Design, test, debug and profile

Simon Wood, University of Edinburgh, U.K.

Introduction

- **Structured programming** involves *designing* your programme before you start coding, so that the programming task is broken down into manageable functions reflecting the structure of the task being programmed.
  - Properly designed well structured programmes are easier to code, test, debug, read and maintain or modify.
- **Testing** is an essential part of programming reliably.
- **Debugging** is the process of finding and correcting code errors.
  - Use of the appropriate tools and the right approach can save large amounts of time and effort.
- **Profiling** is the process of improving your code's computational efficiency (i.e. speed or memory use), by establishing where the computational effort is going, and carefully examining the most expensive steps to see if they can be made more efficient.
  - Efficient use of profiling can be the difference between needing a cluster or laptop to run your programme.

Structure

- *Design* before you *code*!
  1. *Purpose*. Write down what your code should do. What are the inputs and outputs, and how do you get from one to the other?
  2. *Design considerations*. What are the other important issues to bear in mind? e.g. does the code need to be fast? or deal with large data sets? is it for one off use, or for repeated general use?
  3. *Design the structure*. Decide how best to split the code into functions each completing a well defined manageable and testable part of the overall task. Write down this structure.
  4. *Review*. Will the structure achieve the purpose and meet the design considerations?
- Generally the more you write down of the design and code specification before you start coding, the quicker the coding process will be and the fewer mistakes you will make.
- Designing as you code is a recipe for making a mess.

Readability

- As you write it, everything about your code will often seem very obvious to you.
- Two weeks later it won't seem obvious.
- Neither will it seem obvious to anyone else.
- Help others and your later self by:
  1. Using (short) meaningful variable names where possible.
  2. Explaining what the code does using frequent `# comments`, including a short overview at the start of each function.
  3. Laying out the code so that it is clear to read. White space is free.
  4. Always having a another team member review your code and comments when team-working.

Simple design example: ridge regression

- Consider the standard linear statistical model

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

  $\mathbf{X}$ is an $n \times p$ model matrix, $\mathbf{y}$ a response vector, $\boldsymbol{\beta}$ a parameter vector and $\boldsymbol{\epsilon}$ a vector of i.i.d. zero mean errors, variance σ^2.

- The least squares estimate of $\boldsymbol{\beta}$ is

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$$

- When p is large, $\hat{\boldsymbol{\beta}}$ can become quite unstable, and a *ridge regression* estimate can have better predictive performance.

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|^2 = (\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$$

  where λ is a *smoothing parameter* to be selected somehow.

Simple design example: ridge regression

- As $\lambda \to \infty$ the coefficient estimates become less variable, but are shrunk towards zero.
- λ can be chosen to minimize a measure of prediction error, the *Generalized Cross Validation* (GCV) score

$$V(\lambda) = \frac{n\|\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}\|^2}{\{n - \mathrm{tr}(\mathbf{A})\}^2}$$

  where $\mathbf{A} = \mathbf{X}(\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\mathsf{T}$.
- Note that if $\boldsymbol{\mu} = E(\mathbf{y})$ then $\hat{\boldsymbol{\mu}} = \mathbf{A}\mathbf{y}$.
- Also $\mathrm{tr}(\mathbf{A})$ is a measure of the *effective degrees of freedom* of the model - it lies between 0 and p.

Designing a ridge regression function

- **Aim**: Function should take inputs **y** and **X** and a sequence of $\log \lambda$ values, search for the GCV minimizing λ, and return the corresponding $\hat{\boldsymbol{\beta}}$.
- **Considerations**: reasonable to assume that the inputs are correct, important to check that GCV score has a proper minimum, efficiency of some importance.
- **Outline Design**
  - `ridge(y,X,lsp)` loops through log smoothing parameters in `lsp` calling `fit.ridge` for each to evaluate GCV score. Hence find optimal log smoothing parameter. Plot GCV trajectory using `plot.gcv` and return best fit $\hat{\boldsymbol{\beta}}$.
  - `fit.ridge(y,X,sp)` fit model for a single smoothing parameter value using a Cholesky method to solve for $\hat{\boldsymbol{\beta}}$, returning $\hat{\boldsymbol{\beta}}$, GCV score and EDF.
  - `plot.gcv(edf,lsp,gcv)` plot GCV against EDF and log smoothing parameters.

Code, test, debug

- Once you have a decent design on paper - which will usually involve writing out further details for each function - code it up.
- If possible test each function as it is written. This has 2 parts
  1. Initial interactive informal testing.
  2. Tests that the function works as intended for the intended range of inputs, written as code that you can run repeatedly - e.g. each time a function's code is modified. These are known as *unit tests*.
- For any moderately interesting piece of code the testing, or subsequent use, is likely to produce some errors - *bugs*. The cause of these have to be found - *debugging*.
- Rarely is staring at code an efficient approach to debugging.
- Act as a detective, gather evidence, test theories and *use debugging software*.

Debugging approach

- Try to avoid introducing bugs, but accept that humans are error prone. Work in a way that reduces the risk: design before you code. Aim for a good structure that makes coding easy, with readily testable components. Comment carefully.
- Be aware that code modification is often where bugs creep in.
- Interesting code will have bugs at some point. To find them be scientific. Gather data on what might be wrong - including your basic assumptions. Form hypotheses about the problem, and experiment to test them.
- The key skills are narrowing down the problem, and checking that the code is *actually* doing what you *think* it does.
- One careful read through is useful to look for stand-out errors. Code staring is not. Get active and test things.

Profiling

- Once your code is tested and functioning correctly, it may still be inconveniently slow to run.
- This is a frequent and substantial problem in big data settings.
- *Profiling* is the process of investigating which parts of your code are taking the most time to run, so that you can examine closely the time consuming steps and try to improve them.
  1. In `R` inefficiencies often arise from loops with many iterations but little work done at each iteration. Vectorization helps.
  2. In statistical methods generally, inefficiencies often arise from careless coding of matrix operations.
- `R` can measure the time taken by each function of code using . . .

```
Rprof()
## some code to profile
Rprof(NULL)
summaryRprof()
```

- Note that it can be important to profile memory usage too.