_1 of:

$$\begin{aligned} f(x;\theta) &= \frac{\theta_1 x}{\theta_2 + x} \\ \frac{\partial f(x;\theta)}{\partial \theta_1} &= \frac{x}{\theta_2 + x} \end{aligned}$$

depends only on θ_2. We can exploit this by fitting θ_1 as a linear regression conditional to a value of θ_2.

Adjusting the parameters

Least Squares Estimate

As before, we want to find $\hat{\theta}$ such that:

$$\hat{\theta} = \min_{\theta} \|y - \hat{y}\|^2$$

$$\hat{y} = f(x; \theta)$$

Gauss-Newton Method

The **Gauss-Newton** method starts with the idea of using a linear approximation of the expectation function. This can be done by Taylor expansion at an initial guess $\theta^{(0)}$:

$$f(x;\theta) = f(x;\theta^{(0)}) + \sum_{i=1}^{p} \frac{\partial f(x;\theta)}{\partial \theta_i}\Big|_{\theta=\theta^{(0)}} (\theta_i - \theta_i^{(0)})$$

Gauss-Newton Method

The **Jacobian matrix** (J) is the matrix where:

$$J = \begin{bmatrix} \left.\frac{\partial f(x^{(1)};\theta)}{\partial \theta_1}\right|_{\theta=\theta^{(0)}} & \cdots & \left.\frac{\partial f(x^{(1)};\theta)}{\partial \theta_P}\right|_{\theta=\theta^{(0)}} \\ \vdots & \ddots & \vdots \\ \left.\frac{\partial f(x^{(N)};\theta)}{\partial \theta_1}\right|_{\theta=\theta^{(0)}} & \cdots & \left.\frac{\partial f(x^{(N)};\theta)}{\partial \theta_P}\right|_{\theta=\theta^{(0)}} \end{bmatrix}$$

Gauss-Newton Method

So we have:

$$f(x;\theta) \approx f(x;\theta^{(0)}) + J \cdot (\theta - \theta^{(0)})^T$$

with J as a $N \times p$ matrix and θ a $1 \times p$ vector.

Gauss-Newton Method

The approximated residuals are:

$$z(\theta) \approx y - (f(x; \theta^{(0)}) + J \cdot (\theta - \theta^{(0)}))$$
$$z(\theta) \approx (y - (f(x; \theta^{(0)})) - J \cdot (\theta - \theta^{(0)})$$
$$z(\theta) \approx z^{(0)} - J\delta$$

Gauss-Newton Method

Now, we want to find $\delta^{(0)}$ that minimizes $\|z^{(0)} - J^{(0)}\delta\|^2$.

This relates to our solution of least squares, where we wanted to minimize $\|y - X\beta\|^2$.

Gauss-Newton Method

$$J^{(0)} = QR = Q_1 R_1$$
$$\delta^{(0)} = R_1^{-1} Q_1^T z^{(0)}$$

the point $f(x; \theta^{(0)} + \delta^{(0)})$ should be closer to y than $f(x; \theta^{(0)})$.

Gauss-Newton Method

This process is repeated until we cannot minimize the sum of squared residues.

Gauss-Newton Method

```python
df = pd.read_csv("grade.csv")
xcols = ['ETA_mean']

x, y = df[xcols].values, df.grade.values[:,np.newaxis]

def f(x, theta):
  return theta[0]*x/(theta[1] + x)
def J(x, theta):
  return np.hstack([x/(theta[1]+x),
            -theta[0]*x/(theta[1]+x)**2])
def sumofsquares(x, y, t, f):
  return np.sum(np.square(y - f(x, t)))
```

Gauss-Newton Method

```python
def gauss_step(x, y, t0, f, J):
    p = t0.shape[0]
    z = y - f(x, t0)
    q, r = sc.linalg.qr(J(x, t0))
    q1 = q[:,:p]
    r1 = r[:p, :p]
    delta = sc.linalg.solve_triangular(r1, q1.T@z)
    return t0 + delta
```

Gauss-Newton Method

```python
def gauss(x, y, t, f, J, debug=False):
    sos_old = np.inf
    sos = sumofsquares(x, y, t, f)
    if debug:
        print(sos)
    while sos < sos_old:
        t = gauss_step(x,y,t,f,J)
        sos_old = sos
        sos = sumofsquares(x, y, t, f)
        if debug:
            print(sos)
    return t
```

Gauss-Newton Method

```python
t = np.array([205, 0.08])[:,np.newaxis]
gauss(x, y, t, f, J, True)
```

```
245431183.92754266

132435.0246901058

12299140970.505436

array([[ 3.1590888 ],
       [-0.77757744]])
```

Step factor

The last step increased the sum of squares even though we went towards the descent direction.

In these situations we need to decrease the step size when updating the current value of θ.

A simple heuristic is to start with a step size of 1 and half the current step size until the value becomes smaller than a threshold.

A better approach is to perform a line search to find the optimal value for the step size.

Step factor

```python
def gauss_step(x, y, t0, f, J, step):
    p = t0.shape[0]
    z = y - f(x, t0)
    q, r = sc.linalg.qr(J(x, t0))
    q1 = q[:,:p]
    r1 = r[:p, :p]
    delta = sc.linalg.solve_triangular(r1, q1.T@z)
    return t0 + step*delta
```

Step factor

```
def gauss(x, y, t, f, J, debug=False):
    sos_old = np.inf
    sos = sumofsquares(x, y, t, f)
    step = 1
    while step > 1e-12
          and np.square(sos - sos_old) > 1e-6:
        t_i = gauss_step(x,y,t,f,J,step)
        sos_old = sos
        sos = sumofsquares(x, y, t_i, f)
        if sos > sos_old:
            step = step / 2
        else:
            t = t_i
    return t
```

Gauss-Newton Method

```python
t = np.array([205, 0.08])[:,np.newaxis]
gauss(x, y, t, f, J, True)
```

Step factor

```
245431183.92754266
132435.0246901058
12299140970.505436
745214728.2425426
186876895.48195535
47131373.71276204
...
335857.52272309375
335857.5118304746
335857.5091074513
335857.50842667697
array([[ 0.00344322],
       [-0.35566967]])
```

Step factor

Let’s try a different starting value:

```
t = np.array([1.2, -0.1])[:,np.newaxis]
gauss(x, y, t, f, J, True)
```

Step factor

```
208543.89714250065
28984.677830013265
28918.85467418468
28908.738157793916
28907.728409941887
28907.546226136466
28907.519349759037
28907.515040983937
28907.51437219738
array([[ 5.10148992],
       [-0.09892466]])
```

Initial Conditions

Choosing different starting points can lead to different optima, better or worse than the previous attempt.

Confidence Intervals

Approximating the Confidence Interval

With the linear models we could find the joint $(1-\alpha)\%$ CI of the parameters with:

$$(\beta - \hat{\beta})^T x^T x(\beta - \hat{\beta}) \leq Ps^2 F(\alpha, P, N - P)$$

Approximating the Confidence Interval

We can do the same with the linear approximation of the nonlinear model:

$$(\theta - \hat{\theta})^T J^T J(\theta - \hat{\theta}) \leq P s^2 F(\alpha, P, N - P)$$

Approximating the Confidence Interval

And generate the ellipse with:

$$\left\{\theta = \hat{\theta} + \sqrt{Ps^2F(\alpha, P, N-P)}R_1^T d \;\middle|\; \|d\| = 1\right\}$$

Approximating the Confidence Interval

Similarly, the marginal CI can be calculated as:

$$s^2 = \frac{(y - f(x; \hat{\theta}))^2}{N - P}$$

$$se = \sqrt{s^2 \operatorname{diag}(J^T J)}$$

$$\hat{\theta} \pm \mathrm{se} \cdot t(\alpha/2, N - P)$$

Approximating the Confidence Interval

```
j = J(x, theta)
s2 = np.sum((y - f(x,theta))**2)/(n-p)
se = np.sqrt(s2 * np.diag(np.linalg.inv(j.T @ j)))

q, r = sc.linalg.qr(j)
q1 = q[:,:p]
r1 = r[:p, :p]
```

Approximating the Confidence Interval

```python
alpha = 0.05
marginal_t = []
for i, coef in enumerate(theta.flatten()):
    v = np.abs(se[i]*stats.t.ppf(1 - alpha/2, n-p))
    marginal_t.append((coef - v, coef + v))
    print(f"{coef - v:0.4f} <= theta_{i} <= {coef + v:0.4f}")
```

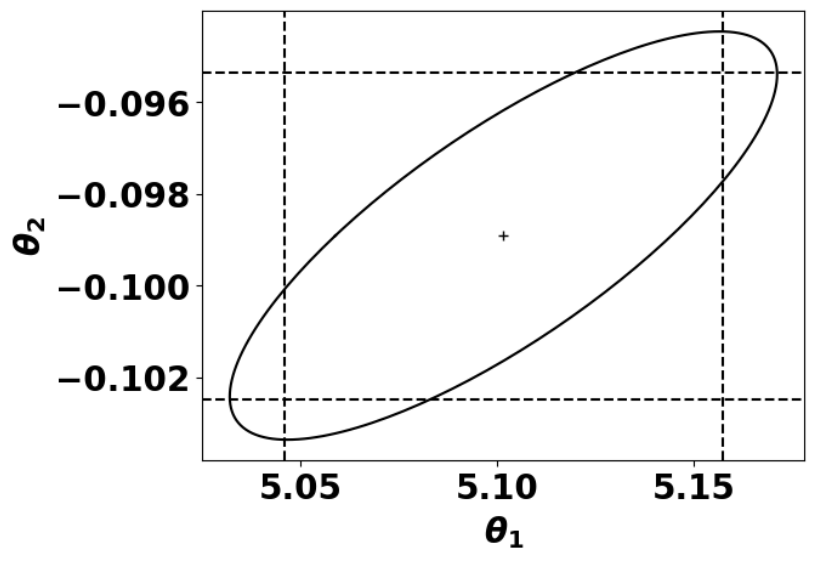
Approximating the Confidence Interval

```
omegas = np.pi*np.arange(0, 2.01, 0.01)
const = np.sqrt(p*s2*stats.f.ppf(1-alpha, p, n-p))
         *np.linalg.inv(r1)
thetapts = np.array([theta + const
            @ np.array([np.cos(w), np.sin(w)])
            for w in omegas])
```

Approximating the Confidence Interval

```
_,ax = plt.subplots()
ax.plot(thetapts[:,0], thetapts[:,1], color='black')
ax.plot(theta[0], theta[1], '+', color='black')
ax.axvline(x=marginal_t[0][0], linestyle='--', color='black')
ax.axvline(x=marginal_t[0][1], linestyle='--', color='black')
ax.axhline(y=marginal_t[1][0], linestyle='--', color='black')
ax.axhline(y=marginal_t[1][1], linestyle='--', color='black')

ax.set_xlabel(r"$\theta_1$")
ax.set_ylabel(r"$\theta_2$")
```

Approximating the Confidence Interval



Approximating the Confidence Interval

We can calculate the CI of the predictions with:

$$\hat{y}^{(i)} \pm s\|J_i R_1^{-1}\| t(\alpha/2, N-P)$$

$$\hat{y} \pm s\|J R_1^{-1}\| \sqrt{f(\alpha, P, N-P)}$$

Approximating the Confidence Interval

```python
for i, y_i in enumerate(f(x, theta).flatten()):
    jr = (j[0,:] @ np.linalg.inv(r1))
    v = np.sqrt(s2 * (jr.T@jr))*np.abs(stats.t.ppf(1 - alpha/2, n-p))
    print(f"{y_i - v:0.2} <= y_{i} <= {y_i + v:0.2}")
```

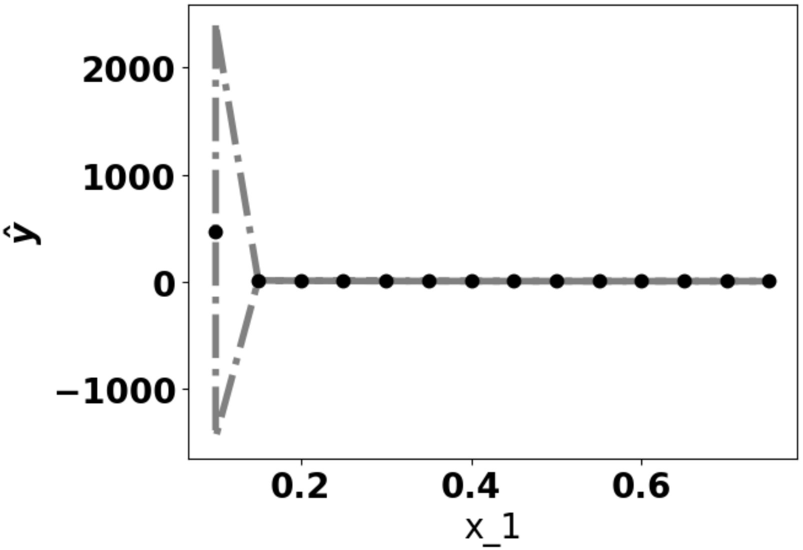
Approximating the Confidence Interval

```
fxi = np.arange(0.1, 0.8, 0.05)[:,np.newaxis]
j = J(xi, theta)

yy = np.linalg.norm(j @ np.linalg.inv(r1), axis=1)
      * np.sqrt(s2*p * stats.f.ppf(1-alpha, p, n-p))
y_h = f(xi, theta)
```

Approximating the Confidence Interval

```python
_,ax = plt.subplots()
ax.plot(xi[:,0], y_h, '.', color='black', markersize=15)
ax.fill_between(xi[:,0], y_h - yy, y_h + yy, edgecolor='gray', facecolo
    linewidth=4, linestyle='dashdot', antialiased=True)
ax.set_xlabel('x_1')
ax.set_ylabel(r'$\hat{y}$')
```

Approximating the Confidence Interval



Approximating the Confidence Interval

When we have more than 2 parameters, we can plot pairwise approximations of the joint intervals by using the i, j-th rows and columns of R_1.

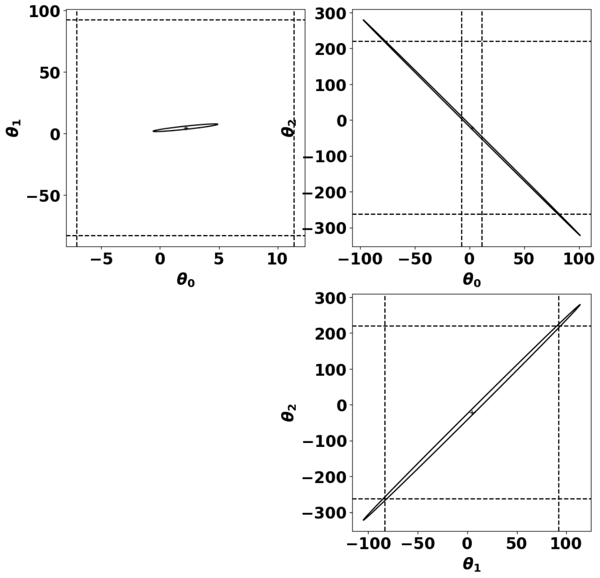
Approximating the Confidence Interval

```
def f(x, theta):
    return np.exp(-theta[0]*x) + theta[1]*x/(theta[2] + x)
def J(x, theta):
    return np.hstack([-x*np.exp(-theta[0]*x),
            x/(theta[1]+x), -theta[0]*x/(theta[1]+x)**2])

theta2 = np.array([0.1, 0.1, 0.1])[:,np.newaxis]
theta2 = gauss(x, y, theta2, f, J)
```

Approximating the Confidence Interval

```
omegas = np.pi*np.arange(0, 2.01, 0.01)

_,axs = plt.subplots(2,2, figsize=(12,12))
for i in range(2):
  for j in range(i+1, 3):
    ixs = [i,j]
    const = np.sqrt(p*s2*stats.f.ppf(1-alpha, p, n-p))
            *np.linalg.inv(r1[[i,j], :][:,[i,j]])
    thetapts = np.array([theta2[ixs].flatten()
           + const
           @ np.array([np.cos(w), np.sin(w)]) for w in omegas])
```

Approximating the Confidence Interval



Closing

Terminology learned today

- **nonlinear regression:** is a model where at least one of the derivatives of the expectation function wrt the parameters depends on at least one of the parameters
- **transformably linear models:** are nonlinear models that can be transformed to linear.
- **conditionally linear:** are models that some of the parameters are linear when we fix the other parameter values.

Further reading

- Chapter 2 and 3 of Bates, Douglas. “Nonlinear regression analysis and its applications.” Wiley Series in Probability and Statistics (1988).

Next lecture

- Symbolic Regression

Acknowledgments