Implicitly defined Population Attributes

Introduction

- In many cases, attributes of a population are defined **implicitly**, typically as the solution to some equation or set of equations.], for example

$$\widehat{\boldsymbol{\theta}} = \arg \min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \rho(\boldsymbol{\theta}; \mathcal{P})$$

where the possible values of $\boldsymbol{\theta}$ may be constrained to be in some set $\boldsymbol{\Theta}$.

- Maximizing a function is the same as minimizing its negation,

$$\max_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \rho(\boldsymbol{\theta}; \mathcal{P}) = \min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} -\rho(\boldsymbol{\theta}; \mathcal{P})$$

- The most common form for ρ is a sum of functions ρ evaluated at each unit $u \in \mathcal{P}$ such as:

Familiar Examples

Familiar examples for a **scalar valued** attribute $\theta \in \mathbb{R}$ include:

- **Least-squares**:

$$\rho(\theta; u) = (y_u - \theta)^2, \quad \text{yields} \quad \widehat{\theta} = \overline{y} \quad \text{as the solution to}$$

$$\widehat{\theta} = \arg\min_{\theta \in \mathbb{R}} \sum_{u \in \mathcal{P}} (y_u - \theta)^2.$$

- **Weighted least-squares**:

$$\rho(\theta; u) = w_u \, (y_u - \theta)^2, \quad \text{yields} \quad \widehat{\theta} = \frac{\sum_{u \in \mathcal{P}} w_u y_u}{\sum_{u \in \mathcal{P}} w_u} \quad \text{as the solution to}$$

$$\widehat{\theta} = \arg\min_{\theta \in \mathbb{R}} \sum_{u \in \mathcal{P}} w_u \, (y_u - \theta)^2.$$

Familiar Examples (cont'd)

- **Least absolute deviations**:

$$\rho(\theta; u) = |y_u - \theta|, \quad \text{yields} \quad \widehat{\theta} = Q_y(1/2) \quad \text{as the solution to}$$

$$\widehat{\theta} = \arg\min_{\theta \in \mathbb{R}} \sum_{u \in \mathcal{P}} |y_u - \theta|.$$

- **Quantiles**: for some $q \in (0, 1)$, the q^{th} quantile of $y_1, \ldots, y_N$ minimizes the vee function

$$\sum_{u \in \mathcal{P}} \rho_q(y_u - \theta)$$

where

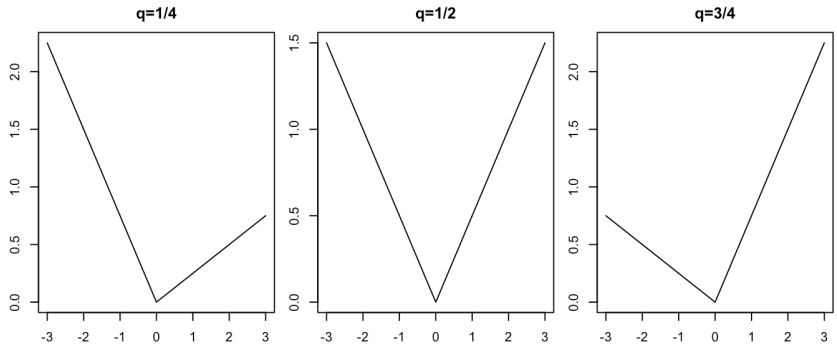
$$\rho_q(z) = \begin{cases} qz & z \geq 0 \\ -(1-q)z & z < 0 \end{cases}$$

is called a **vee function**. The quantile $\widehat{\theta} = Q_y(q)$ is the solution to the sum of vee funcitons below

$$Q_y(q) = \widehat{\theta} = \arg\min_{\theta \in \mathbb{R}} \sum_{u \in \mathcal{P}} \rho_q(y_u - \theta).$$

Vee function

```
rhoq <- function(z, q=1/2) {
  val = q*z
  val[z < 0] = -(1-q)*z[z<0]
  return(val)
}

z = seq(-3,3,.01)
par(mfrow=c(1,3), mar=2.5*c(1,1,1,0.1))
plot(z, rhoq(z,q=1/4), main="q=1/4", type='l')
plot(z, rhoq(z,q=2/4), main="q=1/2", type='l')
plot(z, rhoq(z,q=3/4), main="q=3/4", type='l')
```



An Example

Below is an example for quantiles based on sum of vee functions:

```
#This is so that all simulations give the same answer when you rerun them
set.seed(341)

#Simulated population
y.pop <- rnorm(1000,10,3)

# Writing the sum of vee functions to be minimized
vee.rhoq <- function(theta){
  temp <- sum(sapply(y.pop-theta , rhoq , q=0.3)) # q=0.3 will be passed on to rhoq function
  return(temp)
}

#minimizing the function (the $par part is the requiered quantile)
nlminb(0.5,vee.rhoq)
```

applies a function on members of a vector

check lapply and sapply in R

starting point

```
## $par
## [1] 8.344016
##
## $objective
## [1] 1071.576
##
## $convergence
## [1] 0
##
## $iterations
## [1] 6
##
## $evaluations
## function gradient
##       10        6
##
## $message
## [1] "both X-convergence and relative convergence (5)"
```

```
#Compare to R buit-in function - Note that we used q=0.3 above
quantile(y.pop,0.3)
```

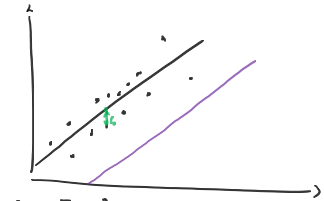
```
##      30%
## 8.343836
```

A vector valued attribute

- A familiar **vector valued** attribute $\boldsymbol{\theta} = (\alpha, \beta) \in \mathbb{R}^2$ is the pair of coefficients of a line being fitted to two variate values y and x as in

$$y_u = \alpha + \beta x_u + r_u$$

for all $u \in \mathcal{P}$.



- *Here $r_u = y_u - \alpha - \beta x_u$ is called the residual*
- *The residual is the signed vertical distance between the point (x_u, y_u) and the line given by the values of $\boldsymbol{\theta} = (\alpha, \beta)$.*

- **Note**: for model interpretation, we often fit the line

$$y_u = \alpha + \beta\,(x_u - c) + r_u$$

- *Choose c such that it is meaningful, e.g. in the middle, i.e. $c = \bar{x}$).*
- *Then the coefficient α has the interpretation as the value of the line when $x_u = c$;*
- *If the choice is outside the range of the x values such as $c = 0$, it is less likely to have a meaningful interpretation.*
- *Whatever the value of c chosen, only the interpretation and value of the intercept α changes.*

Least Squares Regression

- When $\rho(\boldsymbol{\theta}; u) = r_u^2$, the coefficients are determined as

$$\widehat{\boldsymbol{\theta}} = \arg \min_{\boldsymbol{\theta} \in \mathbb{R}^2} \sum_{u \in \mathcal{P}} (y_u - \boldsymbol{\theta})^2$$

  or equivalently with $\widehat{\boldsymbol{\theta}} = (\widehat{\alpha}, \widehat{\beta})^\mathsf{T}$ as

$$(\widehat{\alpha}, \widehat{\beta}) = \arg \min_{(\alpha,\beta) \in \mathbb{R}^2} \sum_{u \in \mathcal{P}} (y_u - \alpha - \beta(x - c))^2$$

  - *the resulting fitted line is called the* ***least-squares*** *line,*

$$y = \widehat{\alpha} + \widehat{\beta}(x - c)$$

  - *and the fitted values are*

$$\widehat{y}_u = \widehat{\alpha} + \widehat{\beta}(x_u - c)$$

- **Exercise**: find closed form expressions for $(\widehat{\alpha}, \widehat{\beta})$.

Least Squares Regression (cont'd)

- Similarly, we could have a weighted least squares line, or a least absolute deviations line. Minimizing a function $\rho(y_u - \alpha - \beta(x - c))$ would produce a fitted line

1. Weighted Least Squares:

$$(\widehat{\alpha}, \widehat{\beta}) = \arg \min_{(\alpha,\beta) \in \mathbb{R}^2} \sum_{u \in \mathcal{P}} w_u \left[y_u - \alpha - \beta(x - c)\right]^2$$

=> mean

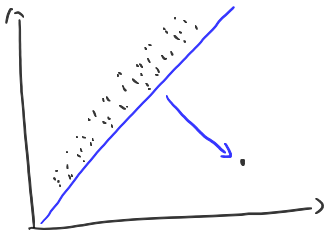
2. Least Absolute Deviations Line:

$$(\widehat{\alpha}, \widehat{\beta}) = \arg \min_{(\alpha,\beta) \in \mathbb{R}^2} \sum_{u \in \mathcal{P}} \left|y_u - \alpha - \beta(x - c)\right|$$

cannot differentiate

=> median

=> if $\exists$ outliers



=> $\sum_{u=1}^{N} w_u r_u^2$

Robustness

- Investigating robustness is to investigate how sensitive attributes are to influential points (similar to the delta plot idea).
- It will show how to design atttributes which are resistent to the effect of influential points a.k.a. outliers.

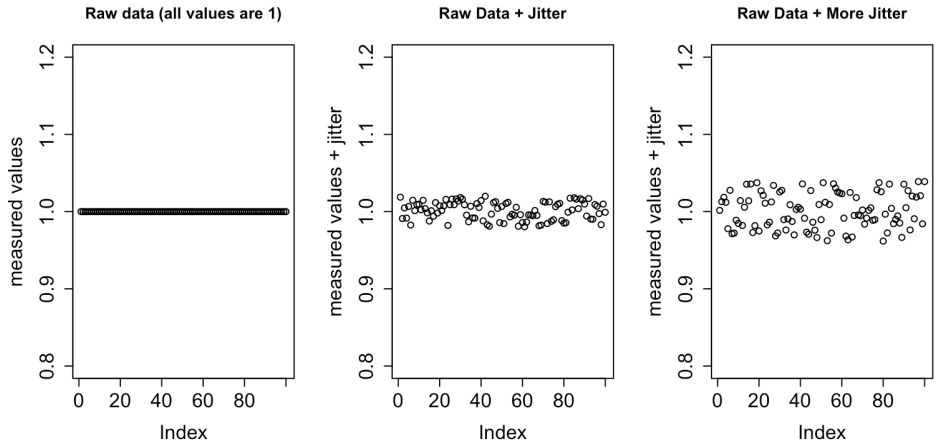
Jittering Data: Adding Random Noise

- Add a small amount of random noise to the data to see how it affects the analysis/attributes values.
- If the value of the attribute does not change significantly when jitter is added, we say that the attribute is **robust to noise injection**.
- Another usage of jitter is when the data is discrete, so lots of values are ``the same'' in the dataset. Jitter speerates these points slightly (provided it makes sense to do so).

A Simulation Example

```
y.example <- rep(1,100)
par(mfrow=c(1,3)) #dividing the panel into 1row and 2 columns for 2 plots
plot(y.example,main='Raw data (all values are 1)',ylim=c(0.8,1.2)
     ,ylab='measured values',cex.lab=1.5,cex.axis=1.5)
plot(jitter(y.example), main='Raw Data + Jitter',ylim=c(0.8,1.2)
     ,ylab='measured values + jitter',cex.lab=1.5,cex.axis=1.5)
plot(jitter(y.example,factor=2), main='Raw Data + More Jitter',ylim=c(0.8,1.2)
     ,ylab='measured values + jitter',cex.lab=1.5,cex.axis=1.5)
```

default = 1



Check the help documentation of the function *jitter* by typing *?jitter* or *help(jitter)* in R.

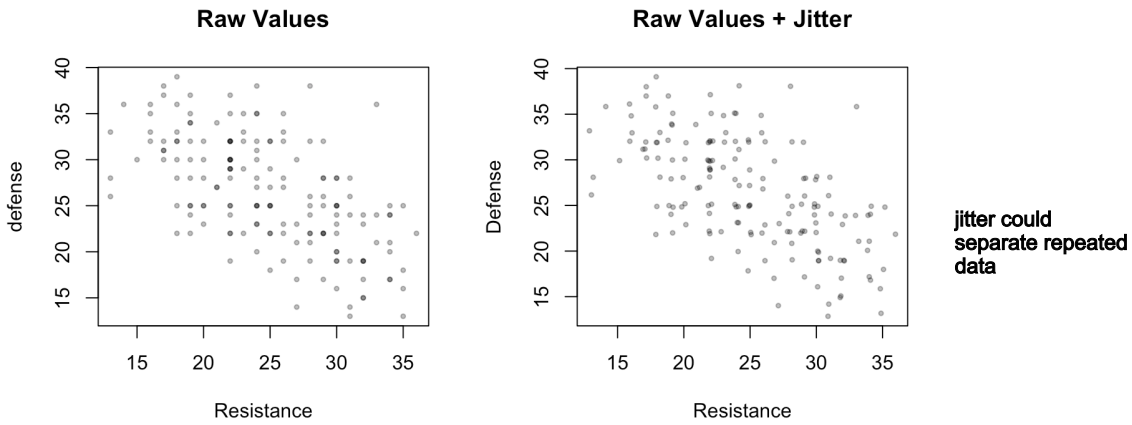
Fire Emblem Heroes Example

- `Resistance` ability to absorb magical attack (use as X)
- `Defense` ability to absorb physical attacks (use as Y)

```
feh = read.csv("../Data/feh.csv", header=TRUE)
feh[1:5,]
```

```
##       Name       Type     Move HP ATK SPD DEF RES Total
## 1    Abel Blue Lance  Cavalry 39  33  32  25  25   154
## 2 Alfonse  Red Sword Infantry 43  35  25  32  22   157
## 3     Alm  Red Sword Infantry 45  33  30  28  22   158
## 4  Amelia  Green Axe  Armored 47  34  34  35  23   173
## 5    Anna  Green Axe Infantry 41  29  38  22  28   158
```

```
par(mfrow=c(1,2))
plot(feh$RES, feh$DEF, main="Raw Values", pch = 19, cex=0.5,
     col=adjustcolor("black", alpha = 0.3), xlab="Resistance", ylab="defense" )

plot( jitter(feh$RES), jitter(feh$DEF), main="Raw Values + Jitter",  pch = 19, cex=0.5,
      col=adjustcolor("black", alpha = 0.3),xlab="Resistance",  ylab="Defense", type="p" )
```



jitter could separate repeated data

Fire Emblem Heroes Example (Cont'd)

- The average for defense is $\bar{y} = 26.4$ and resistance is $\bar{x} = 24.9$
- This relationship can be summarized as

$$Defense = 42.5 - 0.65 \times Resistance + error$$

or equivalently as

theoretically the same, but if jittered, there will be 2 lines

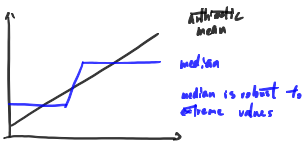
$$Defense = 26.4 - 0.65 \times (Resistance - 24.9) + error$$

- *both equations (centred and non-centred ones above) yield the same line and fitted values.*

1) robustness of optimized method to jitter

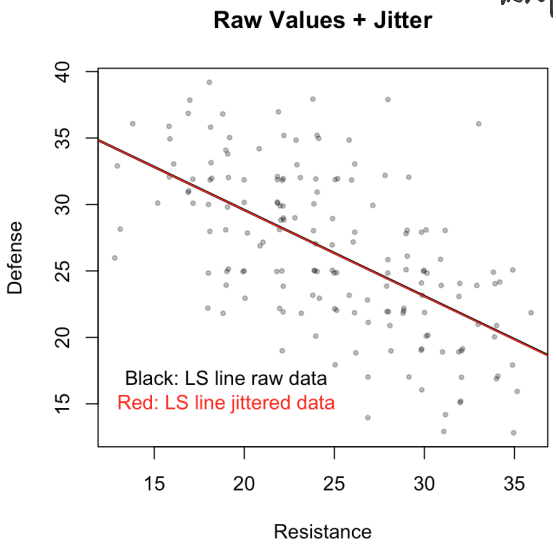
2) attributes robustness to extreme values

e.g.



model = lm(y ~ x) ← $y = \hat{\alpha} + \hat{\beta}x$

abline(model)



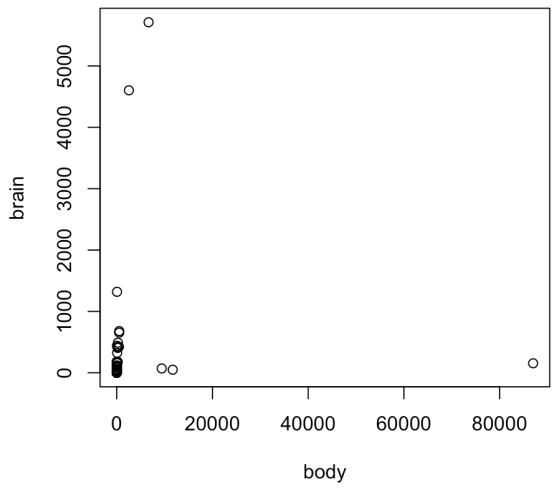
Brain versus Body weight

- This is a built-in data frame in R (robustbase library) with average brain and body weights for 65 species of land mammals and three others.
  - *body represents body weight in kg*
  - *brain represents brain weight in grams*

```
data(Animals2, package = "robustbase")
Animals2[1:5,]
```

```
##                   body brain
## Mountain beaver   1.35   8.1
## Cow             465.00 423.0
## Grey wolf        36.33 119.5
## Goat             27.66 115.0
## Guinea pig        1.04   5.5
```

```
plot(Animals2)
```



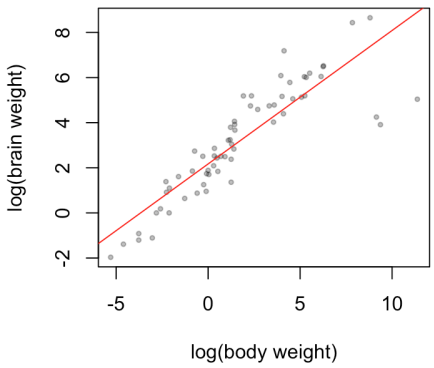
Transformated Data

```
### This requires that the loon package be installed.
### install.package("loon") will install the package from CRAN
### (requires R >= 3.4)
###
library(loon)
###
power <- function(x, y, linkingGroup="linkingGroup", from=-5, to=5, ...) {
  ## Create histograms
  histX <- l_hist(x, linkingGroup = linkingGroup, yshows="density")
  histY <- l_hist(y, linkingGroup = linkingGroup, yshows="density", swapAxes = FALSE)

  ## Now we build an interactive scatterplot
  ## with sliders for power transformations
  ## on each of x and y
  tt <- tktoplevel()
  tktitle(tt) <- "Power Transformation"
  p <- l_plot(x=x, y=y, parent=tt, linkingGroup=linkingGroup, ...)
  ## Alpha values
  alpha_x <- tclVar('1')
  alpha_y <- tclVar('1')
  ## Sliders to change the alphas
  sx <- tkscale(tt, orient='horizontal', variable=alpha_x, from=from, to=to, resolution=0.05)
  sy <- tkscale(tt, orient='vertical', variable=alpha_y, from=to, to=from, resolution=0.05)
  ## Laying out the pieces in one window
  tkgrid(sy, row=0, column=0, sticky="ns")
  tkgrid(p, row=0, column=1, sticky="nswe")
  tkgrid(sx, row=1, column=1, sticky="we")
  tkgrid.columnconfigure(tt, 1, weight=1)
  tkgrid.rowconfigure(tt, 0, weight=1)

  ## This function redraws the plots with the alphas
  ## from the slider values whenever it is called.
  ##
  update <- function(...) {
    ### get transformed x and y
    transformedX <- powerfun(x, as.numeric(tclvalue(alpha_x)))
    transformedY <- powerfun(y, as.numeric(tclvalue(alpha_y)))

    ## First the scatterplot
    l_configure(p, x = transformedX,  y = transformedY)
    l_scaleto_world(p)
    ## Now the histograms
    l_configure(histX, x = transformedX)
    l_scaleto_world(histX)
    l_configure(histY, x = transformedY)
    l_scaleto_world(histY)
  }
  ## Set the function update to be called
  ## whenever the slider values are changed
  tkconfigure(sx, command=update)
  tkconfigure(sy, command=update)
  ## Return the scatterplot if assigned
  invisible(p)
}

###
### Here's an example using the Animals2 data set
### from the MASS packages
library(robustbase)
data("Animals2")
p <- with(Animals2, power(body, brain,
                xlabel="body weight",
                ylabel="brain weight",
                title="Brain and Body Weights for 65 Species of Land Mammals",
                linkingGroup = "Animals2",
                itemLabel=rownames(Animals2),
                showItemLabels=TRUE)
```

)

log(Brain) vs. log(Body) Transformation

```
data(Animals2, package = "robustbase")
plot(log(Animals2$body), log(Animals2$brain), pch = 19, cex=0.5,
     col=adjustcolor("black", alpha = 0.3),xlab='log(body weight)',ylab='log(brain weight)')
abline( lm(log(Animals2$brain) ~ log(Animals2$body)), col=2 )
```



- Does the LS line explains the relationship between brain and body weight?

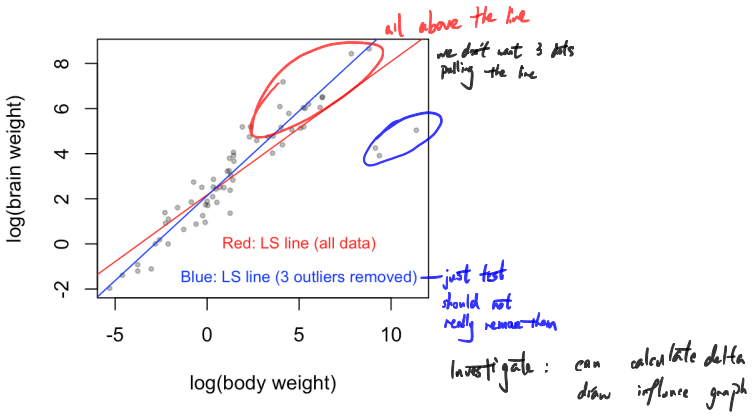
1) attach (testdata)
2) testdata $y

testdata Dataframe

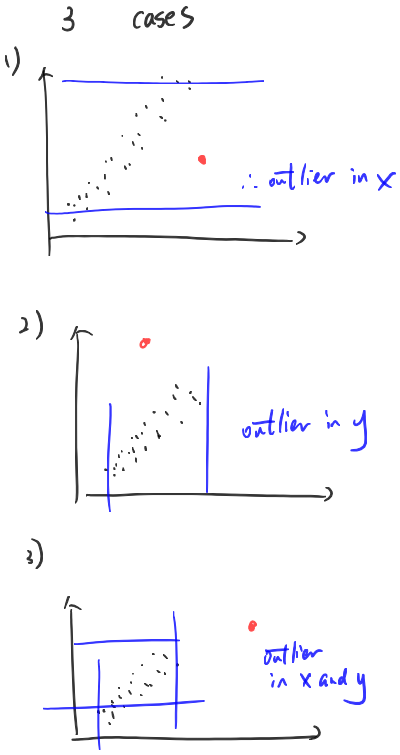
x y z

log(Brain) versus log(Body) weight

- `Red line` LS regression using all the data
- `Blue line` LS regression while removing those three `outlier` points



- It seems we need an attribute which down weights outliers.
- Let us investigate these three points



The "Outliers"

```
indx = which(log(Animals2$body)>9)
indx
```

```
## [1]  6 16 26
```

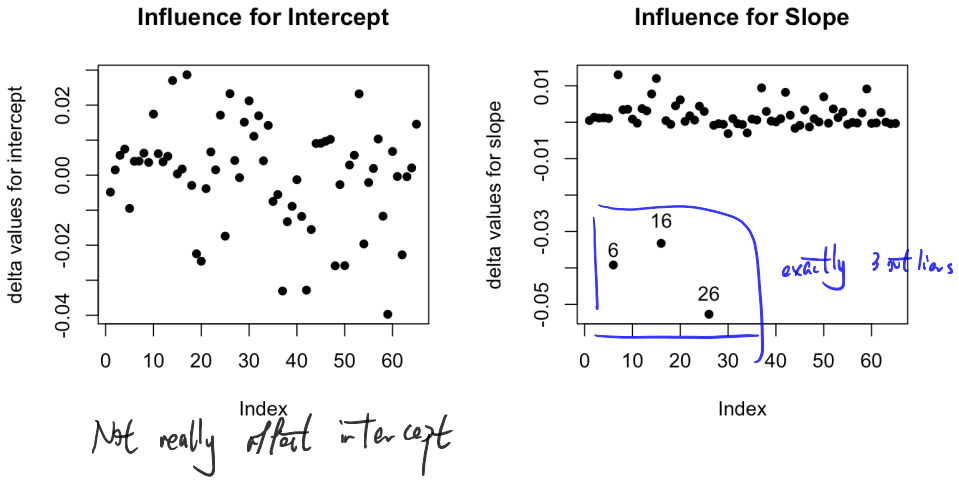
```
log(Animals2[indx,])
```

```
##                    body    brain
## Dipliodocus    9.367344 3.912023
## Triceratops    9.148465 4.248495
## Brachiosaurus 11.373663 5.040194
```

- All three are dinosaurs! The rest of the dataset are land mammals.
- Relative to other lang mammals, dinosaurs have significantly larger bodies relative to their brain weights!

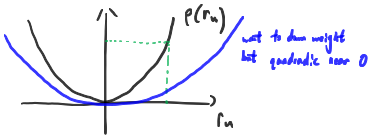
Influence (Delta Values)

- Let us look at the influence of the points (delta values) on $\widehat{\alpha}$ and $\widehat{\beta}$



Conclusion: what is your conclusion based on the two plots above? **Exercise:** Generate the two plots shown above, i.e. delta values for α: LS line intercept and β: LS line slope.

Regression



- For least squares regression, the attribute of interest is

$$(\hat{\alpha}, \hat{\beta}) = \arg \min_{(\alpha,\beta) \in \mathbb{R}^2} \sum_{u \in \mathcal{P}} (y_u - \alpha - \cancel{\beta[x - c]})^2$$
$$= \arg \min_{(\alpha,\beta) \in \mathbb{R}^2} \sum_{u \in \mathcal{P}} \rho\left(y_u - \alpha - \beta[x - c]\right)$$

  where $\rho(r) = r^2$ (or equivalently $\rho(r) = r^2/2$, in some texts).

- One solution to the outlier problem noticed above is to change the $\rho(r)$ function:
  - *Ideally, $\rho(r)$ should be quadratic near 0*
  - *and give lower weight than LS for r far from 0.*

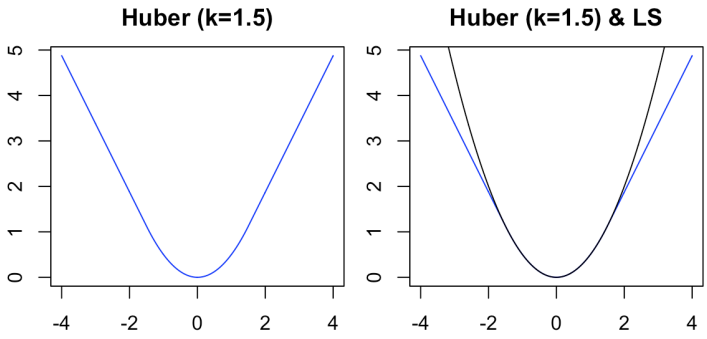
Huber Function

robust to outliers

- Mix of the quadratic and absolute value functions.

$$\rho_k(r) = \begin{cases} \frac{1}{2}r^2 & \text{for } |r| \le k, \\ k\,|r| - \frac{1}{2}k^2, & \text{otherwise.} \end{cases}$$

penalize outliers

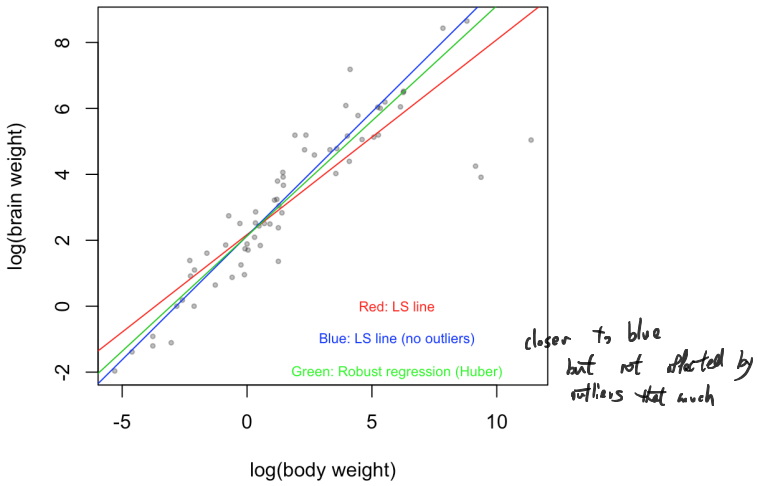


- An attribute based on this adjustment will be affected by the scale. So we might let $k = cS$ where
  - *S is a (possibility robust) measure of scale.* usually s.d.
- $k \approx 1.345\,S$ satisfies a theoretical balance between **efficiency** and **resistance to outliers**.

i.e. convergent to some value

Example - Animals

- log(Brain) versus log(Body) weight:
- The blue line (outliers removed) and the green line (robust regression line) are close to each other.



- But how to find $(\widehat{\alpha}, \widehat{\beta})$ for the green line (robust estimates)?

Robust Estimation

- Our attributes of interest is

whether or not centralize does not matter

$$(\hat{\alpha}, \hat{\beta}) = \arg \min_{(\alpha,\beta) \in \mathbb{R}^2} \sum_{u \in \mathcal{P}} \rho\left(y_u - \alpha - \beta[x - c]\right)$$

  where one choice of ρ in robust regression is the Huber function.

does not have a closed form of derivative

- Optimization methods to be considered:
  - *Gradient descent,*
  - *Newton-Raphson,*
  - *Iteratively reweighted least-squares.*

all robust to jitter

all depend on derivative of ρ

not really ideal

- The algorithms above to be discussed will be viewed in a general framework to handle implicitly defined attributes defined as

$$\widehat{\boldsymbol{\theta}} = \arg \min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \rho(\boldsymbol{\theta}; \mathcal{P})$$

Robustness to outliers:
- use Huber function

Robustness to jitter:
- choose optimization method carefully

Gradient descent

- Goal to construct an **iterative procedure** which produces a sequence of **iterates**

assume the function is differentiable

$$\widehat{\boldsymbol{\theta}}_0,\ \widehat{\boldsymbol{\theta}}_1,\ \widehat{\boldsymbol{\theta}}_2,\ \ \ldots,\ \widehat{\boldsymbol{\theta}}_i,\ \widehat{\boldsymbol{\theta}}_{i+1},\ \ \ldots$$

such that the limit of this sequence as $i \to \infty$ converges to the solution $\widehat{\boldsymbol{\theta}}$.

  - *Ideally, each iterate is closer to the solution than is the one before it.*

- Suppose we want to minimize a function $\rho(\boldsymbol{\theta})$ and we are at some point $\widehat{\boldsymbol{\theta}}_i$ on that surface.

  - *We can improve on this estimate by moving away in a direction which is lower on the surface than is* $\rho(\widehat{\boldsymbol{\theta}}_i)$.
  - *What is that direction though?*

Search Direction

- If $\rho(\boldsymbol{\theta})$ is a **differentiable** function of $\boldsymbol{\theta}$ then we calculate the **gradient**

$$\mathbf{g}_i = \mathbf{g}(\widehat{\boldsymbol{\theta}}_i) = \left.\frac{\partial\, \rho(\boldsymbol{\theta})}{\partial\, \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}_i}.$$

- Normalizing $\mathbf{g}_i$ to unit length provides the **direction** in which $\rho(\boldsymbol{\theta})$ rises, or ascends, fastest.

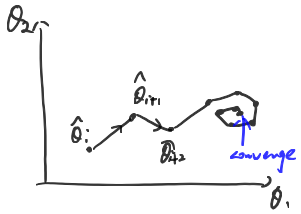
$$\mathbf{g}_i \leftarrow \frac{\mathbf{g}_i}{||\mathbf{g}_i||}$$

  - *The gradient vector $\mathbf{g}_i$ is now the direction of steepest **ascent***
  - *$-\mathbf{g}_i$ is, therefore, in the direction of **steepest descent** (recall that we will focus on minimization problems).*

- Obtain a new estimate or the next iterate $\widehat{\boldsymbol{\theta}}_i$ by
  - *taking downhill step of length $\lambda > 0$ in the direction of $-\mathbf{g}_i$.*

$$\widehat{\boldsymbol{\theta}}_{i+1} = \widehat{\boldsymbol{\theta}}_i - \lambda\, \mathbf{g}_i.$$

Line Search



- The next iterate is based on

$$\widehat{\boldsymbol{\theta}}_{i+1} = \widehat{\boldsymbol{\theta}}_i - \lambda\, \mathbf{g}_i.$$

- We find the step size λ for each i to be that value which minimizes

$$\rho(\widehat{\boldsymbol{\theta}}_i - \lambda\, \mathbf{g}_i)$$

as a function of λ (i.e. for fixed $\widehat{\boldsymbol{\theta}}_i$).

- *Finding the value of λ in this way is called a **line search**.*

Gradient Descent Algorithm

1. **Initialize** ; $i \leftarrow 0$; determine an initial estimate $\widehat{\boldsymbol{\theta}}_0$

2. LOOP:

   1. ***Gradient***:

$$\mathbf{g}_i = \frac{\partial\, \rho(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}_i}$$

   2. *Gradient* ***direction***:

$$\mathbf{g}_i \leftarrow \frac{\mathbf{g}_i}{||\mathbf{g}_i||}$$

   3. ***Line search***: *Find the value* $\widehat{\lambda}$ over λ not θ

$$\widehat{\lambda} = \arg\min_{\lambda>0} \rho(\widehat{\boldsymbol{\theta}}_i - \lambda\, \mathbf{g}_i)$$

   4. ***Update*** *the iterate:*

$$\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}_i - \widehat{\lambda}\, \mathbf{g}_i$$

   5. ***Converged?***

      if *the iterates are not changing, then* ***Return***

      else $i \leftarrow i+1$ *and repeat LOOP.*

3. **Return**: $\widehat{\theta} = \widehat{\boldsymbol{\theta}}_i$

e.g $\theta_i = (5,7)$
$\theta_{i+1} = (5.6, 6.8)$
if $\sqrt{(5.6-5)^2+(6.8-7)^2} < \varepsilon$ predetermined
=> converge

Note: Don't know converge to local minimum or global minimum
=> initial value is sensitive
=> Redo problem with very different initial θ

R code - Gradient Descent

```
gradientDescent <- function(theta = 0,
      rhoFn, gradientFn, lineSearchFn, testConvergenceFn,
      maxIterations = 100,
      tolerance = 1E-6, relative = FALSE,
      lambdaStepsize = 0.01, lambdaMax = 0.5 ) {

  converged <- FALSE
  i <- 0

  while (!converged & i <= maxIterations) {
    g <- gradientFn(theta) ## gradient
    glength <-  sqrt(sum(g^2)) ## gradient direction
    if (glength > 0) g <- g /glength

    lambda <- lineSearchFn(theta, rhoFn, g,
                lambdaStepsize = lambdaStepsize, lambdaMax = lambdaMax)

    thetaNew <- theta - lambda * g
    converged <- testConvergenceFn(thetaNew, theta,
                  tolerance = tolerance,
                  relative = relative)
    theta <- thetaNew
    i <- i + 1
  }

  ## Return last value and whether converged or not
  list(theta = theta, converged = converged, iteration = i, fnValue = rhoFn(theta)
       )
}
```

- The functions required in the code above yet to be defined.
- These will need to be implemented for each particular function $\rho(\cdots)$ we wish to minimize.

R code - Line Search & Convergence

```
### line searching could be done as a simple grid search
gridLineSearch <- function(theta, rhoFn, g,
                       lambdaStepsize = 0.01,
                       lambdaMax = 1) {
  ## grid of lambda values to search
  lambdas <- seq(from = 0, by = lambdaStepsize,  to = lambdaMax)

  ## line search
  rhoVals <- Map(function(lambda) {rhoFn(theta - lambda * g)}, lambdas)
  ## Return the lambda that gave the minimum
  lambdas[which.min(rhoVals)]
}

### Where testCovergence might be (relative or absolute)
testConvergence <- function(thetaNew, thetaOld, tolerance = 1E-10, relative=FALSE) {
   sum(abs(thetaNew - thetaOld)) < if (relative) tolerance * sum(abs(thetaOld)) else tolerance
}
```

Simple Example - Quadratic Function

Suppose for example that we were finding that θ which minimized

$$\rho(\theta) = 2\theta^2 - 5\theta + 3.$$

Then

$$g = 4\theta - 5$$

and we only need to write the corresponding R functions

```
rho <- function(theta) { 2 * theta^2 - 5 * theta +3}
g <- function(theta) {4 * theta - 5}
```

Quadratic Function (cont'd)

and then perform Gradient descent to find the value $\widehat{\theta}$ with

```
gradientDescent(rhoFn = rho, gradientFn = g,
                lineSearchFn = gridLineSearch,
                testConvergenceFn = testConvergence)
```

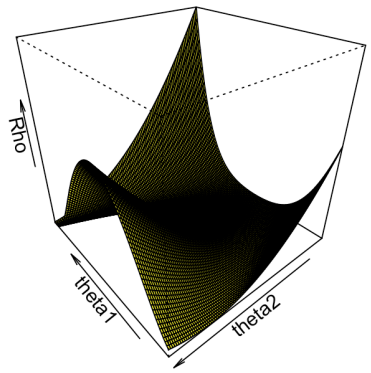
```
## $theta
## [1] 1.25
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 4
##
## $fnValue
## [1] -0.125
```

Another Example - Rosenbrock function

- General form: $f(x, y) = (a - x)^2 + b(y - x^2)^2$
- We want to find $\boldsymbol{\theta}$ which minimizes the following Rosenbrock function

$$\rho(\boldsymbol{\theta}) = (1 - \theta_1)^2 + 100\left(\theta_2 - \theta_1^2\right)^2$$

```
rho <- function(theta1,theta2) {  (1-theta1)^2 + 100*(theta2-theta1^2)^2 }
theta1 <- seq(-1.5,1.8,length=100)
theta2 <- seq(-0.5,3,length=100)
Rho <- outer(theta1,theta2,"rho")
persp(theta1,theta2,Rho, theta = 230, phi = 30,col='yellow')
```



Rosenbrock function (cont'd)

Rosenbrock function is a non-convex function, usually used in performance test problem for optimization algorithms. The gradient is

$$\mathbf{g} = \left[400\theta_1^3 - 400\theta_1\theta_2 + 2\theta_1 - 2 \ , \ -200\theta_1^2 + 200\theta_2\right]$$

and we write the corresponding R functions

```
rho <- function(theta) {  (1-theta[1])^2 + 100*(theta[2]-theta[1]^2)^2 }
g <- function(theta) {
  c( 400*theta[1]^3 -400*theta[1]*theta[2] +2*theta[1]-2,
     -200*theta[1]^2+200*theta[2] ) }
```

and then perform Gradient descent to find the value $\widehat{\theta}$ with

```
gradientDescent(theta= c(0,0), rhoFn = rho, gradientFn = g,
                lineSearchFn = gridLineSearch,
                testConvergenceFn = testConvergence, maxIterations=1000)
```

```
## $theta
## [1] 0.7987955 0.6278095
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 419
##
## $fnValue
## [1] 0.05101962
```

Gradient Descent for Least Squares

We have $\boldsymbol{\theta} = (\alpha, \beta)^{\mathsf{T}}$ and using $c = \bar{x}$ we have

$$\rho(\alpha, \beta) = \sum_{u \in \mathcal{P}} (y_u - \alpha - \beta(x_u - \bar{x}))^2 = \sum_{u \in \mathcal{P}} r_u^2$$

so that

$$\mathbf{g}_i = \begin{pmatrix} \sum_{u \in \mathcal{P}} -2(y_u - \alpha - \beta(x_u - \bar{x})) \\ \\ \sum_{u \in \mathcal{P}} -2(y_u - \alpha - \beta(x_u - \bar{x}))(x_u - \bar{x}) \end{pmatrix}_{\alpha = \hat{\alpha}_i \,;\, \beta = \hat{\beta}_i}$$

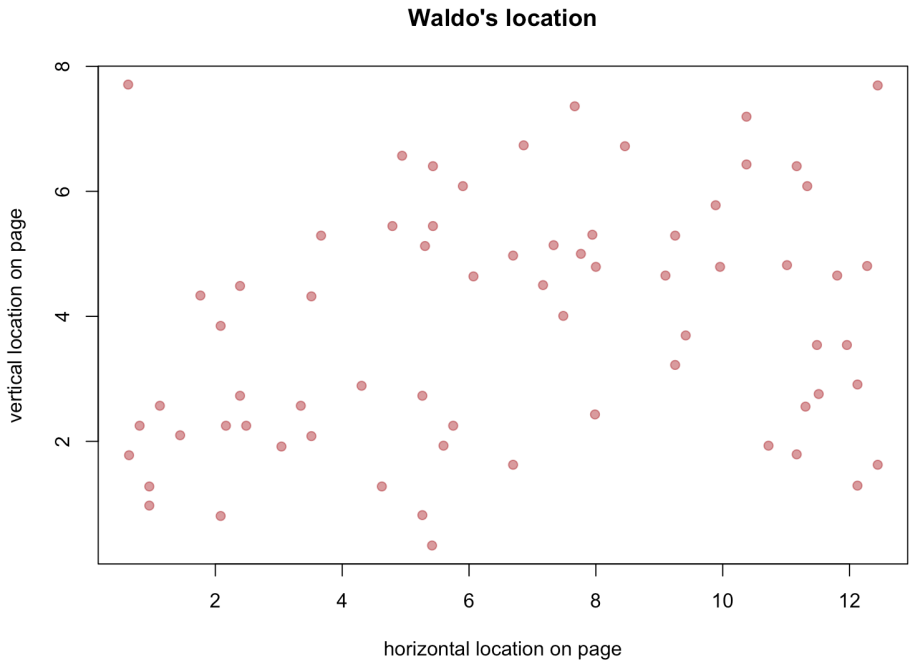
$$= -2 \begin{pmatrix} N\,(\bar{y} - \hat{\alpha}_i) \\ \\ \sum_{u \in \mathcal{P}} (x_u - \bar{x})\, y_u \;-\; \hat{\beta}_i \sum_{u \in \mathcal{P}} (x_u - \bar{x})^2 \end{pmatrix}.$$

For a least-squares line, the ρ function depends on the values of the variates x and y.

$\frac{\partial \rho}{\partial \alpha} = -2 \sum_{u \in \mathcal{P}} (y_u - \alpha - \beta(x - \bar{x})) = -2 \left[\sum_{u \in \mathcal{P}} y_u - \sum_{u \in \mathcal{P}} \alpha - \beta \sum_{u \in \mathcal{P}} (x_i - \bar{x}) \right] = -2 \left[N\bar{y} - N\alpha \right] = -2N(\bar{y} - \alpha)$ ($\sum_{u \in \mathcal{P}} (x_i - \bar{x}) = 0$)

$\frac{\partial \rho}{\partial \beta} = -2 \sum_{u \in \mathcal{P}} (x - \bar{x})(y_u - \alpha - \beta(x - \bar{x}))$

Example - LS for Where's Waldo?



LS - `rho` and `gradient`

For this problem, we need to write the corresponding `R` functions `rho` and `gradient`.

```
rho <- function(theta) {
  alpha <- theta[1]
  beta <- theta[2]
  ## Note that we are accessing waldo from the globalEnv
  sum( (waldo$Y - alpha - beta * (waldo$X - mean(waldo$X)) )^2 )
}

gradient <- function(theta) {
  alpha <- theta[1]
  beta <- theta[2]
  ## Note that we are accessing waldo from the globalEnv
  x <- waldo$X
  y <- waldo$Y
  N <- length(x)
  xbar <- mean(x)
  ybar <- mean(y)
  g <-  -2 * c(N * (ybar - alpha),
               sum((x - xbar) * y) - beta * sum((x - xbar)^2))
  # Return g
  g
}
```

Comment: Global Variables

- Note that in the R codes above *global* variables like `waldo$X` and `waldo$Y` appear inside the functions `rho(...)` and `gradient(...)`.
- This will work provided that `waldo` is available in the *global environment when `rho` and `gradient` are called*. This may not be reliable and so should be avoided in general.

Avoiding the global environment

- Pass the needed variables (here `x` and `y`) to the function as arguments.
  - *write functions which will return the appropriate functions to avoid cluttering*
- Functions containing their own data environment are called **closures**.
  - *Every function has a local environment where variables may be defined; this is the closure of the function.*
  - *Functions also have access to the environment in which they were created (that's why functions can access values in the global environment).*
- We use this property to have one function define another function within itself and return it.
  - *The interior function is enclosed within the function that created it, hence the word closure.*
  - *The returned function (or closure) has access to any variables defined within the enclosing function that defined it.*
- Encapsulation of data within a function is an important and powerful construct.

A function that returns a function

For example, we can write a function that will take `x` and `y` and return an appropriate `rho` (or `gradient`) function as follows:

```
createLeastSquaresRho <- function(x,y) {
  ## local variable
  xbar <- mean(x)
  ## Return this function
  function(theta) {
    alpha <- theta[1]
    beta <- theta[2]
    sum( (y - alpha - beta * (x - xbar) )^2 )
  }
}

### We now get the rho function for the waldo data
rho <- createLeastSquaresRho(waldo$X, waldo$Y)

### Similarly for the gradient function
createLeastSquaresGradient <- function(x,y) {
  ## local variables
  xbar <- mean(x)
  ybar <- mean(y)
  N <- length(x)
  function(theta) {
    alpha <- theta[1]
    beta <- theta[2]
    -2 * c(N * (ybar - alpha),
           sum((x - xbar) * y) - beta * sum((x - xbar)^2)
           )
  }
}
gradient <- createLeastSquaresGradient(waldo$X, waldo$Y)
```

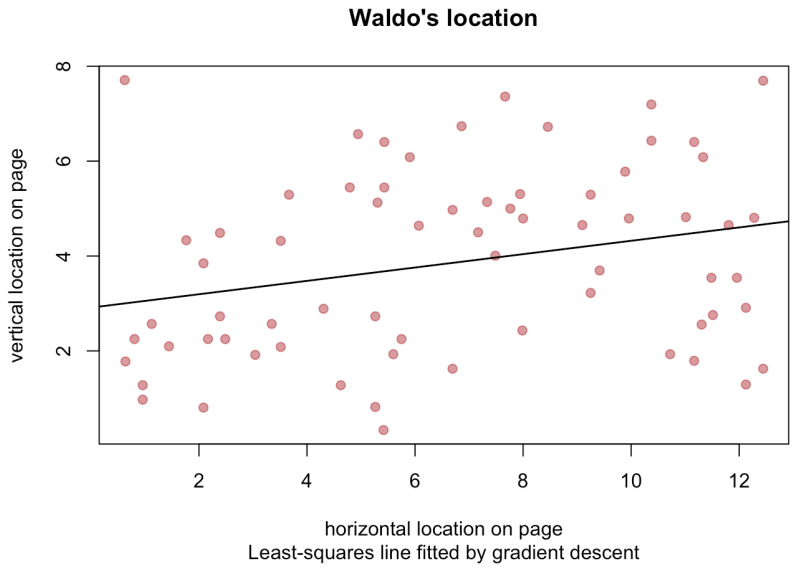
Back to `Where's Waldo` Example

The functions `rho` and `gradient` can now be passed as arguments to `gradientDescent(...)` to find the value $\widehat{\theta}$.

```
result <- gradientDescent(theta = c(0,0),
                          rhoFn = rho, gradientFn = gradient,
                          lineSearchFn = gridLineSearch,
                          testConvergenceFn = testConvergence)

### Print the results (and round everything up to 3 deecimal points)
Map(function(x){if (is.numeric(x)) round(x,3) else x}, result)
```

```
## $theta
## [1] 3.857 0.141
##     estimated y coordinate   estimated x coordinate
## $converged
## [1] TRUE
##
## $iteration
## [1] 29
##
## $fnValue
## [1] 233.774
```

Where's Waldo - LS fitted line



- LS line: guide the eye across any two page spread in a *Where's Waldo?* book to help find him.
- More complex summary than a simple line might be better.

Example - Robust Regression

- Attribute of interest: $\rho(\boldsymbol{\theta}) = \rho(\alpha, \beta) = \sum_{u \in \mathcal{P}} \rho_k \left(y_u - \alpha - \beta(x_u - \bar{x}) \right) = \sum_{u \in \mathcal{P}} \rho_k(r_u)$

  where $r_u = y_u - \alpha - \beta(x_u - \bar{x})$

$$\text{where} \qquad \rho_k(r) = \begin{cases} \frac{1}{2} r^2 & \text{for } |r| \le k, \\ k\ |r| - \frac{1}{2} k^2, & \text{otherwise.} \end{cases}$$

- The gradient can be decomposed as

$$\mathbf{g}_i = \left. \frac{\partial\, \rho(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right|_{\boldsymbol{\theta} = \hat{\boldsymbol{\theta}}_i} = \sum_{u \in \mathcal{P}} \frac{\partial \rho(r_u)}{\partial r_u} \times \left. \frac{\partial\, r_u}{\partial \boldsymbol{\theta}} \right|_{\boldsymbol{\theta} = \hat{\boldsymbol{\theta}}_i}$$

$$\text{where} \qquad \rho'_k(r) = \begin{cases} r & \text{for } |r| \le k, \\ \text{sign}(r)\ k, & \text{otherwise.} \end{cases}$$

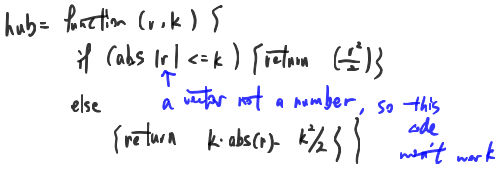
$$\frac{\partial\, r_u}{\partial \boldsymbol{\theta}} = \begin{bmatrix} \frac{\partial\, r_u}{\partial \boldsymbol{\alpha}} \\ \frac{\partial\, r_u}{\partial \boldsymbol{\beta}} \end{bmatrix} = -1 \times \begin{bmatrix} 1 \\ x_u - \bar{x} \end{bmatrix}$$

- Now, if we let

$$\mathbf{z}_u = \begin{bmatrix} 1 \\ x_u - \bar{x} \end{bmatrix}$$

  - *then the gradient can be written as*

$$\mathbf{g}_i = -1 \times \sum_{u \in \mathcal{P}} \rho'_k(r_u)\ \mathbf{z}_u$$



Example - Robust Regression - R code

```r
huber.fn <- function(r, k = 1.5) {
  val = r^2/2
  subr = abs(r) > k
  val[ subr ] = k*(abs(r[subr]) - k/2)
  return(val)
}

createRobustHuberRho <- function(x,y, kval=1.5) {
  ## local variable
  xbar <- mean(x)
  ## Return this function
  function(theta) {
    alpha <- theta[1]
    beta <- theta[2]
    sum( huber.fn(y - alpha - beta * (x - xbar),  k = kval ) )
  }
}


huber.fn1 <- function(r, k = 1.5) {
  val = r
  subr = abs(r) > k
  val[ subr ] = k*sign(r[subr])
  return(val)
}


### Similarly for the gradient function
createRobustHuberGradient <- function(x,y, kval=1.5) {
  ## local variables
  xbar <- mean(x)
  ybar <- mean(y)
  function(theta) {
    alpha <- theta[1]
    beta <- theta[2]
    ru = y - alpha - beta*(x - xbar)
    rhok = huber.fn1(ru, k=kval)
    -1*c( sum(rhok*1), sum(rhok*(x-xbar)) )
  }
}
```

Animals Example - Gradient Descent

R code to do Robust regression using the Huber function

```
data(Animals2, package = "robustbase")
animal.kval = 0.766*1.5

rho  <- createRobustHuberRho(log(Animals2$body), log(Animals2$brain), kval=animal.kval)
gradient <- createRobustHuberGradient(log(Animals2$body), log(Animals2$brain), kval=animal.kval)

result <- gradientDescent(theta = c(0,0), rhoFn = rho, gradientFn = gradient,
                          lineSearchFn = gridLineSearch,testConvergenceFn = testConvergence)
### Print the results (rounded to the third decimal point)
as.data.frame(Map(function(x){if (is.numeric(x)) round(x,3) else x}, result))
```

```
##   theta converged iteration fnValue
## 1 3.331      TRUE         20  28.823
## 2 0.691      TRUE         20  28.823
```

The `rlm` function from `MASS` Package

R function `rlm` in the package `MASS` can fit robust models using the Huber-ψ.

```
library(MASS)
temp = rlm(log(Animals2$brain) ~ I(log(Animals2$body)- mean(log(Animals2$body))), psi="psi.huber")
temp$coef
```

for R to input as a function and compute

```
##                                       (Intercept)
##                                         3.3405534
## I(log(Animals2$body) - mean(log(Animals2$body)))
##                                         0.6985483
```

Some comments: Big Data Applicaitons

- When $\rho(\cdot)$ is a **sum** over $u \in \mathcal{P}$, that is when $\rho(\boldsymbol{\theta}, \mathcal{P}) = \sum_{u \in \mathcal{P}} \rho(\boldsymbol{\theta}; u)$ then the gradient descent algorithm can be tailored to some important (big data) applications.

1. **Initialize** ; $i \leftarrow 0$; determine $\widehat{\boldsymbol{\theta}}_0$
2. LOOP:

1. ***Gradient****:*

$$\mathbf{g}_i = \frac{\partial \sum_{u \in \mathcal{P}} \rho(\boldsymbol{\theta}; u)}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}_i} = \sum_{u \in \mathcal{P}} \frac{\partial\, \rho(\boldsymbol{\theta}; u)}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}_i} = \sum_{u \in \mathcal{P}} \mathbf{g}_i(u),$$

2. Gradient ***directions****:*

$$\mathbf{g}_i(u) \leftarrow \frac{\mathbf{g}_i(u)}{||\mathbf{g}_i||} \qquad \text{and} \qquad \mathbf{g}_i \leftarrow \frac{1}{N} \sum_{u \in \mathcal{P}} \mathbf{g}_i(u)$$

3. ***Line search****: Find*

$$\widehat{\lambda} = \arg\min_{\lambda > 0} \rho(\widehat{\boldsymbol{\theta}}_i - \lambda\, \mathbf{g}_i)$$

4. ***Update****:*

$$\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}_i - \widehat{\lambda}\mathbf{g}_i = \widehat{\boldsymbol{\theta}}_i - \frac{\widehat{\lambda}}{N} \sum_{u \in \mathcal{P}} \mathbf{g}_i(u)$$

5. ***Converged?***

if *the iterates are not changing, then* ***return***

else $i \leftarrow i + 1$ *and repeat LOOP.*

3. **Return**: $\widehat{\boldsymbol{\theta}} = \widehat{\boldsymbol{\theta}}_i$

Variations

- When $\rho(\cdot)$ is a sum over $u \in \mathcal{P}$,
  - *Each of steps 2(1), 2(2), and 2(4) break down into sums of **individual** components.*
  - *This can be very handy when N is large, and is sometimes called the **batch gradient descent**.*
- Typically, the value of the step size, λ, is fixed.
  - *In this case the value of λ is called the "learning rate"*
- Each iteration is a sequence of updates created by looping over $u \in \mathcal{P}$ **in any order** and update the iterate for **every** unit u.
  - *For every unit in some order repeat the update*

$$\widehat{\boldsymbol{\theta}} \leftarrow \widehat{\boldsymbol{\theta}} - \lambda^{\star}\mathbf{g}_i(u) \qquad \text{and then set} \qquad \widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}.$$

  - *This often naturally comes up when N is so large that calculations have been farmed out to (say) N machines over a "cloud" of computers.*
- What if instead of updating based on the whole population, we use some subset?
  - *This is **a variation of stochastic gradient descent***

A Small Illustrative Example

- If the population consists of $\{1, 2, 3\}$ and using a fixed λ
- In Gradient Descent or Batch Gradient Descent, the update is

$$\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}_i - \widehat{\lambda}\mathbf{g}_i = \widehat{\boldsymbol{\theta}}_i - \frac{\lambda}{N}\sum_{u\in\mathcal{P}}\mathbf{g}_i(u)$$

- In Stochastic Gradient Descent,
  - *we first calculate* g_1, g_2, g_3 *based on the current estimate* $\widehat{\boldsymbol{\theta}}_i$.
  - *then for some ordering say,* $\{2, 1, 3\}$*, the update rule is*

$$\widehat{\boldsymbol{\theta}}_{i,1} \leftarrow \widehat{\boldsymbol{\theta}}_i - \lambda\mathbf{g}_2(u)$$

$$\widehat{\boldsymbol{\theta}}_{i,2} \leftarrow \widehat{\boldsymbol{\theta}}_{i,1} - \lambda\mathbf{g}_1(u)$$

$$\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}_{i,2} - \lambda\mathbf{g}_3(u)$$

A variation of Stochastic Gradient Descent

- If the population consists of $\{1, 2, 3\}$ and using a fixed λ
- We might take subset, $\mathcal{S}$, of size n from $\mathcal{P}$. e.g. $\mathcal{S} = \{1, 3\}$. Then
  - *we calculate the gradient based on the sample. e.g. calculate g_1, g_3 based on the current estimate $\widehat{\boldsymbol{\theta}}_i$.*
  - *then the general update rule is*

batch only on samples

$$\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}_i - \frac{\lambda}{n} \sum_{u \in \mathcal{S}} \mathbf{g}_i(u)$$

  - *and the update rule for the example is*

$$\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}_i - \frac{\lambda}{2} \left[\mathbf{g}_1(u) + \mathbf{g}_3(u)\right]$$

change sample every time

- Check the notes for an implementation

Solving a system of equations

- An example of implicit population attribute: the solution $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ that satisfies a system of equations

$$\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P}) = \mathbf{0}$$

if equation is well-behaving find solution to its derivative is the same as minimizing

  - *where there are as many independent equations as there are unknowns p.*

- As with the $\rho(\cdot)$ which we previously considered minimizing, a common choice for $\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P})$ is a sum over the population $\mathcal{P}$ and
  - *the attribute of interest $\widehat{\boldsymbol{\theta}}$ is the value $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ that solves*

$$\sum_{u \in \mathcal{P}} \psi(\boldsymbol{\theta}; u) = \mathbf{0}.$$

- This is a **root finding** problem

- Optimization methods to be considered:
  - *Gradient descent (DONE)*
  - ***Newton-Raphson**,*
  - *Iteratively reweighted least-squares.*

Newton-Raphson: A Root Finding Method

- **Idea:** use a first order *Taylor series approximation* to construct an iterative root finding method

  - *Since, $\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P})$ is differentiable (why?),*

$$\boldsymbol{\psi}'(\boldsymbol{\theta}_i; \mathcal{P}) = \left.\frac{\partial \boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P})}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_i}$$

    be the $p \times p$ matrix of partial derivatives.

- Then, a first order approximation can be written as

$$\boldsymbol{\psi}(\boldsymbol{\theta}_i + \boldsymbol{\Delta}; \mathcal{P}) \approx \boldsymbol{\psi}(\boldsymbol{\theta}_i; \mathcal{P}) + \boldsymbol{\psi}'(\boldsymbol{\theta}_i; \mathcal{P}) \times \boldsymbol{\Delta}.$$

  - *Given our current best guess (or approximation) of the root is $\boldsymbol{\theta}_i$.*
  - *We use the first order approximation as a way to improve our current guess by setting $\boldsymbol{\Delta} = \boldsymbol{\theta} - \boldsymbol{\theta}_i$. Then the approximation becomes*

$$\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P}) \approx \boldsymbol{\psi}(\boldsymbol{\theta}_i; \mathcal{P}) + \boldsymbol{\psi}'(\boldsymbol{\theta}_i; \mathcal{P}) \times (\boldsymbol{\theta} - \boldsymbol{\theta}_i)$$

  - *Setting the left hand side $\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P}) = \mathbf{0}$ and solving for $\boldsymbol{\theta}$ yields*

$$\boldsymbol{\theta} \approx \boldsymbol{\theta}_i - \left[\boldsymbol{\psi}'(\boldsymbol{\theta}_i; \mathcal{P})\right]^{-1} \boldsymbol{\psi}(\boldsymbol{\theta}_i; \mathcal{P})$$

    *which suggests the iterative **Newton-Raphson** algorithm for root finding.*

Newton-Raphson algorithm

1. **Initialize** ; $i \leftarrow 0$; determine an initial estimate $\widehat{\boldsymbol{\theta}}_0$ (how, depends on problem)

2. LOOP:

    1. ***Update*** *the iterate:* If only 1 parameter => scalar => newton method

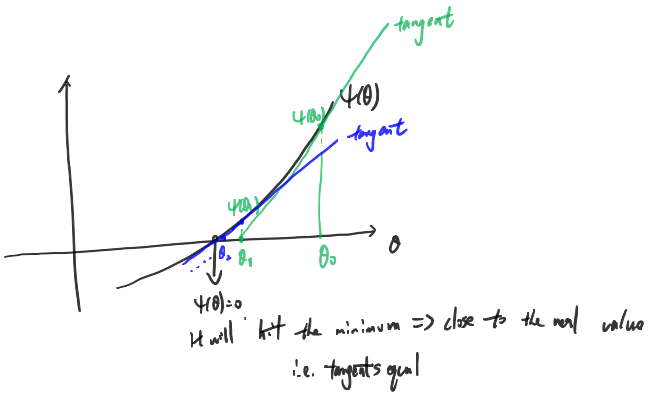
$$\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}_i - \left[\boldsymbol{\psi}'(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})\right]^{-1} \boldsymbol{\psi}(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})$$

    2. ***Converged?***

    if *the iterates are not changing, then* ***return***

    else $i \leftarrow i + 1$ *and repeat LOOP.*

3. **Return**: $\widehat{\boldsymbol{\theta}} = \widehat{\boldsymbol{\theta}}_i$



Newton's method

- When $p = 1$, all vectors above are scalars and the updates simplifies to

$$\widehat{\theta}_{i+1} \leftarrow \widehat{\theta}_i - \frac{\psi(\theta_i; \mathcal{P})}{\psi'(\theta_i; \mathcal{P})}$$

  and the method is known simply as **Newton's method**.

- In other words, when θ is a scaler (not a $p \times 1$ vector), Newton-Raphson's method is called Newton's method.

Newton's Method: R code

```
Newton <- function(theta = 0,
                   psiFn, psiPrimeFn,
                   testConvergenceFn = testConvergence,
                   maxIterations = 100,    # maximum number of iterations
                   tolerance = 1E-6,       # parameters for the test
                   relative = FALSE        # for convergence function
) {
  ## Initialize
  converged <- FALSE
  i <- 0
  ## LOOP
  while (!converged & i <= maxIterations) {
    ## Update theta
    thetaNew <- theta - psiFn(theta)/psiPrimeFn(theta)
    ##
    ## Check convergence
    converged <- testConvergenceFn(thetaNew, theta,
                                   tolerance = tolerance,
                                   relative = relative)
    ## Update iteration
    theta <- thetaNew
    i <- i + 1
  }
  ## Return last value and whether converged or not
  list(theta = theta,
       converged = converged,
       iteration = i,
       fnValue = psiFn(theta)
       )
}
```

Example - Weighted Sum

Use Newton's method to finr the root of the weighted sum $\sum_{u \in \mathcal{P}} w_u(y_u - \theta) = 0$.

- We have

$$\psi(\theta) = \sum_{u \in \mathcal{P}} w_u(y_u - \theta)$$

and

$$\psi'(\theta) = -\sum_{u \in \mathcal{P}} w_u$$

We'll define `psi(...)` and `psiPrime(...)` in `R`.

Example: Weighted Sum & Facebook Data

Suppose we look at the `facebook` data set and the number of `likes` a posting receives.

```
facebookfile <- paste(directory, "FacebookMetrics", "facebook.csv", sep=dirsep)
facebook <- read.csv(facebookfile)
### Remove all rows with missing data
fb <- na.omit(facebook)
head(fb)
```

```
##   All.interactions share like comment Impressions.when.page.liked
## 1              100    17   79       4                        3078
## 2              164    29  130       5                       11710
## 3               80    14   66       0                        2812
## 4             1777   147 1572      58                       61027
## 5              393    49  325      19                        6228
## 6              186    33  152       1                       16034
##   Impressions Paid Post.Hour Post.Weekday Post.Month    Category   Type
## 1        5091    0         3            4         12     Product  Photo
## 2       19057    0        10            3         12     Product Status
## 3        4373    0         3            3         12 Inspiration  Photo
## 4       87991    1        10            2         12     Product  Photo
## 5       13594    0         3            2         12     Product  Photo
## 6       20849    0         9            1         12     Product Status
##   Page.likes
## 1     139441
## 2     139441
## 3     139441
## 4     139441
## 5     139441
## 6     139441
```

Weighted Sum & Facebook Data (ψ and ψ')

- Let's weight the likes inversely proportional to the number of `Impressions` the post has.
  - *That is the more often it is seen, the lower the weight we give to the `likes` it receives.*

```
### Here we create both functions at once
createPsiFns <- function(y, wt) {
  psi <- function(theta = 0) {sum(wt * (y -theta))}
  psiPrime <- function(theta = 0) {-sum(wt)}
  list(psi = psi, psiPrime = psiPrime)
}

### Create them for the particular data
psiFns <- createPsiFns(y = fb$like, wt = 1/fb$Impressions)
psi <- psiFns$psi
psiPrime <- psiFns$psiPrime
```

Use these functions together with Newton's method

Weighted Sum & Facebook Data (cont'd)

```
library(knitr)
result <- Newton(theta = mean(fb$like), psiFn = psi, psiPrimeFn = psiPrime)
kable(as.data.frame(result))
```

theta	converged	iteration	fnValue
78.46845	TRUE	2	0

```
### The theta of which should be the same as the weighted average
y <- fb$like
wt <- 1/fb$Impressions
kable(data.frame(theta = result$theta, weightedAve = (sum(wt*y) / sum(wt))))
```

theta	weightedAve
78.46845	78.46845

- **Exercise** Why does this converge in two iterations?

  The tangent of a line is itself

- **Exercise** Try out some other ψ functions for location estimation and write the code to implement them. For example, Andrews's sine function, the bisquare weight function, Huber's psi, or Hampel's psi for more complex coding challenges (not everywhere differentiabele ψs).

General Newton-Raphson R code

```
NewtonRaphson <- function(theta,
                          psiFn, psiPrimeFn, dim,
                          testConvergenceFn = testConvergence,
                          maxIterations = 100, tolerance = 1E-6, relative = FALSE
) {
  if (missing(theta)) {
    ## need to figure out the dimensionality
    if (missing(dim)) {dim <- length(psiFn())}
    theta <- rep(0, dim)
  }
  converged <- FALSE
  i <- 0
    while (!converged & i <= maxIterations) {
    thetaNew <- theta - solve(psiPrimeFn(theta), psiFn(theta))
    converged <- testConvergenceFn(thetaNew, theta, tolerance = tolerance,
                                   relative = relative)

    theta <- thetaNew
    i <- i + 1
  }
  ## Return last value and whether converged or not
  list(theta = theta, converged = converged, iteration = i, fnValue = psiFn(theta)
       )
}
```

Newton versus Newton-Raphson: R code

- `NewtonRaphson(...)` is identical to `Newton(...)` with two important exceptions:

  1. First, since this is Newton's method in arbitrary dimensions, we must be given the dimensionality either implicitly from `theta` or explicitly via `dim`.
     - *If neither of these arguments are given, it is inferred by evaluating the `psiFn(...)` once with no arguments.*
  2. The update for `theta` is attained by using `solve(...)` instead.
     - *This is needed because*

       $$\left[\boldsymbol{\psi}'(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})\right]^{-1} \boldsymbol{\psi}(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})$$

       is the value of $\mathbf{x}$ *that solves the system of equations*

       $$\left[\boldsymbol{\psi}'(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})\right] \mathbf{x} = \boldsymbol{\psi}(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})$$

       which is precisely what the function `solve(...)` does and does so in a numerically reliable way (e.g. forming an LU *decomposition and backsolving twice).*

General Newton-Raphson Example

Suppose for example that we were finding that $\boldsymbol{\theta}$ which minimized the Rosenbrock function

$$\rho(\boldsymbol{\theta}) = (1-\theta_1)^2 + 100\left(\theta_2 - \theta_1^2\right)^2$$

In optimization, the roots of the gradient are of importance

$$\mathbf{g} = \boldsymbol{\psi}(\boldsymbol{\theta}) = \left[400\theta_1^3 - 400\theta_1\theta_2 + 2\theta_1 - 2 \;\;,\;\; -200\theta_1^2 + 200\theta_2\right]$$

for Newton-Raphson we need the matrix of partial derivatives

$$\boldsymbol{\psi}'(\boldsymbol{\theta}) = \begin{bmatrix} 1200\,\theta_1^2 - 400\,\theta_2 + 2 & -400\,\theta_1 \\ -400\,\theta_1 & 200 \end{bmatrix}$$

$\nabla\psi(\theta) = \begin{bmatrix} \frac{\partial\psi_1(\theta_1,\theta_2)}{\partial\theta_1} & \frac{\partial\psi_1(\theta_1,\theta_2)}{\partial\theta_2} \\ \frac{\partial\psi_2(\theta_1,\theta_2)}{\partial\theta_1} & \frac{\partial\psi_2(\theta_1,\theta_2)}{\partial\theta_2} \end{bmatrix}$

```
psiPrime <- function(theta = c(0,0)) {
 val = matrix(0, nrow=length(theta), ncol=length(theta))
 val[1,1] = 1200*theta[1]^2 - -400*theta[2] + 2
 val[1,2] = -400*theta[1]
 val[2,1] = -400*theta[1]
 val[2,2] = 200
 return(val)
}
```

General Newton-Raphson (cont'd)

- Recall the R functions for the objective function and gradient

```
rho <- function(theta) {  (1-theta[1])^2 + 100*(theta[2]-theta[1]^2)^2 }
g <- function(theta=c(0,0)) { c( 400*theta[1]^3 -400*theta[1]*theta[2] +2*theta[1]-2,
                                 -200*theta[1]^2+200*theta[2] ) }
```

We find the value $\widehat{\theta}$ as

```
gresult = gradientDescent(theta= c(0,0), rhoFn = rho, gradientFn = g,
                lineSearchFn = gridLineSearch,
                testConvergenceFn = testConvergence, maxIterations=1000)
kable(as.data.frame(gresult))
```

theta	converged	iteration	fnValue
0.7987955	TRUE	419	0.0510196
0.6278095	TRUE	419	0.0510196

← use the smaller value => minimum

```
nresult <- NewtonRaphson(theta= c(0,0), psiFn = g, psiPrimeFn = psiPrime)
kable(as.data.frame(nresult))
```

not the same

1. multiple minimal
   2 different objectives
2. sensitive to the initial value
3. derivative = 0
   => can have 3 cases
   not necessarily minimum

theta	converged	iteration	fnValue
1	TRUE	3	0
1	TRUE	3	0

```
rho(nresult$theta)
```

```
## [1] 0
```

Example with different staring value

- An algorithm's performance depends on the initial guess.

We find the value $\widehat{\theta}$ as

```
gresult = gradientDescent(theta= c(-2,-1), rhoFn = rho, gradientFn = g,
                lineSearchFn = gridLineSearch,
                testConvergenceFn = testConvergence, maxIterations=10^4)
kable(as.data.frame(gresult))
```

theta	converged	iteration	fnValue
0.7989451	TRUE	426	0.0509606
0.6280480	TRUE	426	0.0509606

```
nresult <- NewtonRaphson(theta= c(-2,-1), psiFn = g, psiPrimeFn = psiPrime, maxIterations=10^4)
kable(as.data.frame(nresult))
```

theta	converged	iteration	fnValue
1.000133	TRUE	4195	0.0002661
1.000266	TRUE	4195	0.0000000

```
rho(nresult$theta)
```

initial point is on the flat part

```
## [1] 1.769575e-08
```

Exercise: repeat this with the initial values `theta= c(-7,-2)`. What do you observe?

Iteratively reweighted least-squares

- Optimization methods to be considered:
  - *Gradient descent (DONE)*
  - *Newton-Raphson (DONE)*
  - ***Iteratively reweighted least-squares***

This is an iterative method to solve weighted least squares problems, hence the name.

Iteratively reweighted least-squares (setup)

- $\min_{(\alpha,\beta)} \sum_{u\in\mathcal{P}} \rho(y_u - \alpha - \beta(x_u - c))$ for some function $\rho(\dots)$.
  - *Differentiating this with respect to α and β gives the minimum as a solution to*

$$\sum_{u\in\mathcal{P}} \psi(y_u - \alpha - \beta(x_u - c)) \begin{pmatrix} 1 \\ \\ x_u - c \end{pmatrix} = \mathbf{0}, \quad where \;\; \psi(\cdots) = \rho'(\cdots)\$.$$

  - *If we let $r_u = y_u - \alpha - \beta(x_u - c)$ and $\mathbf{z}_u = (1,\ x_u - c)^{\mathsf{T}}$, then the equation is*

$$\sum_{u\in\mathcal{P}} \psi(r_u)\, \mathbf{z}_u = \mathbf{0}.$$

- For weighted least-squares, $\rho(r) = wr^2$, $\psi(r) = 2wr$ and solve

$$\sum_{u\in\mathcal{P}} w_u r_u\, \mathbf{z}_u = \mathbf{0}$$

- The above general equation can be made to look like the weighted least squares equation as follows:

$$\sum_{u\in\mathcal{P}} \psi(r_u)\, \mathbf{z}_u = \sum_{u\in\mathcal{P}} \left(\frac{\psi(r_u)}{r_u}\right) r_u\, \mathbf{z}_u = \sum_{u\in\mathcal{P}} w_u r_u\, \mathbf{z}_u$$

  where the weight is $w_u = \psi(r_u)/r_u$ (provided $r_u \neq 0$).

Iteratively reweighted least-squares algorithm

1. **Initialize** ; $i \leftarrow 0$; determine an initial estimate $\widehat{\boldsymbol{\theta}}_0$
2. LOOP:
    1. ***Construct residuals*** *for all* $u \in \mathcal{P}$

$$r_u = y_u - \mathbf{z}_u{}^{\mathsf{T}} \boldsymbol{\theta}_i$$

    2. ***Construct weights*** *for all* $u \in \mathcal{P}$

$$w_u = \frac{\psi(r_u)}{r_u}$$

    3. ***Solve*** *the weighted least squares problem*

$$\sum_{u \in \mathcal{P}} w_u (y_u - \mathbf{z}_u{}^{\mathsf{T}} \boldsymbol{\theta}) \, \mathbf{z}_u$$

    for $\widehat{\boldsymbol{\theta}}$

    4. ***Update*** *the iterate:*

$$\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}$$

    5. ***Converged?***

    if *the iterates are not changing, then* ***return***

    else $i \leftarrow i + 1$ *and repeat LOOP.*

3. **Return**: $\widehat{\boldsymbol{\theta}} = \widehat{\boldsymbol{\theta}}_i$

Some comments

- It is completely general for any attribute that can be expressed as

$$y_u = \mathbf{z}_u^\top \boldsymbol{\theta} + r_u.$$

  Such attributes are called **linear response models** which are fitted to the population according to the definition of $\psi(\cdot)$.

- When we choose $c = \bar{x}_w$, i.e. $c =$ the weighted average of the x_i's,
  - *the solution to the weighted least squares problem has a closed form.*
  - *Recall* $\widehat{\boldsymbol{\theta}} = (\widehat{\alpha},\ \widehat{\beta})^\top$ *and*

$$\widehat{\alpha} = \bar{y}_w \quad \text{and} \quad \widehat{\beta} = \frac{\sum_{u \in \mathcal{P}} w_u (x_u - \bar{x}_w) y_u}{\sum_{u \in \mathcal{P}} w_u (x_u - \bar{x}_w)^2}$$

  where $\bar{y}_w$ *is the weighted average of the* y_i*'s.*

Iteratively reweighted least-squares R code

We can implement iteratively weighted least squares for a straight-line model as follows:

```
irls <- function(y, x, theta, psiFn,
                 dim = 2, delta = 1E-10,
                 testConvergenceFn = testConvergence,
                 maxIterations = 100,    # maximum number of iterations
                 tolerance = 1E-6,       # parameters for the test
                 relative = FALSE        # for convergence function
) {
  if (missing(theta)) {theta <- rep(0, dim)}
  ## Initialize
  converged <- FALSE
  i <- 0
  N <- length(y)
  wt <- rep(1,N)
  ## LOOP
  while (!converged & i <= maxIterations) {
    ## get residuals
    resids <- getResids(y, x, wt, theta)
    ## update weights  (should check for zero resids)
    wt <- getWeights(resids, psiFn, delta)
    ## solve the least squares problem
    thetaNew <- getTheta(y, x, wt)
    ##
    ## Check convergence
    converged <- testConvergenceFn(thetaNew, theta,
                                   tolerance = tolerance,
                                   relative = relative)
    ## Update iteration
    theta <- thetaNew
    i <- i + 1
  }
  ## Return last value and whether converged or not
  list(theta = theta,
       converged = converged,
       iteration = i
       )
}
```

Iteratively reweighted least-squares for Least Squares

- For least squares, it remains to write the functions
  - `getResids(...)`,
  - `getWeights(...)`,
  - `getTheta(...)`.

We have,

- $\rho(r) = r^2/2$,
- $\rho'(r) = \psi(r) = r$
- $w = \frac{\psi(r)}{r} = \frac{r}{r} = 1$

`irls` function for Least Squares

```
getResids <- function(y, x, wt, theta) {
  xbarw <- sum(wt*x)/sum(wt)
  alpha <- theta[1]
  beta <- theta[2]
  ## resids are
  y - alpha - beta * (x - xbarw)
}

getWeights <- function(resids, psiFn, delta = 1E-10) {
  ## for calculating weights, minimum |residual| will be delta
  smallResids <- abs(resids) <= delta
  ## take care to preserve sign (in case psi not symmetric)
  resids[smallResids] <- delta * sign(resids[smallResids])
  ## calculate and return weights
  psiFn(resids)/resids
}


getTheta <- function(y, x, wt) {
  theta <- numeric(length = 2)
  ybarw <- sum(wt * y)/sum(wt)
  xbarw <- sum(wt * x)/sum(wt)
  theta[1] <- ybarw
  theta[2] <- sum(wt * (x - xbarw) * y) / sum(wt * (x - xbarw)^2)
  ## return theta
  theta
}
```

Iteratively reweighted least-squares for Least Squares (cont'd)

We can try these out for the least-squares problem

```
psi <- function(resid) {resid}

result <- irls(waldo$Y, waldo$X, theta = c(0,0), psiFn = psi)
kable(as.data.frame(result))
```

theta	converged	iteration
3.8753064	TRUE	2
0.1429041	TRUE	2

```
#commpare these to closed form solutions for alpha and beta
alpha.hat = mean(waldo$Y)
beta.hat = sum((waldo$X-mean(waldo$X))*waldo$Y)/sum((waldo$X-mean(waldo$X))^2)
c(alpha.hat , beta.hat)
```

```
## [1] 3.8753064 0.1429041
```

Example - Robust Regression

- The $\rho(\cdot)$ function was

$$\rho_k(r) = \begin{cases} \frac{1}{2}r^2 & \text{for } |r| \leq k, \\ k\ |r| - \frac{1}{2}k^2, & \text{otherwise.} \end{cases}$$

  - *and now we have that*

$$\psi(r) = \rho_k'(r) = \begin{cases} r & \text{for } |r| \leq k, \\ \text{sign}(r)\ r, & \text{otherwise.} \end{cases}$$

```
huber.psi <- function(resid, k = 1.5*0.7763) {
  val = resid
  subr = abs(resid) > k
  val[ subr ] = k*sign(resid[subr])
  return(val)
}
```

Animals Example

```
result <- irls(log(Animals2$brain), log(Animals2$body), theta = c(1,1),
               psiFn = huber.psi, maxIterations = 100)
kable(as.data.frame(result))
```

theta	converged	iteration
3.1145983	TRUE	8
0.6950183	TRUE	8

```
temp = rlm(log(Animals2$brain) ~ log(Animals2$body) , psi="psi.huber")
temp$coef
```

```
##         (Intercept) log(Animals2$body)
##           2.1281218          0.6985483
```

- **Note** these two fitting models are different because the first one (developed code) is fitting

$$y_u = \alpha + \beta\,(x_u - c) + r_u$$

and the second one (`rlm` function) is fitting

$$y_u = \alpha + \beta x_u + r_u$$

Animals Example: finding c

- We can fit the matching model by finding the value of c.

```
c0 = (result$theta[1] - temp$coef[1])/result$theta[2]
c0
```

```
## (Intercept)
##    1.419353
```

```
temp2 = rlm(log(Animals2$brain) ~ I(log(Animals2$body) - c0)  , psi="psi.huber")
temp2$coef
```

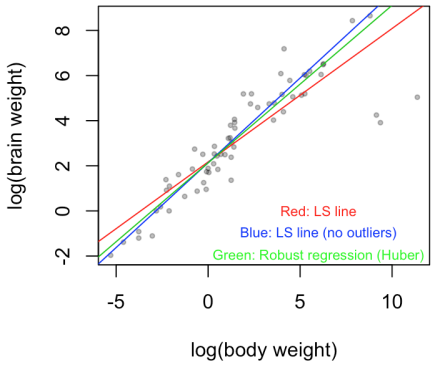
```
##                (Intercept) I(log(Animals2$body) - c0)
##                  3.1196088                  0.6985483
```

```
result$theta
```

```
## [1] 3.1145983 0.6950183
```

Animals Example

- log(Brain) versus log(Body) weight
  - *`Red line` - LS regression using all the data*
  - *`Blue line` - LS regression while removing those three `outlier` points*
  - *`Green line` - Robust regression with Huber function.*



- Which units are those three ``influential" points?

Animals Example - the three outliers

- 63 Land Mammals and 3 dinosaurs!

```
Animals2[60:65,]
```

```
##                    body brain
## Echidna           3.000  25.0
## Brazilian tapir 160.000 169.0
## Tenrec            0.900   2.6
## Phalanger         1.620  11.4
## Tree shrew        0.104   2.5
## Red fox           4.235  50.4
```

```
Animals2[c(6,16,26,60:65),] #Recall from slide 19 : observations 6, 16, 26
```

```
##                     body brain
## Dipliodocus     11700.000  50.0
## Triceratops      9400.000  70.0
## Brachiosaurus   87000.000 154.5
## Echidna             3.000  25.0
## Brazilian tapir   160.000 169.0
## Tenrec              0.900   2.6
## Phalanger           1.620  11.4
## Tree shrew          0.104   2.5
## Red fox             4.235  50.4
```

The three outliers (cont'd)

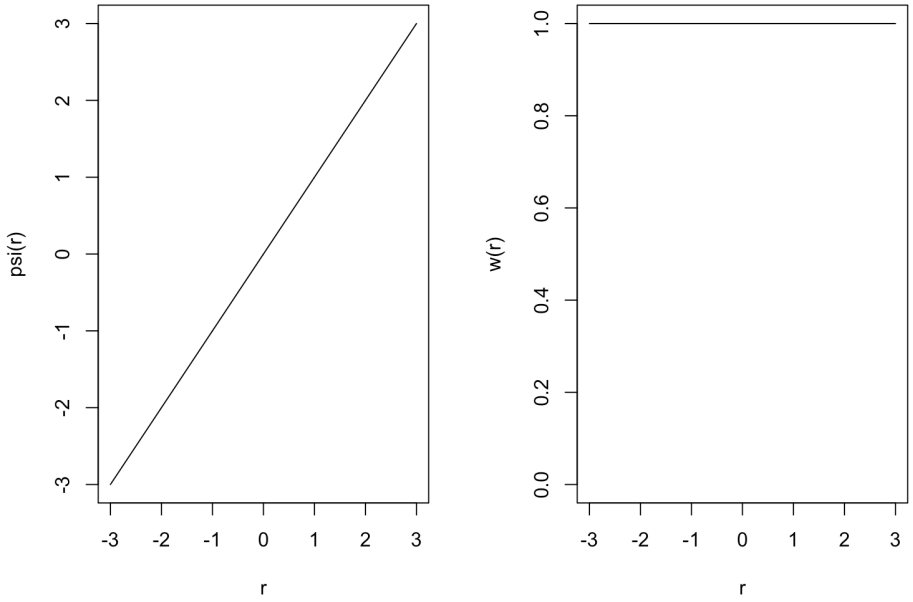
All three are dinosaurs!

Exercises

- Explain how zero residuals are handled in the construction of weights and why it might make sense.
- Try the iteratively reweighted least squares for some other ψ functions.
- Show how the above iteratively reweighted least squares could be made to work to fit a parabola in x.
- Using the `R` function `lm(...)` (and any related functions) show how the above iteratively reweighted least squares can be used for any linear model (and ψ).

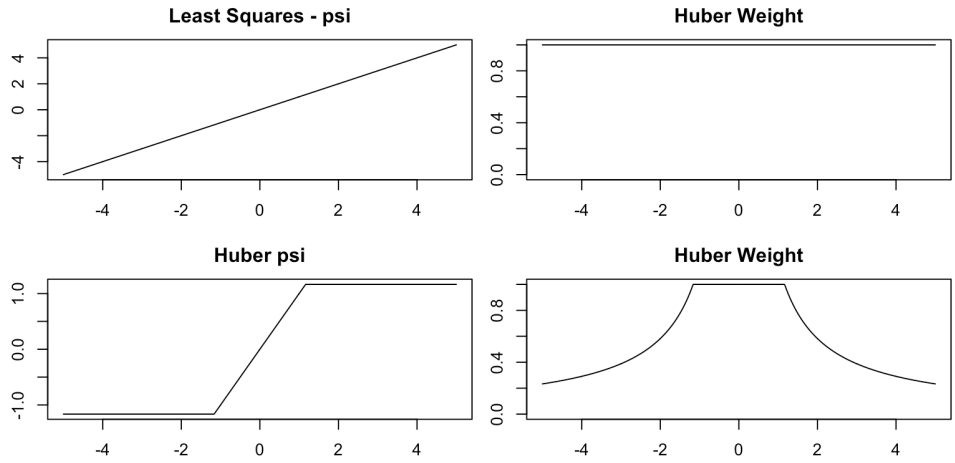
Least Squares Review ($\rho' = \psi$ and w)

- To perform iteratively reweighted least-squares we only need to specify $\psi(\cdot)$
  - *for least squares we had* $\psi(r) = r$ *and weights* $w(r) = \psi(r)/r = 1$



Robust Regression - Huber Function Review

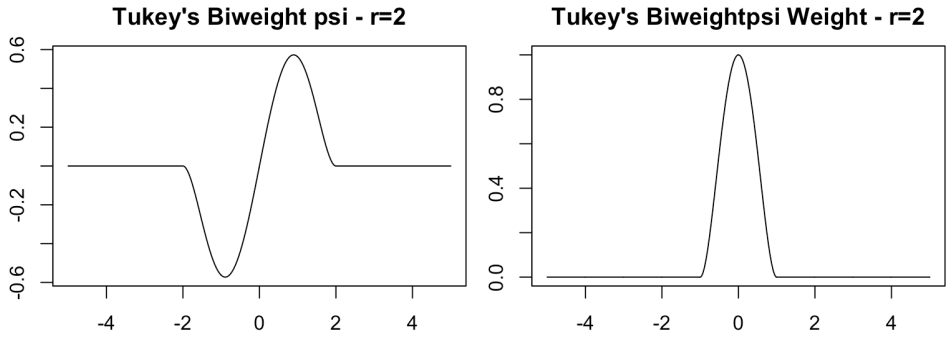
- To perform iteratively reweighted least-squares we only need to specify $\psi(\cdot)$
  - *for least squares we had that* $\psi(r) = r$ *and weights* $w(r) = \psi(r)/r = 1$
- Compared to least squares we have
  - *What does this look like?*



Tukey's biweight

- Another idea then might be to have the psi function "redescend", that is actually come back to the horizontal axis.
  - *"Tukey's biweight" psi named after John Tukey.*

$$\psi(r) = \begin{cases} r\,(1-(r/c)^2)^2 & |r| \le c \\ \\ 0 & |r| > c. \end{cases}$$



ψ-function

The sorts of "psi functions" we might be interested generally satisfy the following:

- ψ is a piecewise continuous function $\psi : \mathfrak{R} \to \mathfrak{R}$
- ψ is odd, i.e. $\psi(-r) = -\psi(r) \ \forall \ r$
- $\psi(r) \geq 0$ for $r \geq 0$
- $\psi(r) > 0$ for $0 < r < r^*$ where $r^* = \sup\{x : \psi(x) > 0\}$. Note that $r^* > 0$ and possibly $r^* = \infty$
- Its slope is 1 at 0, that is $\psi'(0) = 1$

Of course every such function $\psi(r)$ induces a corresponding $\rho(r)$ function

$$\rho(r) = \int_{-\infty}^{r} \psi(u)du.$$

So we can actually choose ψ based on how we think it will behave and infer ρ from our choice.