EngSci 721

Inverse Problems and Learning From Data

Oliver Maclaren (oliver.maclaren@auckland.ac.nz)

1. *Basic concepts* [5 lectures + 1 Tutorial]

   Forward vs inverse problems. Well-posed vs ill-posed problems. Algebra and calculus of inverse problems (left and right inverses, generalised and pseudo inverses, resolution operators, matrix calculus). Representing higher dimensional problems (image data etc).

2. *Instability and regularisation in linear and nonlinear problems* [6 lectures + 1 Tutorial]

   Instability and related issues for generalised inverses. Introduction to regularisation and trade-offs. Tikhonov regularisation. Higher-order Tikhonov regularisation. Sparsity and regularisation using different norms. Truncated singular value decomposition. Iterative regularisation, including stochastic/mini-batch gradient descent.

3. *Further topics* [3 lectures + 1 Tutorial]

   Regularisation parameter choice, including statistical and machine learning views of regularisation. Confidence sets for linear and nonlinear models. Physics-informed machine learning and neural networks.

Module overview

Inverse Problems and Learning From Data (*Oliver Maclaren*)
[~14 lectures/3 tutorials]

Lecture 13: Nonlinear models

Topics:

- Fitting a simple nonlinear scientific model (ODE)
- Constructing confidence sets
- Complications

EngSci 721 : Lecture 13 : Nonlinear models

Fitting simple nonlinear models to data

Eg 'How can I fit a nonlinear ODE to data?'

'What about stochastic / large models?'

→ simple practical example

→ pointers to lit. for more complex cases

- L derivative calculation
- L surrogates/emulators
- L ensemble methods
- L scientific machine learning (L14)
- + Ru's section (Bayes)

Nonlinear?

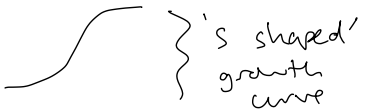
We have indicated several times that our ideas apply to both linear & nonlinear (in parameters) models

→ however we have spent a lot of time on linear models!

Let's look at a 'simple' nonlinear example, where number of params < data points, ie the nonlinear version of a 'tall' system

We will also consider some elementary statistical ideas like noise & confidence intervals

Logistic growth



S-shaped growth curves describe many physical & biological phenomena, e.g.

- increase of cell number in cell culture
- population growth in ecology
- infection curves in disease dynamics
- reaction rate in enzyme kinetics

etc!

The simplest (but not only!) such model is given by the logistic growth model:

$$\frac{dC}{dt} = rC\left(1 - \frac{C}{K}\right)$$

$$C(0) = C_0$$

C = count or concentration
r = growth rate
K = 'carrying' capacity (saturation value)

Logistic growth

Technically we can solve this analytically for:

$$C(t) = \frac{KC_0}{C_0 + (K - C_0)e^{-rt}}$$

Notes:

- $t \to \infty$, $C(t) \to K$
- C/K small $\Rightarrow$

$$\frac{dC}{dt} = rC(1 - C/K) \approx rC$$

- model is nonlinear in parameters r, C
- data have form $(t, C)_{i=1,\dots m}$

However, pretend we don't know analytical soln!

Trajectory matching

A simple way to fit this model is by nonlinear least squares:

$$\min_{\theta} \| y^{data} - y^{model}(\theta) \|^2 \quad \Big\} \text{ nonlinear in } \theta$$

where
- $\theta = (r, K)$
- $y^{data} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{bmatrix} = \begin{bmatrix} y(t_1) \\ y(t_2) \\ \vdots \\ y(t_d) \end{bmatrix}$
- $y^{model}(\theta) = \begin{bmatrix} y^{model}(t_1, \theta) \\ \vdots \\ \vdots \end{bmatrix}$

→ If we don't know $y^{model}(\theta)$ analytically we simple call a numerical solver for the ODE to evaluate for any parameter

→ The above is sometimes called 'trajectory matching', as we aim to fit the 'trajectory' to data.

Optimisation

For small/fast to solve problems we can just use standard libraries to solve.

→ nonlinear least squares form: can use 'least_squares' function from scipy.optimize (python). Similar functions exist in MatLab, Julia, R etc.

→ for more general objective functions can use eg 'minimize' from scipy.optimise (python) or eg fminsearch (MatLab) etc.

→ these interface to various derivative-based & derivative-free optimisers.

↳ will compute derivatives for you if needed.

Derivative calculation?

For standard languages & optimizers, if you need derivatives with respect to a function $f(y(\theta))$ of the model output $y(\theta)$, the library will typically compute them using finite differences, e.g. for step-size h,

$$\frac{\partial f(y(\theta))}{\partial \theta_i} \simeq \frac{f(y(\theta + h e_i)) - f(y(\theta))}{h}$$

where e_i is the unit vector in the ith dir.

- If f is vector-valued, this gives a column of derivatives for each θ_i

$$\frac{\partial f}{\partial \theta^T} = \begin{matrix} f_1 \\ f_2 \\ f_3 \end{matrix} \begin{bmatrix} \theta_1 & \theta_2 & \cdots \\ \uparrow & & \\ \downarrow & & \end{bmatrix}$$

- If scalar-valued it will just be a number for each θ_i:

$$\frac{\partial f}{\partial \theta^T} = f \begin{bmatrix} \theta_1 & \theta_2 & \cdots \\ & & \end{bmatrix}$$

Derivative calculation

This finite difference approach

- has a cost that scales with number of parameters, for each iteration
  - ok for small, cheap problems, expensive for large & or costly to run models
- noisy / inaccurate & dependent on choice of h

More sophisticated approaches:

- automatic / algorithmic
- symbolic / analytical

Modern ML libraries like tensorflow / pytorch etc are really 'just' autodiff libraries!

+ Julia has 'native' autodiff!

Idea: automatically differentiate any code you write!

Example

```
# logistic
from scipy.integrate import solve_ivp

def dydt_p(t,y,prm):
    # params
    r = prm[0]*1E-3 #scale relative to 1E-3
    K = prm[1]*10
    # vars
    C = y[0]
    # ode
    dydt = r*C*(1-C/K)
    return dydt
```

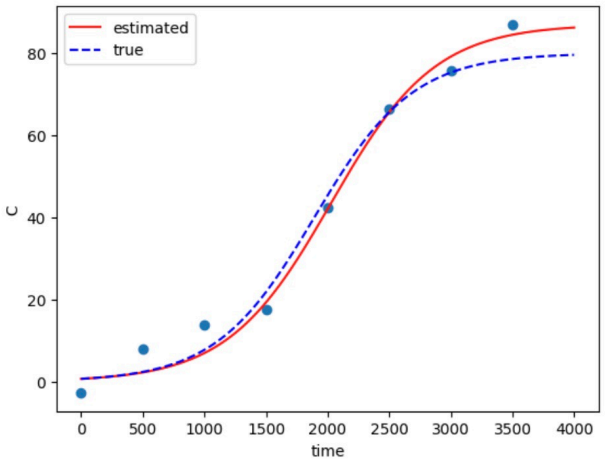
```
# ODE for given param.
dydt = lambda t, y: dydt_p(t,y,prm)
```
✓ 0.0s

```
# parameters
r_true = 2.5
K_true = 8
prm_true = np.array((r_true,K_true))

# initial condition
C0 = 0.7
y0 = [C0]

# ODE for given param. (closure...)
dydt = lambda t, y: dydt_p(t,y,prm_true)

# t_data
t_span=[0,4000]
t_grid = np.arange(t_span[0],t_span[1],1)
t_data = np.arange(0,4000,500)

# true solution for given param
solution_object = solve_ivp(fun=dydt,t_span=t_span,t_eval=t_data,y0=y0,dense_output=True)
sol = solution_object.sol

sigma = 5
y_true = sol(t_grid)[0]
y_data = sol(t_data)[0] + np.random.normal(loc=0,scale=sigma,size=len(t_data))
```

Example

```
# best fit
def residual(prm):
    # ODE for given param. (closure...)
    dydt = lambda t, y: dydt_p(t,y,prm)

    # t_data
    t_span=[0,4000]
    t_data = np.arange(0,4000,500)

    # true solution for given param
    solution_object = solve_ivp(fun=dydt,t_span=t_span,t_eval=t_data,y0=y0,dense_output=True)
    sol = solution_object.sol

    y_pred = sol(t_data)[0]

    return y_data - y_pred
```

```
# nonlinear least squares
from scipy.optimize import least_squares
x_initial = np.array([1,1])
result = least_squares(fun=residual,x0=x_initial)
prm_est = result.x
# ODE for given param. (closure...)
dydt = lambda t, y: dydt_p(t,y,prm_est)
# est solution for given param
solution_object = solve_ivp(fun=dydt,t_span=t_span,t_eval=t_data,y0=y0,dense_output=True)
sol = solution_object.sol
plt.plot(t_data,y_data,'o')
plt.plot(t_grid,sol(t_grid)[0],'r-',label='estimated')
plt.plot(t_grid,y_true,'b--',label='true')
plt.xlabel('time')
plt.ylabel('C')
print(prm_est)
```



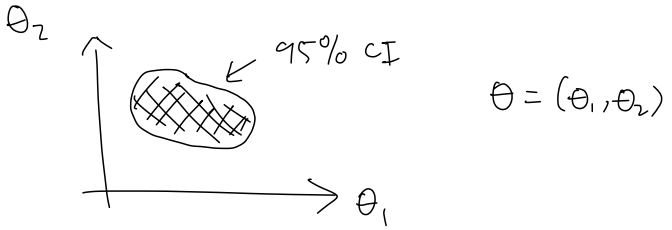
Confidence sets

As discussed in the previous lecture, if we know the noise level, we can form '$1-\alpha$' confidence sets (eg $95=1-5$) by 'hypothesis test inversion':

$$CI(\theta) = \left\{ \theta : \frac{1}{\sigma^2} \| y^{obs} - y^{mod}(\theta) \|^2 \leq c_{1-\alpha} \right\}$$

for 'cut-off' $c_{1-\alpha}$ given by the $1-\alpha$ threshold of the chi-square distribution

ie 'confidence set = all non-rejected parameters'



→ Applies to nonlinear models with additive error: $y = f(\theta) + \epsilon$

→ Can generalise using likelihood functions (see Rn)

Example

Here we can just 'grid up':

```
r_min = 0
r_max = 10
r_grid = np.linspace(r_min,r_max,500)

K_min = 5
K_max = 15
K_grid = np.linspace(K_min,K_max,500)

nllikes = np.zeros((len(r_grid),len(K_grid)))
for i, ai in enumerate(r_grid):
    for j, bj in enumerate(K_grid):
        nllikes[i,j] = nllikelihood((ai,bj))
```

✓ 4m 42.0s

(would have to sample or profile in high dim)

Plot contour of 95% boundary:

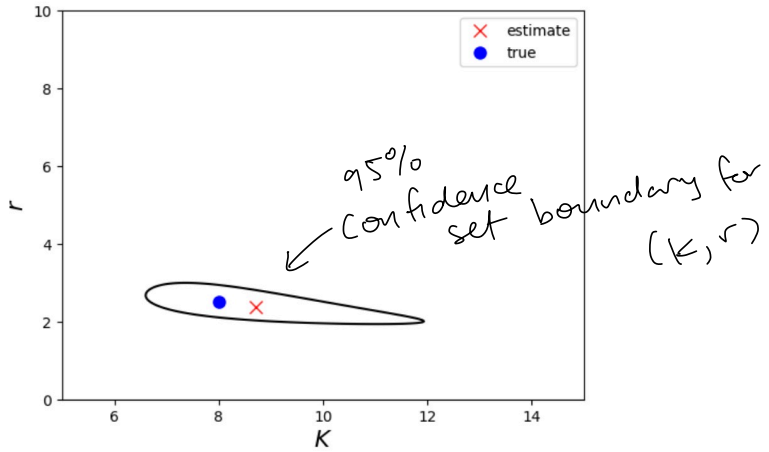
```
from scipy.stats import chi2
delta = chi2.ppf(q=0.95,df=len(y_data))

contours = plt.contour(K_grid,r_grid, nllikes, levels=[delta], colors='black')

plt.xlabel(r'$K$',fontsize=16)
plt.ylabel(r'$r$',fontsize=16)

plt.plot(prm_est[1],prm_est[0],markersize=8,color='r',marker='x',linestyle='None',label='estimate')
plt.plot(prm_true[1],prm_true[0],markersize=8,color='b',marker='o',linestyle='None',label='true')
plt.legend()
plt.show()
```

→

Example

```
from scipy.stats import chi2
delta = chi2.ppf(q=0.95,df=len(y_data))

contours = plt.contour(K_grid,r_grid, nllikes, levels=[delta], colors='black')

plt.xlabel(r'$K$',fontsize=16)
plt.ylabel(r'$r$',fontsize=16)

plt.plot(prm_est[1],prm_est[0],markersize=8,color='r',marker='x',linestyle='None',label='estimate')
plt.plot(prm_true[1],prm_true[0],markersize=8,color='b',marker='o',linestyle='None',label='true')
plt.legend()
plt.show()
```

✓ 0.1s



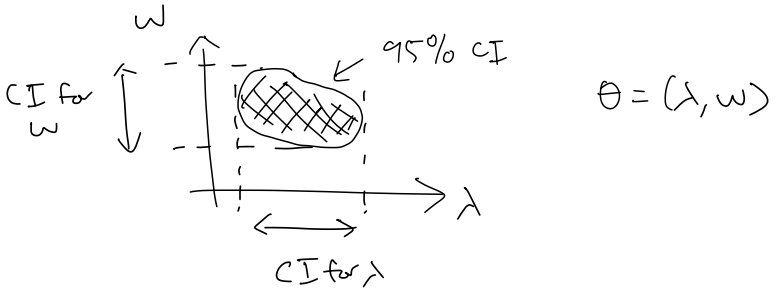
Note: choose units for r & K to make similar scale

(K actually 80, here use $\bar{K}=8$ & $K=10\bar{K}$)

Profile projections for 'targeted CIs'

we can 'project' the full parameter confidence set onto the individual parameter axes to get conservative ($\geq 1-\alpha$ coverage) CIs for individual param.

Idea: we can't reject a subcomponent (parameter) λ of a parameter vector θ if there exists a 'complementary' parameter w such that $\theta = (\lambda, w)$ is not rejected



Profile projections: computing

idea: compute min for each candidate target parameter value & compare to threshold

→ if max within threshold then exists a pair in confidence set (why?)

$$\text{Profile}(\lambda) = \min_{w|\lambda} \left\{ \frac{1}{\sigma^2} \| y^{obs} - y^{mod}(w,\lambda) \| \right\}$$

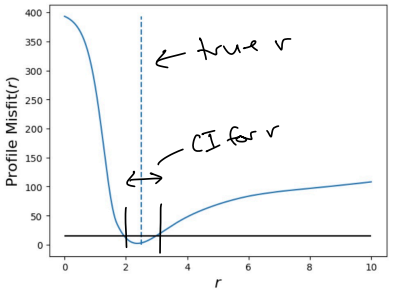
---

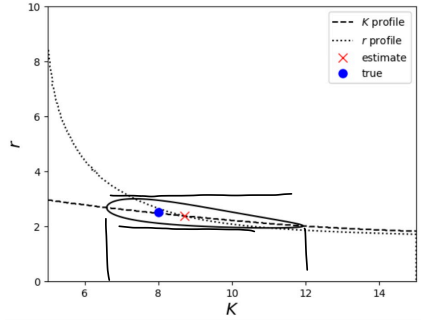
can generalise to profile likelihood functions & other fit measures

---

Example

```
# plot profile paths
nuisance_K = [K_grid[np.argmin(row)] for row in nllikes]
nuisance_r = [r_grid[np.argmin(col)] for col in nllikes.T]

contours = plt.contour(K_grid,r_grid, nllikes, levels=[delta], colors='black')

plt.xlabel(r'$K$',fontsize=16)
plt.ylabel(r'$r$',fontsize=16)

plt.plot(K_grid,nuisance_r,label=r'$K$ profile',color='k',linestyle='--')
plt.plot(nuisance_K[2:],r_grid[2:],label=r'$r$ profile',color='k',linestyle=':')

plt.plot(prm_est[1],prm_est[0],markersize=8,color='r',marker='x',linestyle='None',label='estimate')
plt.plot(prm_true[1],prm_true[0],markersize=8,color='b',marker='o',linestyle='None',label='true')

plt.legend()
plt.show()
```
✓ 0.1s



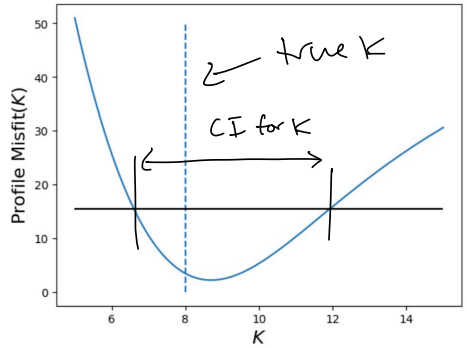


CI(r): [1.94, 2.99]

true r: 2.5

```
# project via 'manually' profiling over grid cf resolving optimisation.
misfit_K = np.min(nllikes,axis=0)
plt.plot(K_grid,misfit_K)

plt.xlabel(r'$K$',fontsize=16)
plt.ylabel(r'Profile Misfit($K$)',fontsize=16)
plt.vlines(x=K_true,ymin=0,ymax=np.max(fit_K),linestyles='--')
plt.hlines(y=delta,xmin=K_min,xmax=K_max,linestyles='-',color='k')
K_llim = K_grid[misfit_K <= delta][0]
K_ulim = K_grid[misfit_K <= delta][-1]
print((K_llim,K_ulim))
plt.show()
```
✓ 0.1s

```
np.float64(6.603206412825651), np.float64(11.93386773547094))
```



CI(K): [6.60, 11.93]

true K: 8

Unknown error?

If the error is unknown, we can (try to) estimate as another parameter

→ OK for small dim problems!

But need to think about bias & regularisation for very flexible functions

↳ essentially our regularisation parameter

→ need either another est. eqn for σ or full likelihood function

↳ can't min $\frac{1}{\sigma^2}\|y - f(\theta)\|^2$ in both θ & σ!

→ another approach is to estimate via different empirical model then treat as fixed in sci model

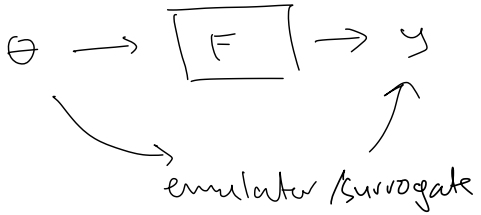
↳ underestimates overall uncert but often 'OK'

Expensive models

For expensive models we can't afford to run as many times and/or compute derivatives

Strategies include:

Construct 'emulator'/'surrogate' from limited budget of runs:

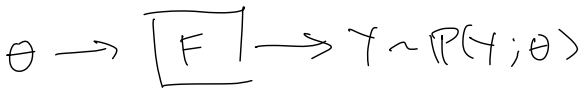


eg
- Gaussian process
- neural net
- Spline

Pay cost up front† but can be cheaper long term as emulator cheap to evaluate

≠ 'amortised'

Stochastic simulation models



Don't know $P(Y;\theta)$ in closed form but can simulate samples

Options

→ Can often find approximate loss function with distribution independent of parameter
- ↳ compute dist. for some est.
- ↳ invert hypothesis test

→ Construct 'surrogate' $P(Y;\theta)$ eg 'synthetic likelihood'

→ simulation-based: ABC, LFZI etc

See canvas

Scientific machine learning

Hybrid { flexible data model
scientific regularisation
↳ not exactly satisfied

→ Eg assignment:

spline fit + ODE penalty

Current hype: 'PINNs'

Physics-informed neural networks

→ we'll look at neural nets in last lecture