Developing models, simulations, and optimisation

Joe Parker

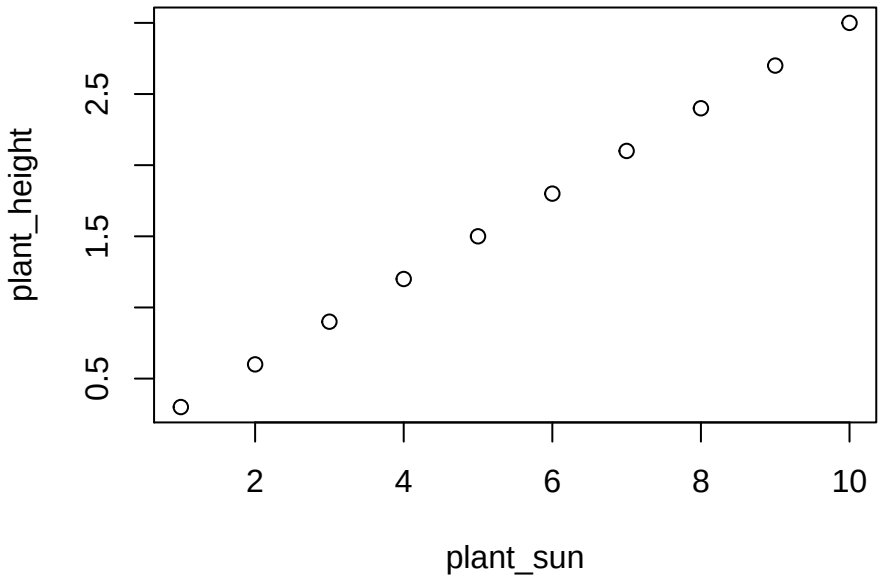
12th November 2018

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.

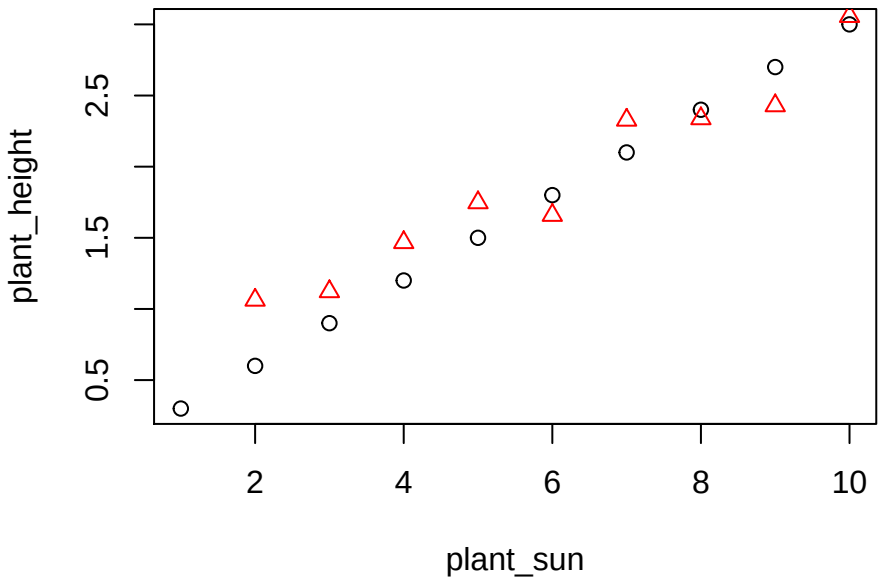
1) Can you simulate and plot the data as a `plant_height` variable? *(If there is no measurement error, or noise from random variation)*

```
plant_sun = 1:10
plant_height = plant_sun*0.3
plot(plant_sun,plant_height)
```



2) Suppose you want to include some random variation as well? Try to introduce normally-distributed errors with a standard deviation of 0.25.

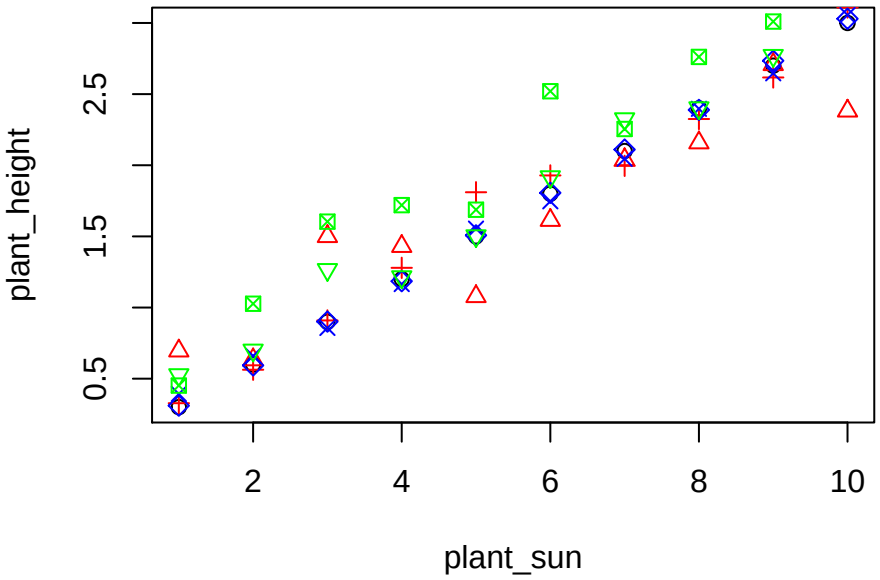
```
plant_height_noisy = plant_sun*0.3 + rnorm(10,0,0.25)
plot(plant_sun,plant_height)
points(plant_sun,plant_height_noisy,pch=2,col="red")
```


plant_height
0.5
1.5
2.5
2
4
6
8
10
plant_sun

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.*)

```
plot(plant_sun,plant_height)
# normal, s.d. = 0.25
points(plant_sun,plant_sun*0.3 + rnorm(10,0,0.25),pch=2,col="red")
# normal, s.d. = 0.1
points(plant_sun,plant_sun*0.3 + rnorm(10,0,0.1),pch=3,col="red")
# unif, bound = [-0.1,+0.1]
points(plant_sun,plant_sun*0.3 + runif(10,-0.1,0.1),pch=4,col="blue")
# unif, bound = [-0.05,+0.05]
points(plant_sun,plant_sun*0.3 + runif(10,-0.05,0.05),pch=5,col="blue")
# exponential, rate=5
points(plant_sun,plant_sun*0.3 + rexp(10,5),pch=6,col="green")
# exponential, rate=3
points(plant_sun,plant_sun*0.3 + rexp(10,3),pch=7,col="green")
```

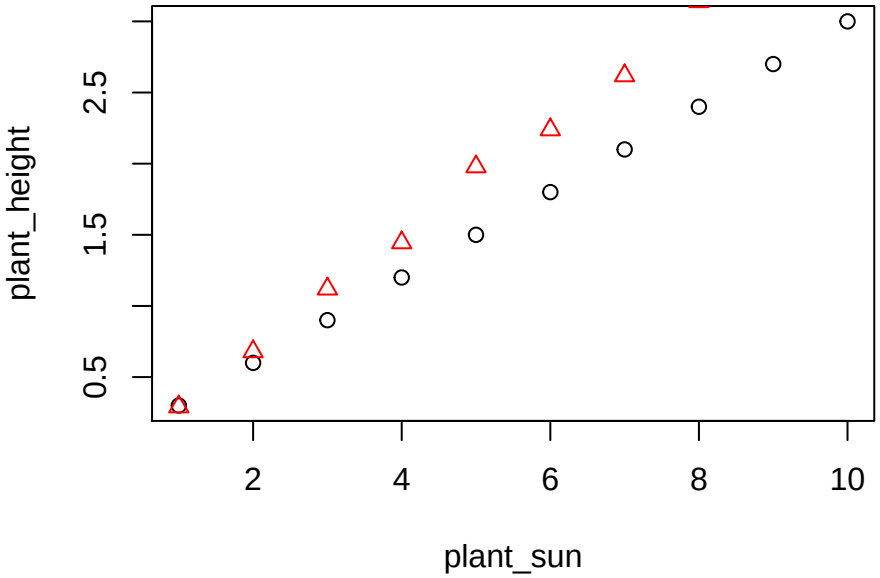


3) What do you notice about the impact of the type of error on the simulation? In a real simulation, how could you choose an error function?

Highly sensitive to error funciton parameters. Will usually be observed from data

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)?

```
# make sure we only randomise the coefficients once(!)
noisy_a = rnorm(1,0,0.1)
noisy_b = rnorm(1,0,0.1) + 0.3
plant_noisiest = (noisy_a + noisy_b*plant_sun)+rnorm(10,0,0.1)
plot(plant_sun,plant_height)
# normal, s.d. = 0.25
points(plant_sun,plant_noisiest,pch=2,col="red")
```



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?

Bloody well should do...

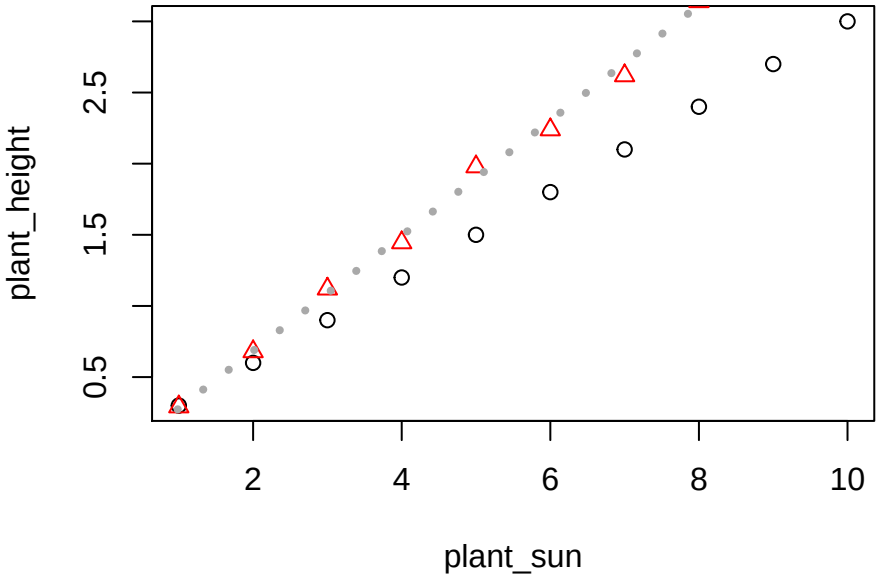
```
anova(lm(plant_noisiest~plant_sun))
```

```
## Analysis of Variance Table
##
## Response: plant_noisiest
##           Df  Sum Sq Mean Sq F value    Pr(>F)
## plant_sun  1 13.5180 13.5180  3116.8 1.176e-11 ***
## Residuals  8  0.0347  0.0043
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
summary(lm(plant_noisiest~plant_sun))
```

```
##
## Call:
## lm(formula = plant_noisiest ~ plant_sun)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.089947 -0.046862  0.003108  0.029303  0.101370
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.125182   0.044989  -2.782   0.0238 *
## plant_sun    0.404789   0.007251  55.828 1.18e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.06586 on 8 degrees of freedom
## Multiple R-squared:  0.9974, Adjusted R-squared:  0.9971
## F-statistic:  3117 on 1 and 8 DF,  p-value: 1.176e-11
```

```
# replot
plot(plant_sun,plant_height)
points(plant_sun,plant_noisiest,pch=2,col="red")
abline(lm(plant_noisiest~plant_sun),col="dark grey",lty=3,lwd=4)
```



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?

R will probably get the slope right mostly but the intercept will wobble

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.

```
Variables: input, output, target_error, parameters

parameters = random()

While actual_error > target_error:
  actual_error = calculate_fit(input,output,parameters)
  proposed_parameters = random()
  proposed_error = calculate_fit(input,output,proposed_parameters)
  if(proposed_error < actual_error): parameters = new_parameters
```

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

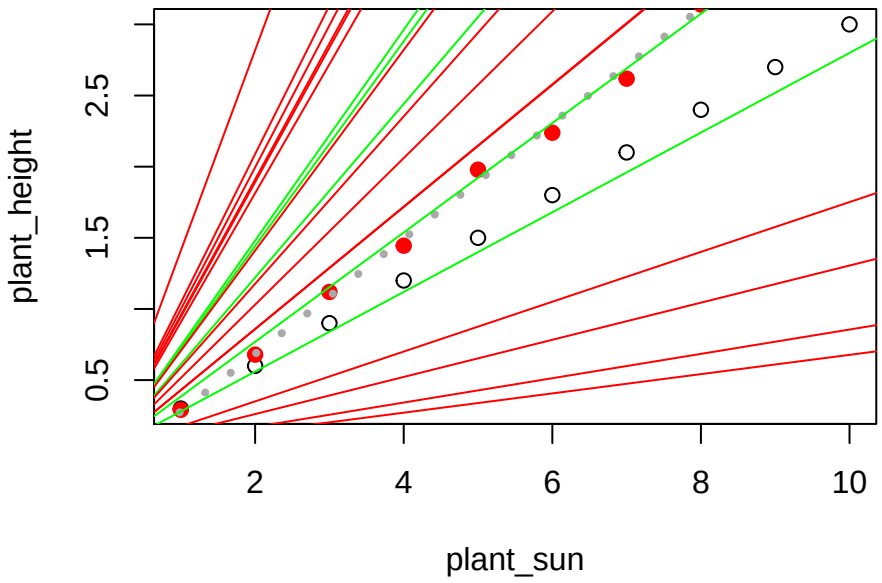
```
# first we'll write a utility function to tell us what the error
# is for a given parameter
error_function = function(predicted,observed){
  # calculate errors
  SSerrors = sum((observed - predicted)^2)

  # 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)
```

```
## [1] 0.6097089
```

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))
  }
}
```




```
## [1] "miss  1"
## [1] "miss  2"
## [1] "miss  3"
## [1] "HIT  1"
## [1] "miss  4"
## [1] "HIT  2"
## [1] "miss  5"
## [1] "miss  6"
## [1] "HIT  3"
## [1] "miss  7"
## [1] "HIT  4"
## [1] "miss  8"
## [1] "miss  9"
## [1] "miss  10"
## [1] "HIT  5"
## [1] "miss  11"
## [1] "miss  12"
## [1] "miss  13"
## [1] "miss  14"
## [1] "miss  15"
```

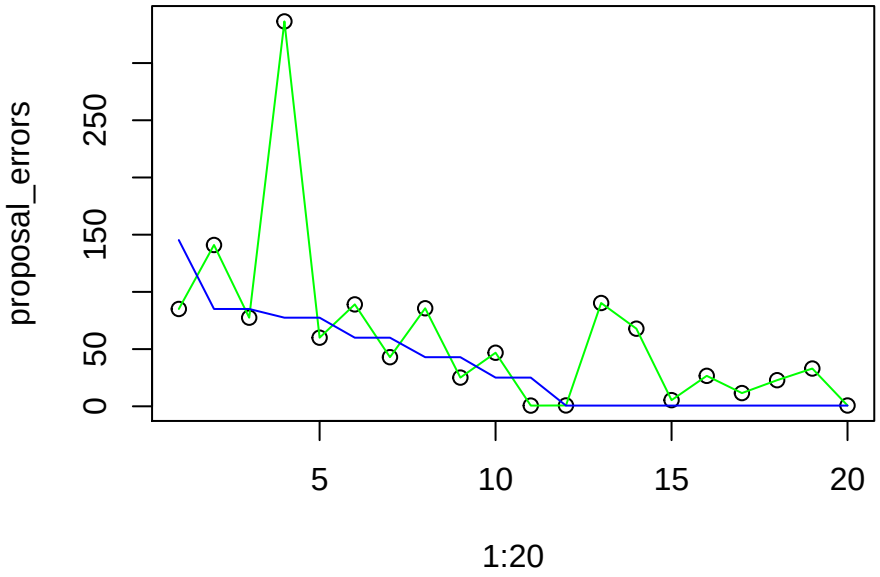
```
print(paste("total hits/misses: ",hits," / ",miss))
```

```
## [1] "total hits/misses:  5  /  15"
```

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.

```
starting_slope = runif(1,0,1)

proposal_errors = rep(NA,20)
model_errors = rep(NA,20)

# the loop where we actually optimise the slope
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
    starting_slope = new_slope
  } else {
    # boo.. rubbish guess
  }
  #store the error
  proposal_errors[i] = new_error
  model_errors[i] = actual_error
}

plot(1:20,proposal_errors)
lines(1:20,proposal_errors,col="green")
lines(1:20,model_errors,col="blue")
```



3) Does this seem efficient to you? Why not?

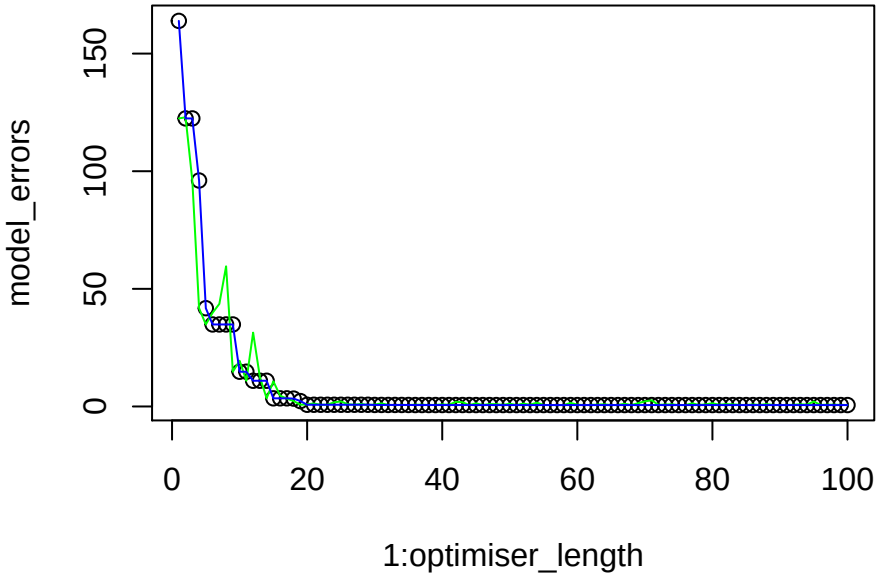
We propose guesses with the same width of error every time, even if our last guess is close to the true value.

4) Can you think of a better proposal mechanism? Implement it and rerun the optimiser

```
# new proposal mechanism
clever_slope_proposal = function(existing_slope){
  return(existing_slope + rnorm(1,0,0.1*existing_slope))
}
```

```
starting_slope = runif(1,0,1)

optimiser_length = 100
proposal_errors = rep(NA,optimiser_length)
model_errors = rep(NA,optimiser_length)

# the loop where we actually optimise the slope
for (i in 1:optimiser_length){
  # 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 = clever_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
    starting_slope = new_slope
  } else {
    # boo.. rubbish guess
  }
  #store the error
  proposal_errors[i] = new_error
  model_errors[i] = actual_error
}

plot(1:optimiser_length,model_errors)
lines(1:optimiser_length,proposal_errors,col="green")
lines(1:optimiser_length,model_errors,col="blue")
```



5) Why is this better?

We're making smaller changes

6) Simulate the input data again, but this time with a slope of 10. Try your awesome optimiser on this data. What happens? Why?

Hm, not so great this time - we set up the optimiser for much smaller numbers

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?

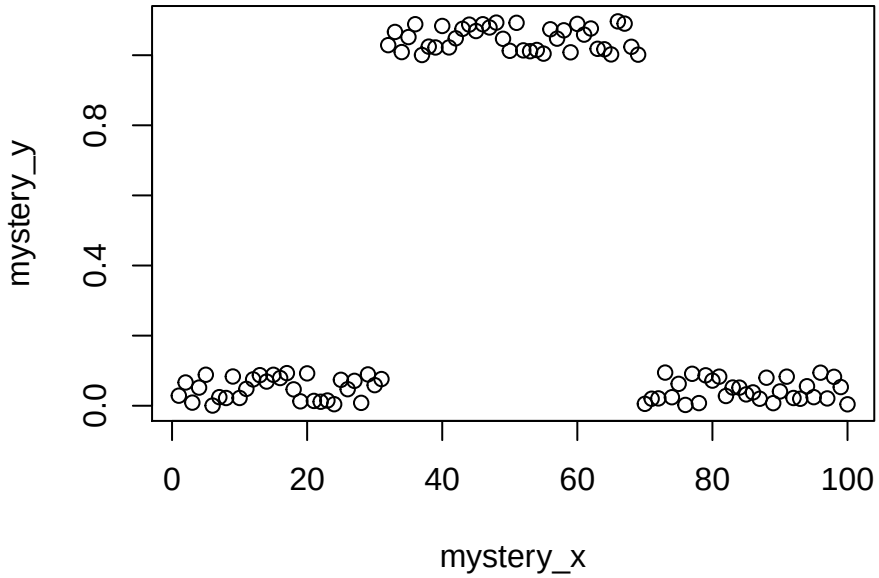
Other models

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 (try not to look at it):

```
mystery_x = 1:100
mystery_y = runif(100,0,0.1)
mystery_y[32:69] = mystery_y + 1
```

```
## Warning in mystery_y[32:69] = mystery_y + 1: number of items to replace is
## not a multiple of replacement length
```

```
plot(mystery_x,mystery_y)
```




```
write.table(data.frame(mystery_x,mystery_y),"mystery.tdf",col.names=T,sep="\t")
```

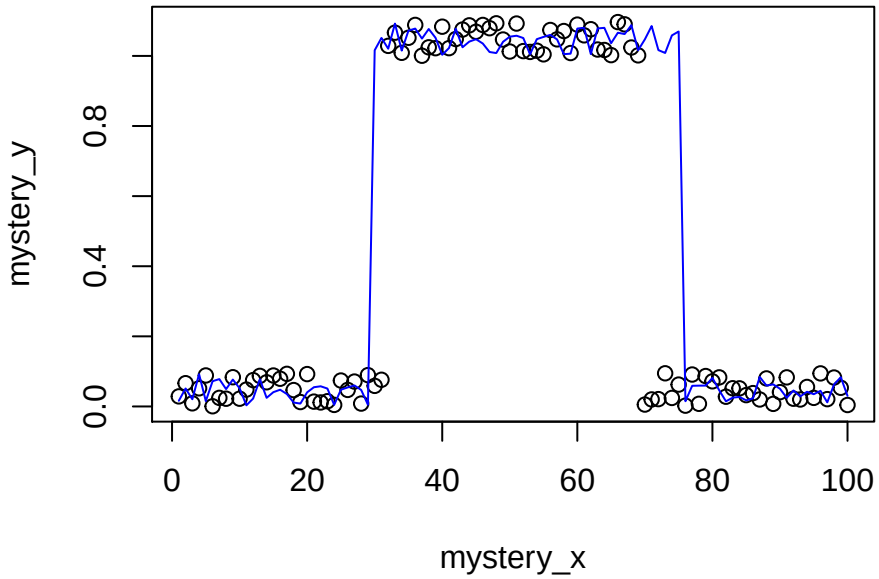
1) How can you model this data?

It's a line at 1, plonked in a line of 0s. We can model it with a 'start, finish' or a 'start, length'

2) Can you simulate data under this model yourself?

```
# simulate in units
simulate_mystery = function(input_x,start,line_length){
  mystery_y_simulated = runif(100,0,0.1)
  mystery_y_simulated[start:(start+line_length)] = mystery_y_simulated + 1
  return(mystery_y_simulated)
}

plot(mystery_x,mystery_y)
lines(mystery_x,simulate_mystery(mystery_x,30,45),col="blue")
```

```
## Warning in mystery_y_simulated[start:(start + line_length)] =
## mystery_y_simulated + : number of items to replace is not a multiple of
## replacement length
```



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...*

```r
# simulate in units
predict_mystery = function(input_x,start,line_length){
  mystery_y_predicted = rep(0,100)
  # in case the line parameter tries to clip the boundries, max/min it
  start = max(start,1)
  line_end = min(start+line_length,100)
  mystery_y_predicted[start:line_end] = 1
  return(mystery_y_predicted)
}

# proposal mechanism
line_parameter_proposal = function(param){
  return(round(param+runif(1,-4,4)))
}

# fit the model

# first we need starting values
line_start=round(runif(1,1,100))
line_length=round(runif(1,1,100))

# calc existing fit
predicted_line = predict_mystery(mystery_x,line_start,line_length)
current_error = error_function(predicted_line,mystery_y)

for(i in 1:10){
  # propose new line params
  new_start  = line_parameter_proposal(line_start)
  new_length = line_parameter_proposal(line_length)

  # predict new y values
  predicted_line = predict_mystery(mystery_x,new_start,new_length)

  # recalculate error
  new_error = error_function(predicted_line,mystery_y)

  # accept if better
  if(new_error < current_error){
    line_start = new_start
    line_length = new_length
    current_error = new_error
  }
  print(paste(i,current_error,line_start,line_length))
}
```

```
## [1] "1 46.4285282949833 96 49"
## [1] "2 46.4285282949833 96 49"
## [1] "3 46.4285282949833 96 49"
## [1] "4 46.4285282949833 96 49"
## [1] "5 46.4285282949833 96 49"
## [1] "6 43.8237933395812 99 52"
## [1] "7 43.8237933395812 99 52"
```

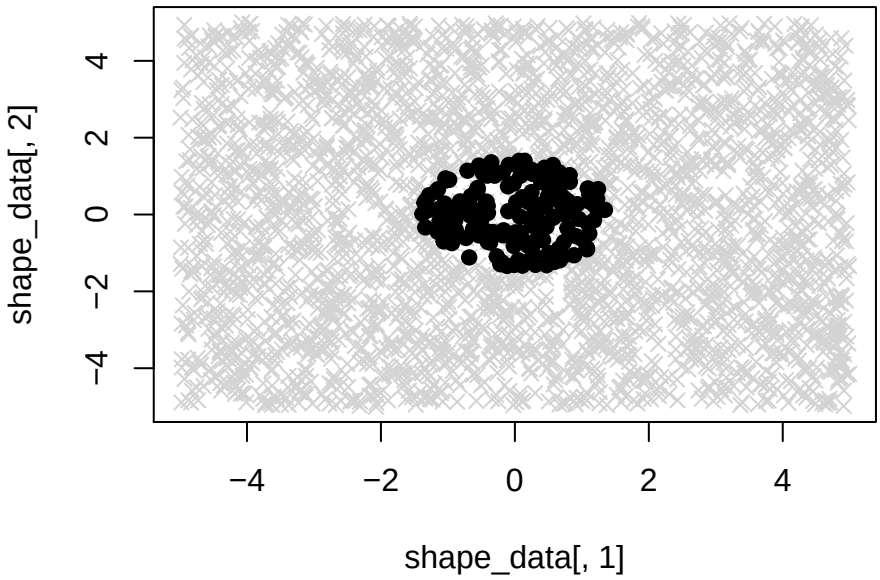
```
## Warning in observed - predicted: longer object length is not a multiple of
## shorter object length
```

```
## [1] "8 43.8237933395812 99 52"
## [1] "9 43.8237933395812 99 52"
## [1] "10 42.9302609835202 100 49"
```

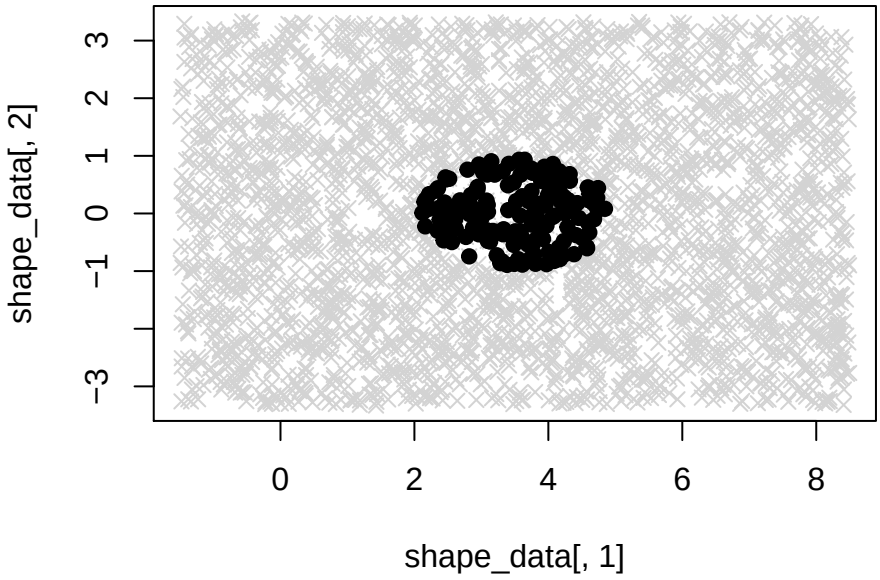
Harder

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:

```
# set up
shape_x = runif(3000,-5,5)
shape_y = runif(3000,-5,5)
shape_z = 0
shape=data.frame(shape_x,shape_y,shape_z)

# add a circle
shape$shape_z[(shape$shape_x^2+shape$shape_y^2)<=2] = 1

# write out
write.table(shape,"shape.tdf",sep="\t",col.names=T)

# translate to be cheeky and scale
shape_translated_x = shape_x + 3.5
shape_translated_y = shape_y / 1.5
shape_translated = data.frame(shape_translated_x,shape_translated_y,shape$shape_z)
shape_translated$shape.shape_z = shape_translated$shape.shape_z + rnorm(length(shape_x),0,0.1)
write.table(shape_translated,"hard_shape.tdf",sep="\t",col.names=T)

# 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)
}

plot_shape(shape)
```




```
plot_shape(shape_translated)
```




```
# 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...**