Fitting\_functions

Wouter van den Bos

16 März 2017

rm(list=ls()) # clear the workspace.. always a good idea before you start a new project  
set.seed(123) # so we will all be talking about the same ranom results  
setwd("~/Dropbox/WOUTER/TEACHING/UvA/BREIN&COG/practicum/LC1/") # set to your working dir with the files you need.

# 1 creating a function

Creating functions in R is easy. The basic structure is:  
*function\_name* <- **function**(list of variables) {

do something with variables

**return**(another variable)  
}

for example:

himmelblau <- function(x,y){  
 h=(x^2+y-11)^2+(x+y^2-7)^2  
 return(h)  
}  
  
## or just enter a list of variable v = c(x,y)  
himmelblau <- function(v){  
 x<-v[1]  
 y<-v[2]  
 h=(x^2+y-11)^2+(x+y^2-7)^2  
 return(h)  
}

# 2 Fitting a function to data

In this exercise we will fit different functions to data and compare their fits. In the first step we will generate the data, in the second step we will fit different functions to it. This excersise will also be used to illustrate the difference between fit and prediction.

### 2.1 Generating Simulation data

### Q0 To begin write your first function: and call this function *curve*. To be clear the input should be *x* and it should return *y*.

### Q1 (provide answers inline, you need the constructed code later)

The next step is to generate some data using this function:

1. create an empty vector c() called **sim\_data**
2. make and save a list called **points** with numbers between 1 and 10 in steps of .5 using **seq()**

3a. Use the numbers in **points** to generate y values using your **curve()** function and store them in **sim\_data**

3b. also add noise to the **sim\_data** from a uniform distribution between -1.5 to 1.5

Now we will create a dataframe with these x and y values called **dat\_full** and a subset of that data called **dat\_short** :

# create a dataframe  
dat\_full<-as.data.frame(cbind(points,sim\_data))  
colnames(dat\_full)<-c("x","y")  
# create a subset for illstruation  
dat\_short<-subset(dat\_full,dat\_full$x<6)

# Generating curves to fit to the data:

Next we will generate a few curves that will then fit to the data we just generated. The output of the functions that we will write will be the **Mean Squared Error** or **MSE** which is what is says.

The function will:

1.take in the values for **x** and **y**

1. generate estimations or prediction of **y**, called **y\_hat** using a specific function (in this example linear)
2. compare the estimated **y\_hat** with the true **y**, the difference is called the **error**
3. MSE is the mean of the squared error

## create several functions to compare fits  
linear <- function(data,v){  
 b1<-v  
 y\_hat=b1\*data$x  
 error = y\_hat-data$y  
 MSE = mean(error^2)  
 return(MSE)  
}

### Q2 Generate a polynomial function.

Generate a polynomial function:

# Fitting functions

We can use the standard optimization toolbox **optim** to find the best fitting parameters for a function. The optimization tool box requires us to provide:  
- starting values for the parameters  
- function\_name  
- data to fit function on  
- optimization method - possibly upper and lower bounds for the parameters

What the optim toolbox will do is try to find the parameters settings that correspond the minimum value that is returned by the function that is being optimized. In our case it is trying to minimize the Mean Squared Error. It is beyond the scope of this course to discus the different methods, but they all minimize.

Let's run through an example:  
- starting value = 0  
- function = linear - data = dat\_short - method = "Brent" (only one you can choose if you have just one parameter) - upper and lower bound values = 10 and -10

results are stored in results1

## Perform some analyses on the short\_data  
  
results1<-optim(c(0),linear,data=dat\_short,method="Brent", upper=10, lower=-10)  
results1

## $par  
## [1] 0.5047321  
##   
## $value  
## [1] 0.6895368  
##   
## $counts  
## function gradient   
## NA NA   
##   
## $convergence  
## [1] 0  
##   
## $message  
## NULL

When you look at the results1 you get the parameter estimate:

results1$par

## [1] 0.5047321

But you also get the corresponding MSE:

results1$value

## [1] 0.6895368

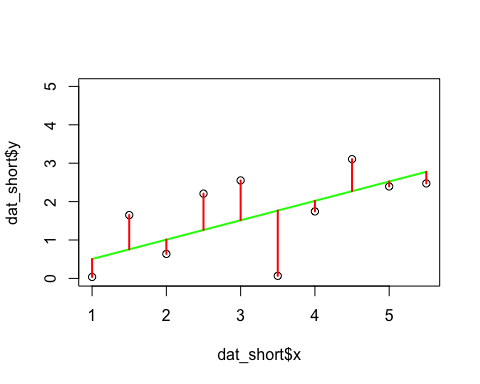
## Visualizing the Fit

to create residul (error) lines in plot we will create an empty list and fill it with the predicted y\_hat values for each x value in the data set, given the parameter we just estimated results1$par:

fit\_data<-c()  
for(i in 1:nrow(dat\_short)){  
 fit\_data[i]<-results1$par[1]\*dat\_short$x[i]  
}  
dat\_short$error1<-fit\_data

Next we plot the data

# simply use the plot() function to plot the true data  
plot(dat\_short$x,dat\_short$y,ylim = c(0,5))  
# add a line with lines() that corresponds with the predicted y\_hat.   
# to do so you need to invoke the linear function b1\*x and use the estimated b1  
# so in this case: results1$par[1]\*dat\_short$x  
lines(dat\_short$x,results1$par[1]\*dat\_short$x,col="green",lwd=2)  
# here we add segments that indicate the distance or error from the model to the data:   
segments(dat\_short$x,dat\_short$y,dat\_short$x,dat\_short$error1, col="red",lwd=2)



### 

### Q3 Now fit the the polynomial function to the data and compare the model fits

* Fit the model as above, and display the best fitted lines in a plots
* Compare the MSE of the best fit for each model and indicate which is the best fitting one. Why do you think did the best fitting model won?

Hint 1: use the method="Nelder-Mead", it does not require definitions of upper and lower bounds  
Hint 2: for the starting values use c c(-5,9,-4,1,.01)

# Cross-Validation

We have fit the models only using part of the data (dat\_short), now we can see how the models perform if we use them to predict the data points that we have left out. This is the very basic principle of cross-validation. We first fit the model on part of the data and then see how it performs on a seperate set of data (validation). In practice we may use slightly different strategies to do this, but the principle is the same.

### Q4 check the goodness of fit for the full data set using the parameter estimates derived from the small datasets. Describe which model fits the data the best and explain why, also plot the results to illustrate your point.

**hint 1**: just use same script as before but change the data file to dat\_full. **hint 2**: use ylim = c(-2,12)

# Fitting ebbinghaus data.

Now let's look at some real data

Ebbinghaus proposed the following equation to best described how savings change over time:

where the u1 and a1 are free parameters and log is taken with base 10. Q(t) are the savings at time interval t. Note that he calculated the best fitting parameters by hand, using **minutes** for the time interval. According to these calculations he suggested that u1= 1.84 and a1 = 1.25 were the best fitting parameters.

## Please load the ebbinghaus data ("Ebbinghaus.txt"), and see if you can fit the same model to his orignal data and see if you get the same parameters. Note the intervals in this file are in seconds.

Later the simpler exponental function has been proposed which is described by the equation

where Q(t) is savings at time t and μ1 and a1 are parameters.

first lets import some real data:

#importing data  
ebbing\_data <- read.delim("Ebbinghaus.txt",header= TRUE, sep = "\t") # open a file, will be stored as data.frame

### Q5 Check if the exponential indeed fits better, report the fit and the parameters. Show in a plot(s) how the best fitting models fit the data points.

# The fitness landscape

# DISPLAY EVALUATION OF FUNCTION

to illustrate this issue let's visualize the problem optim is trying to solve. Before we can do this we need to install two packages:

## ggplot is the plotting package of choice, can be used to make much nice graphs than simple plot()  
install.packages("ggplot")  
## rgl will be used to plot 3D images  
install.packages("rgl")  
##

In the following code you will see we call these libaries by library() function, check if that works to see if you installed the packages.

please run the code step by step to see if you understand what is going on. Basically we are running the function for a large grid of parameter combinations. Imagine that what is returned is the MSE or another measure of fitness.

himmelblau <- function(v){  
 x<-v[1]  
 y<-v[2]  
 h=((x^2+y-11)^2+(x\*1+y^2-7)^2)  
 return(h)  
}  
  
  
## making a grid of all possible combinatinos of X and Y  
xgrid <- seq(-5,5,.1)  
ygrid <- seq(-5,5,.1)  
grid <- expand.grid(xgrid, ygrid) # not limited to 2D can be longer list of lists  
  
## save the output of each x,y combination in the grid  
hlist<-c()  
for (i in 1:nrow(grid)){  
 h<-himmelblau(c(grid[i,1],grid[i,2]))  
 hlist[i]<-h  
}  
  
  
  
## 3D plotting   
hmatrix<-matrix(hlist,nrow=length(xgrid),ncol=length(xgrid)) # for 3dploting  
  
c = hmatrix  
c = cut(c, breaks=64)  
cols = rainbow(64)[as.numeric(c)]

Now let's plot this in 3D:

library(rgl)

## Warning: package 'rgl' was built under R version 3.2.5

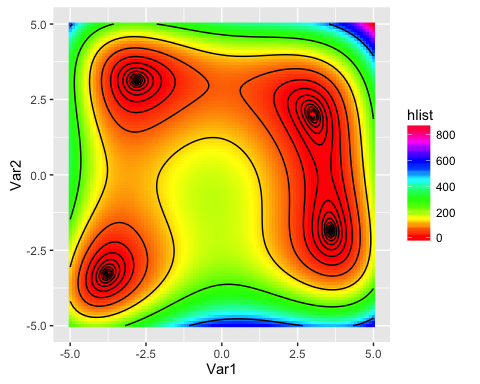
persp3d(xgrid, ygrid, hmatrix, col=cols)  
  
# combine the lists of function evaluations with original x and y inputs  
df<-cbind(grid,hlist)  
df$log<-log(df$hlist) # put function evals in log space for visualization

If all is well a window popped up with a 3D graph that you can turn around...

Now let's plot the same in 2D

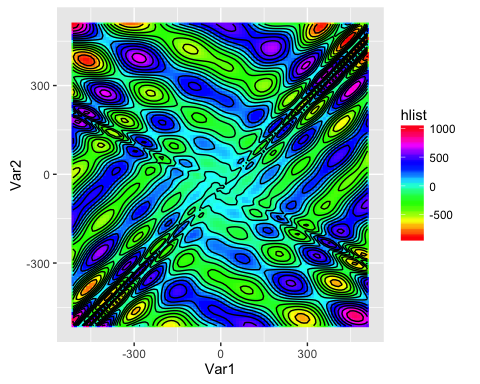
library(ggplot2)

ggplot() + geom\_raster(data=df, aes(x = Var1, y = Var2, z=hlist, fill = hlist, interpolate=T)) +  
 scale\_fill\_gradient(colours =rainbow(264))+  
 stat\_contour(data=df, aes(x = Var1, y = Var2, z= log), colour = "black",bins = 15)



Let's do one more:

eggholder<- function(v){  
 x<-v[1]  
 y<-v[2]  
 f=-(y+47)\*sin(sqrt(abs(x/2+(y+47))))-(x\*sin(sqrt(abs(x-(y+47)))))  
 return(f)  
}  
  
  
## making a grid of all possible combinatinos of X and Y  
xgrid <- seq(-512,512,10)  
ygrid <- seq(-512,512,10)  
grid <- expand.grid(xgrid, ygrid) # not limited to 2D can be longer list of lists  
  
## save the output of each x,y combination in the grid  
hlist<-c()  
for (i in 1:nrow(grid)){  
 h<-eggholder(c(grid[i,1],grid[i,2]))  
 hlist[i]<-h  
}  
  
  
  
## 3D plotting   
hmatrix<-matrix(hlist,nrow=length(xgrid),ncol=length(xgrid)) # for 3dploting  
  
c = hmatrix  
c = cut(c, breaks=64)  
cols = rainbow(64)[as.numeric(c)]  
  
library(rgl)  
persp3d(xgrid, ygrid, hmatrix, col=cols)  
  
# combine the lists of function evaluations with original x and y inputs  
df<-cbind(grid,hlist)  
#df$log<-log(df$hlist) # put function evals in log space for visualization  
  
library(ggplot2)  
  
ggplot() + geom\_raster(data=df, aes(x = Var1, y = Var2, z=hlist, fill = hlist, interpolate=T)) +  
 scale\_fill\_gradient(colours =rainbow(264))+  
 stat\_contour(data=df, aes(x = Var1, y = Var2, z= hlist), colour = "black",bins = 15)

 **BONUS POINTS Q6 Try to find the best parameters for the eggholder function!**

OK this is difficult, instead just report the results of optim() using 5 very different combinations of starting values  
Why do you think they are different, and what do you think we can do about it ?