Developing models, simulations, and optimisation

Joe Parker

13th November 2017

Simulating data

Last week we ran a lot of other peoples' code. In this practical we're going to get our hands nice and dirty. We'll start with some light simulation: we're going to simulate some data from a linear relationship. Suppose a collection of 10 plants have been grown in a greenhouse. Each plant has had increasing hours of sunlight (`plant_sun`, from 1 to 10) and their height after a month will be. A month passes. Before we head out to the greenhouse with our tape-measure, can we simulate their heights? Suppose that a plant with 1hr each day normally grows to be 0.3m high, and every extra hour of sunlight adds 0.3m onto a plant's height after a month.

1) Can you simulate and plot the data as a `plant_height` variable? *(If there is no measurement error, or noise from random variation)*
2) Suppose you want to include some random variation as well? Try to introduce normally-distributed errors with a standard deviation of 0.25.

Continues overleaf…

You probably spotted the formula for this one - it's a linear relationship through the origin, of course. To add error, we simply need an error term: `plant_height = (0 + 0.3 * plant_sun) + rnorm(10,0,1.5)`.

But which error term should we use? The normal distribution can take any standard deviation we like. We might also use exponential or uniform distributions to generate errors - or anything we like. Try a few now, and plot the results over your first simulation (***hint: use*** `points()` ***instead of*** `plot()` ***to overlay points - and you may want to use different colours or the*** `pch=` ***par command to distinguish between them.***)

3) What do you notice about the impact of the type of error on the simulation? In a real simulation, how could you choose an error function?
4) So far we've only added noise in the error term. But we might be unsure of the parametisation. How can you simulate noise in the line equation itself (stick to an s.d. of about 0.1)?
5) Out of curiosity, can we break R's `lm()` function yet? Try fitting a regression line and plotting it. Do the coefficients' estimates from the regression agree with the parametised model we simulated from?
6) Look at the significance values for the line intercept and slope estimates in the coefficients table of the ANOVA. Has R estimated them to be different zero? Was it correct? Try repeating the simulation a few times - what do you notice?

Optimisation

So far, we've had R's `lm()` to do all the hard work for us. But let's imagine we didn't: can we write an algorithm to fit a straight line, through the origin, to this data?

1) Try and write some pseudocode to do this. Don't worry about formatting.

To make this simpler, we'll break it down. First, the goodness-of-fit:

```
# first we'll write a utility function to tell us what the error
# is for a given parameter
error_function = function(predicted,observed){
  # calculate errors
  SSerrors = sum((observed - predicted)^2)

  # return them
  return(SSerrors)
}

get_line_model_error = function(values,response,slope){
  # fits a line response ~ values * slope through the origin
  # and returns sum of squared errors

  # get predicted values (nb assumes origin)
  predicted = values*slope

  # what is their error?
  return(error_function(predicted,response))
}

get_line_model_error(plant_sun,plant_height_noisy,0.3)
```

We can write a fairly dumb optimiser now, all we need is something called a ***proposal mechanism***. This simply describes how we want to try new combinations of parameters, and the conditions under which we'll use them to update our model. Here's an **insanely** dumb one to start you off, can you do better?

```
# proposal mechanism
slope_proposal = function(existing_slope){
  return(existing_slope + rnorm(1,0,0.2))
}

# start with a random slope from 0,1
starting_slope = runif(1,0,1)
plot(plant_sun,plant_height)
points(plant_sun,plant_noisiest,pch=19,col="red")
```

```
abline(lm(plant_noisiest~plant_sun),col="dark grey",lty=3,lwd=4)

# the loop where we actually optimise the slope
hits = 0
miss = 0
for (i in 1:20){
  # work out the error for the current slope guess
  actual_error = get_line_model_error(plant_sun,plant_height_noisy,starting_slope)

  # guess a new slope
  new_slope = slope_proposal(starting_slope)

  # what's the error for the new slope?
  new_error = get_line_model_error(plant_sun,plant_height_noisy,new_slope)

  # if the new slope's better, accept it
  if(new_error < actual_error){
    # accept our guess
    hits = hits + 1
    starting_slope = new_slope
    abline(0,starting_slope,col="green")
    print(paste("HIT ",hits))
  } else {
    # boo.. rubbish guess
    miss = miss + 1
    abline(0,new_slope,col="red")
    print(paste("miss ",miss))
  }
}

print(paste("total hits/misses: ",hits," / ",miss))
```

2) It seems like we're making a lot of rubbish guesses. Tweak the optimiser so that the SS errors for each guess go into a variable, and plot them against guess number. Collect the current 'best' error at each guess, too.
3) Does this seem efficient to you? Why not?
4) Can you think of a better proposal mechanism? Implement it and rerun the optimiser
5) Why is this better?
6) Simulate the input data again, but this time with a slope of 10. Try your awesome optimiser on this data. What happens? Why?

We've stumbled on something called **tuning** - that is, we really want a proposal mechanism that takes into account the size of the existing error, and the order of magnitude of the current guess. Can you have a go?

Other models

Regressions are well and good, but a bit boring. Take a look at this data I've simulated according to my own special secret formula (`mystery.tdf`):

1) How can you model this data? (You aren't allowed categories, e.g. means fitted with an ANOVA. Sorry..)
2) Can you simulate data under this model yourself?
3) You can probably guess the model parameters. Can you fit them, and calculate the errors? ***Hint: you'll need to predict values from your model, calculate fits, propose new parameters, and accept or reject them…***

Harder

If you want to try something hard, I've hidden some high points in `shape.tdf`. They are x, y, and z co-ordinates, but to keep things simple the z-coordinates only take on 0 or 1 values, like the line example. Load up the .tdf file and look at the data. I've writen a quick plot surface to help you visualise the data:

```
# custom plotting function
plot_shape = function(shape_data){
  plot(shape_data[,1],shape_data[,2],col="light grey",pch=4)
  points(shape_data[shape_data[,3]>0.5,1],shape_data[shape_data[,3]>0.5,2],col="black",pch=19)
}
```

Can you fit this? Hint: the line equation for a circle is all points that exactly satisfy `x^2 + y^2 == 1`…

Finally…

If you really, really want to stretch yourself, try `hard_shape.tdf`. Hint you might want to think about translations…