2.3 Implicitly defined Population Attributes

Contents

2.3.1 The minimum of a function

- In many cases, attributes of a population are defined **implicitly**, typically as the solution to some equation or set of equations.
  - For example, suppose we are interested in a (possibly vector-valued) attribute $\boldsymbol{\theta}$ which minimizes some function of $\rho(\mathcal{P})$ of the variates in the population.
  - That is, we want the value $\widehat{\boldsymbol{\theta}}$ which satisfies

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

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

- Note that maximizing a function is the same as minimizing its negation,
  - i.e., $\max_{\boldsymbol{\theta} \, \in \, \boldsymbol{\Theta}} \rho(\boldsymbol{\theta}; \mathcal{P}) = \min_{\boldsymbol{\theta} \, \in \, \boldsymbol{\Theta}} -\rho(\boldsymbol{\theta}; \mathcal{P})$.
  - Therefore, we only need to consider minimization here.

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

Examples

Some 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**:

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

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

- **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

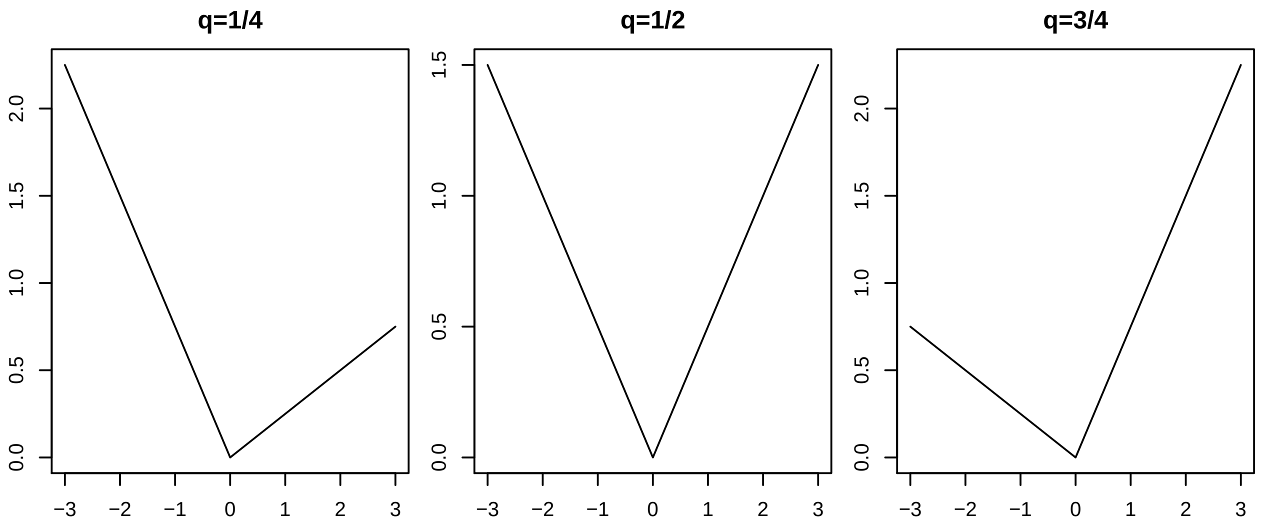
$$\rho_q(z) = \begin{cases} qz & z \geq 0 \\ -(1-q)z & z < 0 \end{cases}$$

That is, the quantile $\widehat{\theta} = Q_y(q)$ is the solution to

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

Some example of the vee function are

```r
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')
```



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

- Set the seed `set.seed(341)`, so that all simulations give the same answer when you rerun them.

- Generate a simulated population

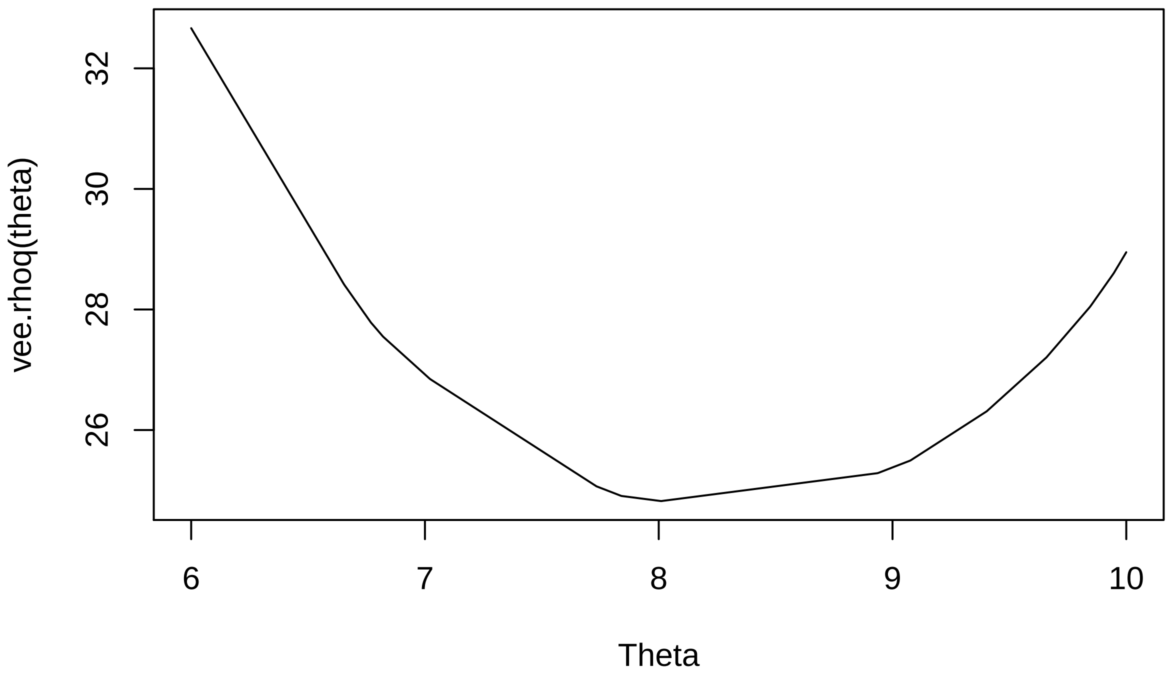
```r
set.seed(341)
y.pop <- rnorm(25, 10, 3)
```

- Write a function the sum of vee functions to be minimized

```r
vee.rhoq <- function(theta, q.val=0.3){
  # the default value is q=0.3
  temp <- sum(sapply(y.pop-theta, rhoq, q=q.val))
  return(temp)
}
```

Plotting the function:

```r
theta = seq(6,10,length.out=1000)
vee.rhoq.of.theta.for.q.0.3 = numeric(length(theta))
for(i in 1:length(theta)){
  vee.rhoq.of.theta.for.q.0.3[i] = vee.rhoq(theta[i], q.val=0.3)
}
plot(theta, vee.rhoq.of.theta.for.q.0.3,
     xlab = 'Theta' , ylab = 'vee.rhoq(theta)', type='l')
```



Using a numerical minimizer `nlminb` and `optimize`

```r
nlminb(0.5, vee.rhoq, q.val=0.3)
```

```
## $par
## [1] 8.010171
##
## $objective
## [1] 24.82168
##
## $convergence
## [1] 0
##
```

```
## $iterations
## [1] 12
##
## $evaluations
## function gradient
##       22       14
##
## $message
## [1] "X-convergence (3)"
```

```
optimize( vee.rhoq, interval=range(y.pop), q.val=0.3)
```

```
## $minimum
## [1] 8.010159
##
## $objective
## [1] 24.82169
```

Compare to R built-in function - Note that we used q=0.3 above

```
quantile(y.pop, 0.3)
```

```
##      30%
## 8.195491
```

Least Squares Regression

A familiar **vector valued** attribute is the regression coefficients of a line

$$y_u = \alpha + \beta\,(x_u - c) + r_u \qquad \forall\, u \in \mathcal{P}$$

where c is meaningful in the data set (e.g. in the middle, i.e. $c = \overline{x}$). The parameters of interest are $\boldsymbol{\theta} = (\alpha, \beta)^{\mathsf{T}}$

- These coefficients are determined implicitly by

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

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

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

- and the fitted values are

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

- The residuals are
$$r_u = y_u - \alpha - \beta x_u$$
Each residual is the signed vertical distance between the point (x_u, y_u) and the point $(x_u, \alpha + \beta x_u)$. The latter point is the value of the line calculated at $x = x_u$ given by the values of $\boldsymbol{\theta} = (\alpha, \beta)$.

- **Note**: we often fit the line
$$y_u = \alpha + \beta \ (x_u - c) + r_u$$
  - Then the coefficient α has the interpretation as the value of the line when $x_u = c$;
  - One popular choice is $c = \overline{x}$ (what is the interpretation of α?)
  - 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.
  - In this case, we have (show this)

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

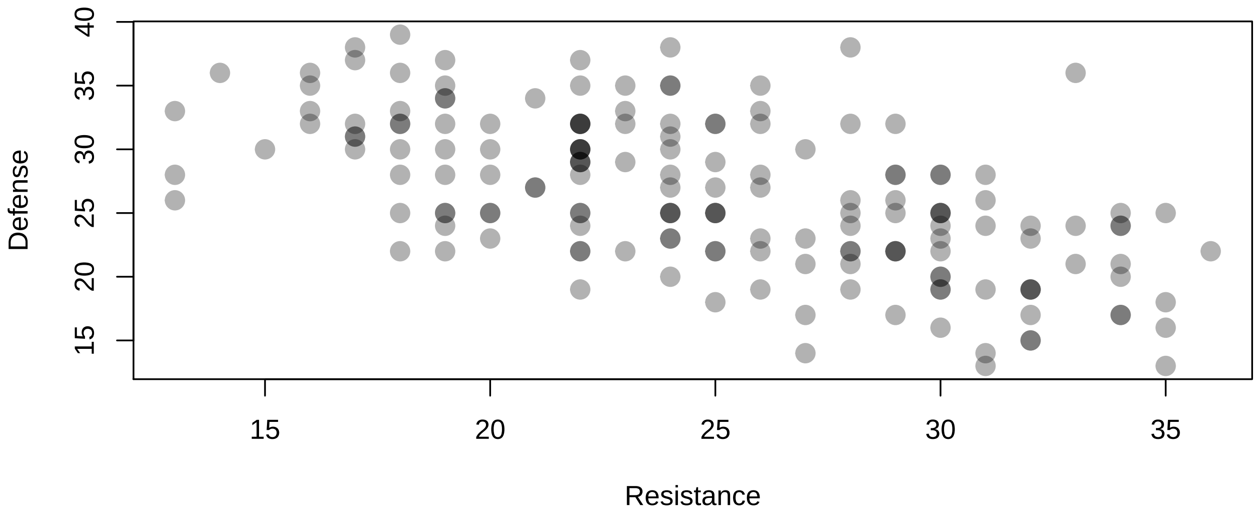
Fire Emblem Heroes Example

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

```
feh = read.csv("../Data/feh/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
```

```
plot(feh$RES, feh$DEF, main= "",
     pch = 19,  cex=1.5,
     col=adjustcolor("black", alpha = 0.3 ),
     xlab="Resistance", ylab="Defense" )
```



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

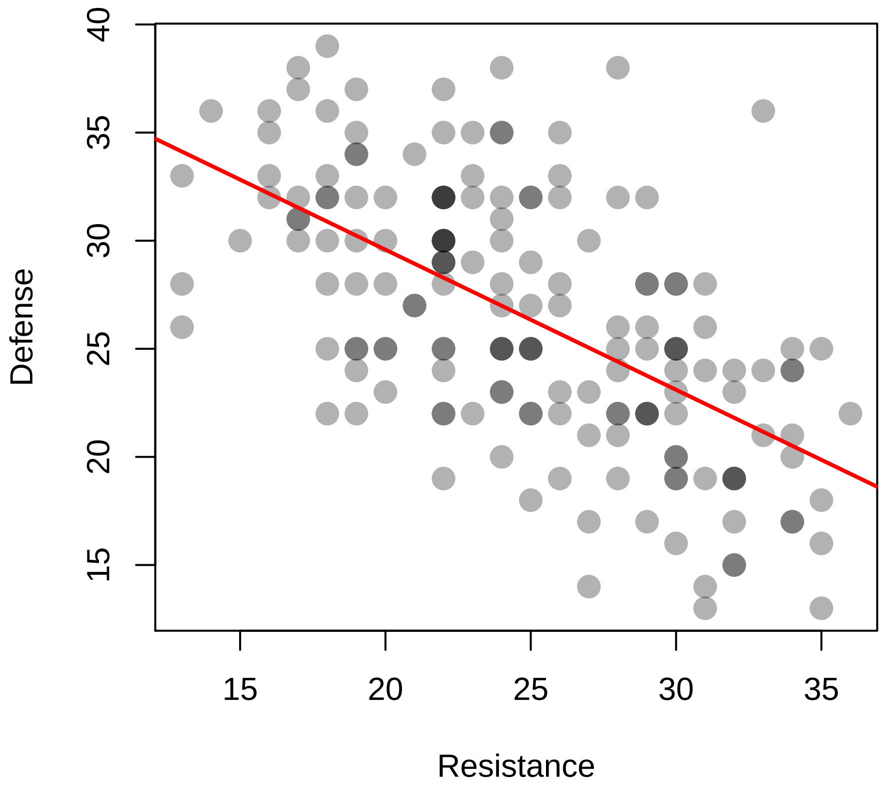
$$\text{Defense} = 42.5 - 0.65 \times \text{Resistance} + \text{error}$$

  - or equivalently as

$$\text{Defense} = 26.4 - 0.65 \times [\text{Resistance} - 24.9] + \text{error}$$

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

```r
feh = read.csv("../Data/feh/feh.csv", header=TRUE)
par(mfrow=c(1,1))

plot(feh$RES, feh$DEF, main= "",
     pch = 19,  cex=1.5,
     col=adjustcolor("black", alpha = 0.3 ),
     xlab="Resistance", ylab="Defense" )

temp = lm(DEF ~ RES, data=feh)
abline(coef=temp$coef, col=2, lwd=2)
```



- What have we done here?

```
## We have summarized the relationship between
## two variables in a population with a line.
```

- Does it make sense to use a regression line here?
- In terms of the game is the regression line important?

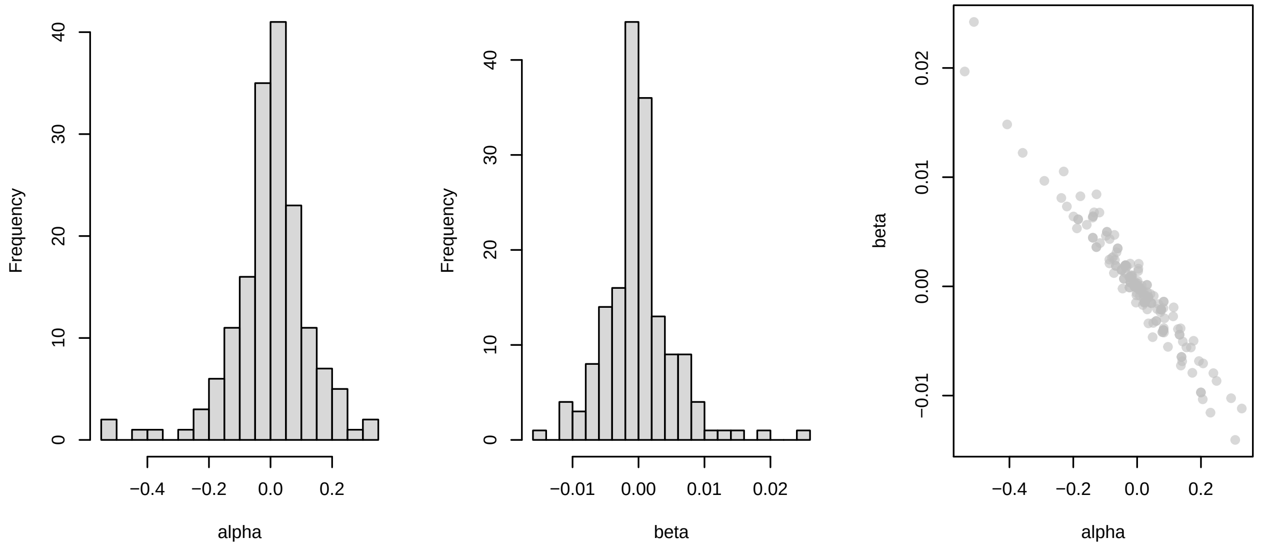
```
## Yes, it indicates there is negative relationship
## between defense and resistance.
##
## Most importanly, there is only one character with
## high defense and resistance.
##
## Rule of thumb: A character has either high
## defense or resistance.
```

- We can calculate the influence of each observation.

```
N = nrow(feh)
delta = matrix(0, nrow=N, ncol=2)
for (i in 1:N) {
  ## feh[-i] removes the ith row from a vector
  tempi = lm(DEF ~ RES, data=feh[-i,])
  delta[i,] = temp$coef - tempi$coef
}
```

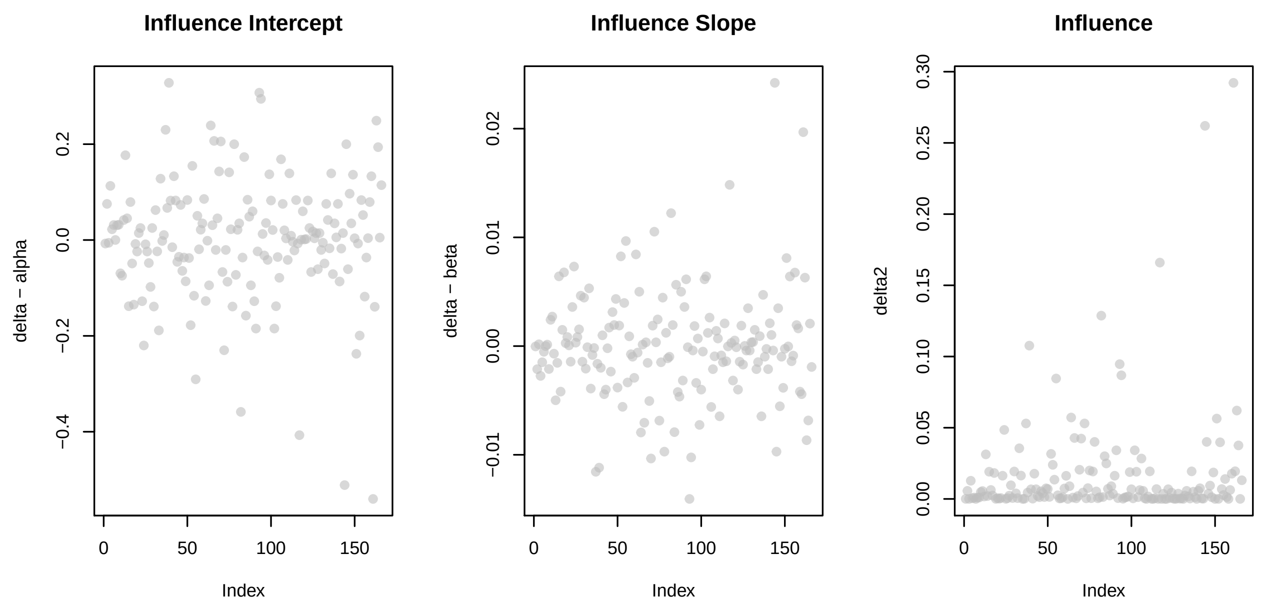
Then plot the difference among the parameters

```
par(mfrow=c(1,3))

hist(delta[,1], breaks="FD", main="", xlab="alpha",
     col=adjustcolor("grey", 0.6))
hist(delta[,2], breaks="FD", main="", xlab="beta",
     col=adjustcolor("grey", 0.6))
plot(delta, xlab="alpha", ylab="beta",
      pch=19, col=adjustcolor("grey", 0.6))
```



To combine the vector attributes, we use the squared distance and plot this measure of influence by index.

```
par(mfrow=c(1,3))

plot(delta[,1], ylab="delta - alpha", main="Influence Intercept",
     pch=19, col=adjustcolor("grey", 0.6) )
plot(delta[,2], ylab="delta - beta", main="Influence Slope",
     pch=19, col=adjustcolor("grey", 0.6) )

delta2 = apply(delta,1, function(z) { sum(z^2) } )
plot(delta2, main="Influence",
      pch=19, col=adjustcolor("grey", 0.6) )
```



- What other way could we combine the attributes?

The two observations with the largest influence.

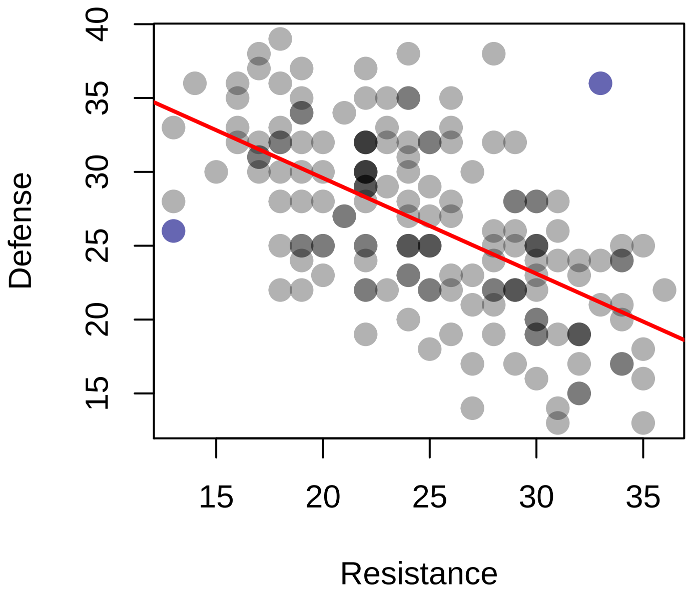
```
feh[delta2>0.25, ]
```

```
##       Name        Type     Move HP ATK SPD DEF RES Total
## 144 Sheena   Green Axe  Armored 45  30  25  36  33   169
## 161 Virion Neutral Bow Infantry 46  31  31  26  13   147
```

These are the weakest and one of the stronger characters in terms of Resistance.

- We can highlight these points on the scatterplot.

```
col.nam = c(adjustcolor("black", alpha = 0.3 ), adjustcolor("navyblue", alpha = 0.6 ) )

plot(feh$RES, feh$DEF, main= "",
     pch = 19,  cex=1.5,
     col= col.nam[(delta2>0.25) + 1],
     xlab="Resistance", ylab="Defense" )

temp = lm(DEF ~ RES, data=feh)
abline(coef=temp$coef, col=2, lwd=2)
```



Animals - Example

In this section we show how you can investigate how sensitive attributes (intercept and slope) are to influential points and will show how to design attributes which are resistant to the effect of influential points a.k.a. outliers.

- **Brain versus Body weight**
  - This is a built-in data frame in R (`robustbase` library) with average brain and body weights for 62 species of land mammals and three others.
  - `body` represents body weight in `kg`
  - `brain` represents brain weight in `grams`

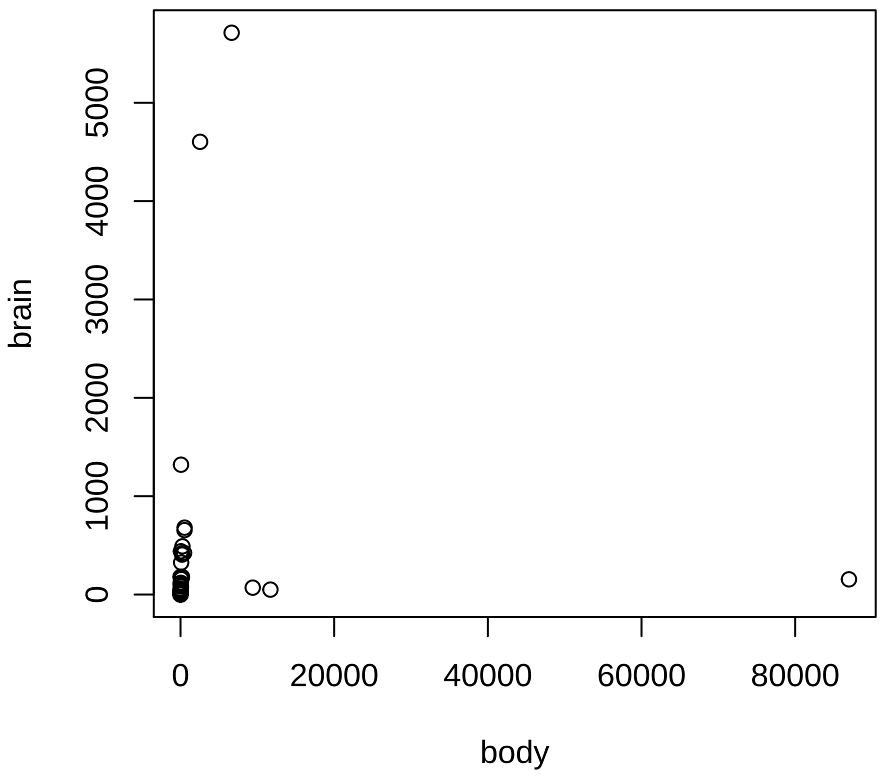
The first five observations,

```r
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
```

followed by a plot.

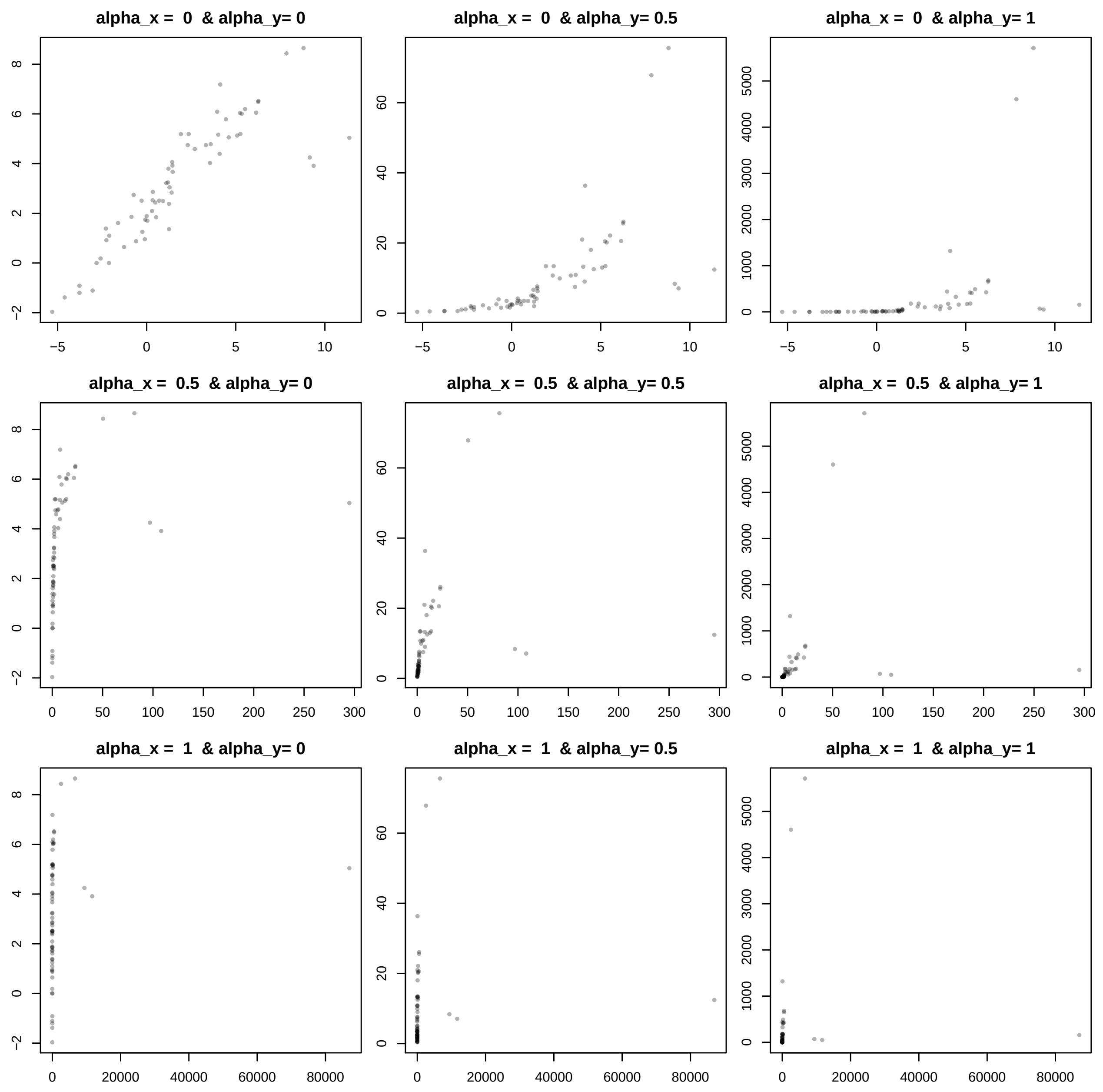
```r
plot(Animals2)
```



- Given the scatterplot above, can you see any relationship between the two variables body weight and brain weight?
- The population is fairly skewed. A transformation might help.
- Which direction should we go with the powers?

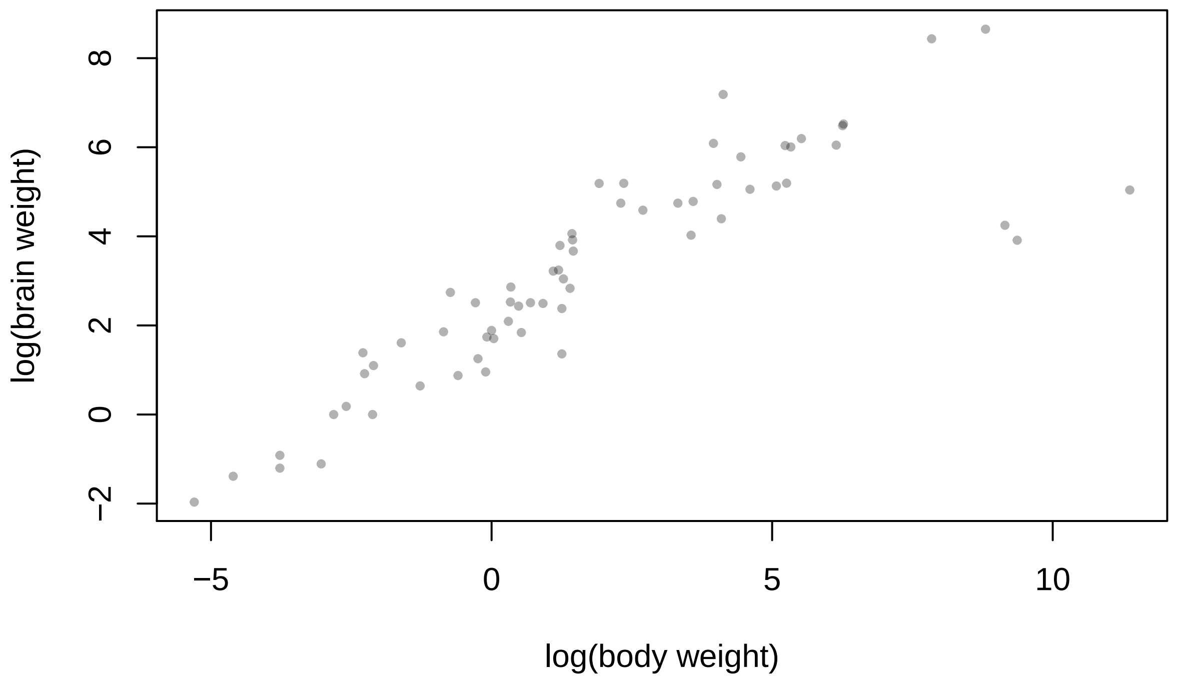
- Let us look at the transformed data under different transformations:

```
powerfun <- function(x, alpha) {
  if(sum(x <= 0) > 1) stop("x must be positive")
  if (alpha == 0)
    log(x)
  else if (alpha > 0) {
    x^alpha
  } else -x^alpha
}

par(mfrow=c(3,3), mar=2.5*c(1,1,1,0.1))
a = rep(c(0,1/2,1),each=3)
b = rep(c(0,1/2,1),times=3)

for (i in 1:9) {
plot( powerfun(Animals2$body, a[i]), powerfun(Animals2$brain, b[i]), pch = 19, cex=0.5,
     col=adjustcolor("black", alpha = 0.3), xlab = "", ylab = "",
     main = paste('alpha_x = ', a[i] ,' & alpha_y=', b[i] ) )
}
```



- Can we apply the rules for power transformations?
- Which transformation do you prefer?

```
##  The preferred transformation is
##  log(Brain) versus log(Body) weight

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)')
```



- Now, let's fit a linear regression model to the data and study the effect of the extreme values on the regression line.
- First, we write the objective function in a more general form

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

- By varying the function $\rho(\cdot)$ we can produce different fitted lines.
  - Least Squares Regression $\rho(r) = r^2$
  - Weighted Least Squares Line:

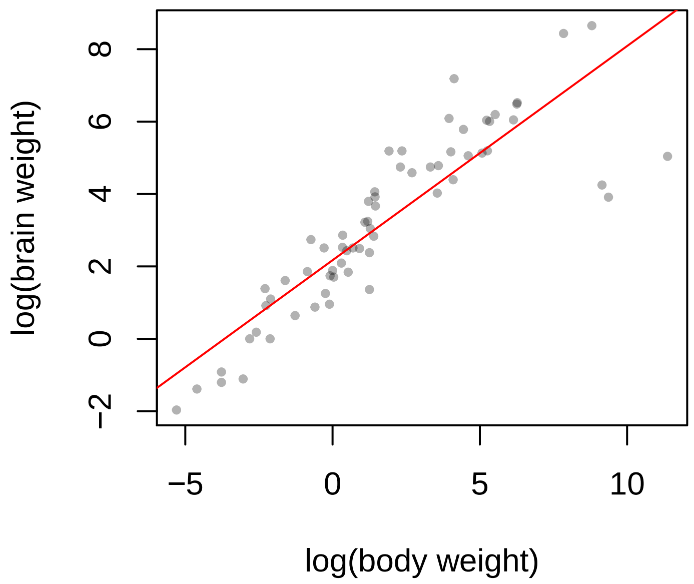
$$(\widehat{\alpha}, \widehat{\beta}) = \arg \min_{(\alpha,\beta) \in \mathbb{R}^2} \sum_{u \in \mathcal{P}} w_u \left[y_u - \alpha - \beta(x - c)\right]^2$$

  - Least Absolute Deviations Line:

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

- Fit a linear model to the transformed variates.

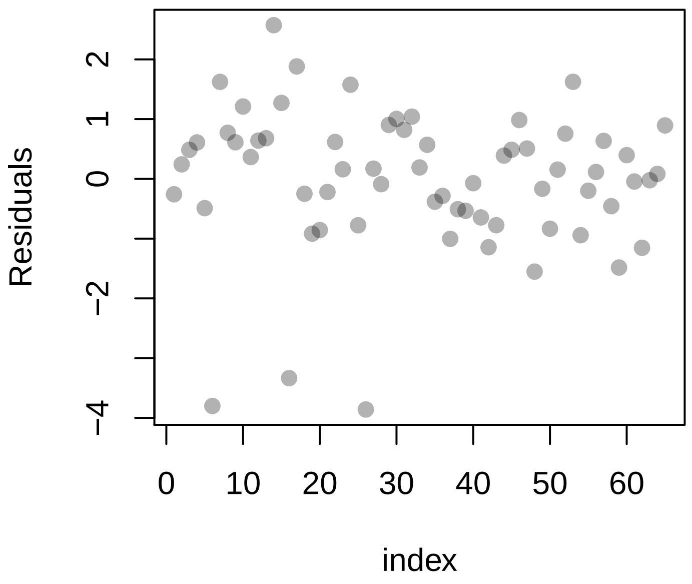
```r
data(Animals2, package = "robustbase")
plot(log(Animals2$body), log(Animals2$brain), pch = 19, cex=0.5,
     col=adjustcolor("black", alpha = 0.3),xlab='log(body weight)',ylab='log(brain weight)')
abline( lm(log(Animals2$brain) ~ log(Animals2$body)), col=2 )
```



- Does the least squares (LS) regression line do a good job at representing this relationship?

- Let’s check out the residuals

```r
data(Animals2, package = "robustbase")

mod = lm(log(Animals2$brain) ~ log(Animals2$body))

plot( residuals(mod), col=adjustcolor("black", alpha = 0.3),
      ylab='Residuals', xlab='index', pch=19 )
```



- It seems the residuals behave differently for certain units.

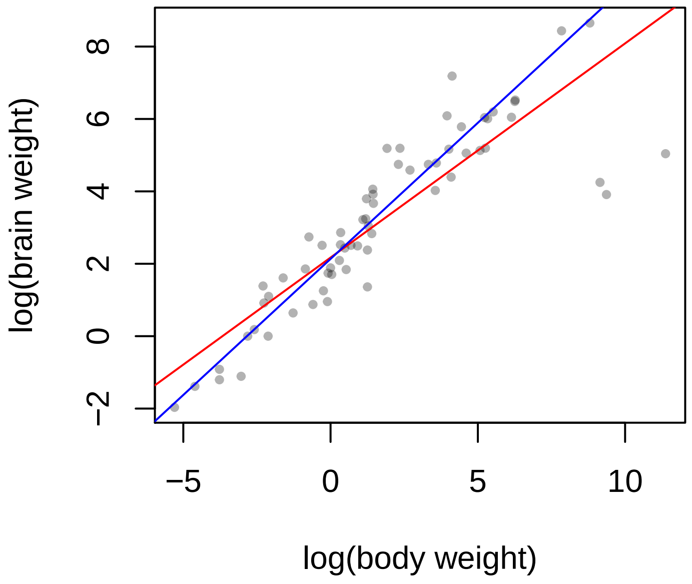
- We can either

– remove the units, or
– assign weights to the observations according to their variation, or
– use a method to find the regression line which is *robust* to potential outliers.

Removing the outliers

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

```
data(Animals2, package = "robustbase")

mod = lm(log(Animals2$brain) ~ log(Animals2$body))

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( mod, col=2 )

indx = which(log(Animals2$body)>9)
W = rep(1,65)
W[indx]=0
abline( lm(log(Animals2$brain) ~ log(Animals2$body),weights=W), col=4 )
```



- This seems like a better representation of the population.

Note that the visually detected outliers are the following observations

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

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

Examining the data we see that they are

```
round(log(Animals2[indx,]),2)
```

```
##                body brain
## Dipliodocus    9.37  3.91
## Triceratops    9.15  4.25
## Brachiosaurus 11.37  5.04
```

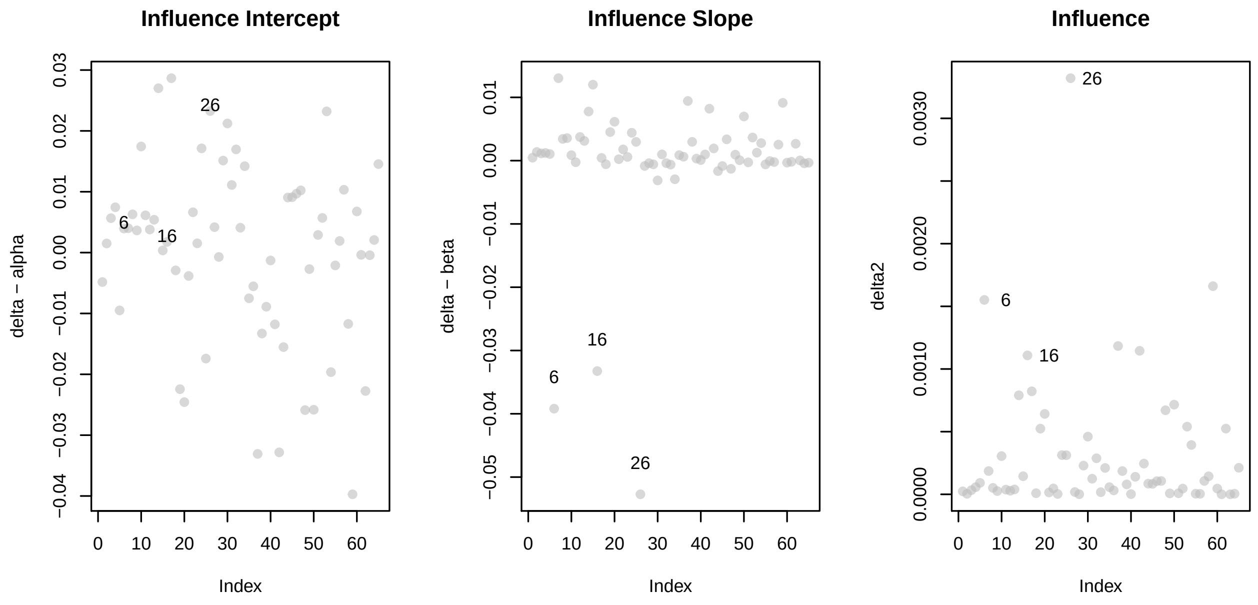
- These three data points are dinosaurs!
  - The rest of the data-set are land mammals.
  - Relative to other land mammals, dinosaurs have significantly larger bodies relative to their brain weights!

```
par(mfrow=c(1,3))

myimages<-list.files("../img/dino/", pattern = ".jpg", full.names = TRUE)
include_graphics(myimages)
```

- Let us look at the influence of the points on the slope and the intercept of the LS regression line.

```
model.pop <- lm(log(Animals2$brain) ~ log(Animals2$body))
theta.hat  <- model.pop$coef

N = nrow(Animals2)
delta = matrix(0, nrow=N, ncol=2)

for(i in 1:N){
  temp.model = lm(brain ~ body, data = log(Animals2[-i,]) )
  delta[i, ] = theta.hat-temp.model$coef
}

par(mfrow=c(1,3))
plot(delta[,1], ylab="delta - alpha", main="Influence Intercept",
     pch=19, col=adjustcolor("grey", 0.6) )

obs = c(6,16,26)
text(obs, delta[obs,1]+ 0.001 , obs)

plot(delta[,2], ylab="delta - beta", main="Influence Slope",
     pch=19, col=adjustcolor("grey", 0.6) )
```

```r
obs = c(6,16,26)
text(obs, delta[obs,2]+ 0.005 , obs)

delta2 = apply(delta,1, function(z) { sum(z^2) } )
plot(delta2, main="Influence",
      pch=19, col=adjustcolor("grey", 0.6) )

text(obs+5, delta2[obs], obs)
```



- Note these points are influential on the slope but not on the intercept.

- Maybe we should reweigh these units?

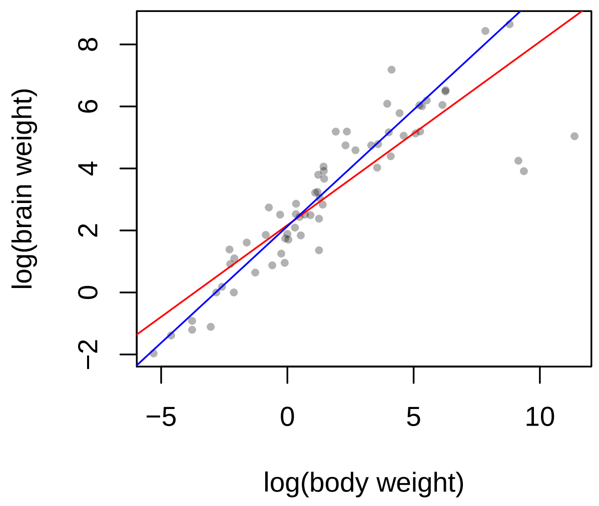
Weighted Least Squares

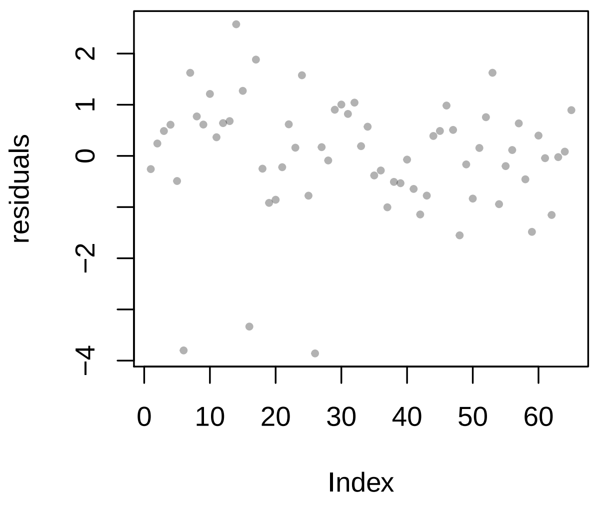
The weighted Least Squares line minimizes

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

- It is assumed the weights are known but
  - here we will estimate them using the residuals.

```r
data(Animals2, package = "robustbase")
log.Animals2 = log(Animals2)

par(mfrow=c(1,2))
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( brain ~ body, data= log.Animals2), col=2 )

indx = which(log(Animals2$body)>9)
W = rep(1,65)
W[indx]=0
abline( lm(brain ~ body, data= log.Animals2, weights=W), col=4)
#text(5,1,"Red: LS line",col="Red")
#text(5,0,"Blue: LS line without 3 points",col="blue")

ru = residuals(lm( brain ~ body, data= log.Animals2))
plot( ru,
      pch = 19, cex=0.5, col=adjustcolor("black", alpha=0.3),
      ylab="residuals")
```





- Some units have different variation than others.
- The variation of the outliers and remaining points is

```
obs = c(6,16,26)
ru = residuals(lm( brain ~ body, data= log.Animals2))

outlier.sd = sqrt(sum(ru[obs]^2)/(length(obs)))
remaining.sd = sqrt(sum(ru[-obs]^2)/(length(ru[-obs])))

c(outlier.sd, remaining.sd )
```

```
## [1] 3.6723598 0.8626269
```

Their relative variation is

```
remaining.sd/outlier.sd
```

```
## [1] 0.2348972
```

- Since the dinosaurs have larger variation we should give them lower weight equal to roughly 0.235

- Then we want the weighted Least Squares line which minimizes

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

  - where $w_u = 1$ for mammals and $w_u = 0.235$ for dinosaurs.

- Here we might set c to the weighted average, i.e.

$$c = \overline{x}_w = \frac{\sum_{u \in \mathcal{P}} w_u x_u}{\sum_{u \in \mathcal{P}} w_u}$$

- To find the parameter values for weighted least squares
  - We take the derivative with respect to each parameter. Then set this gradient to zero to obtain the system of equations,

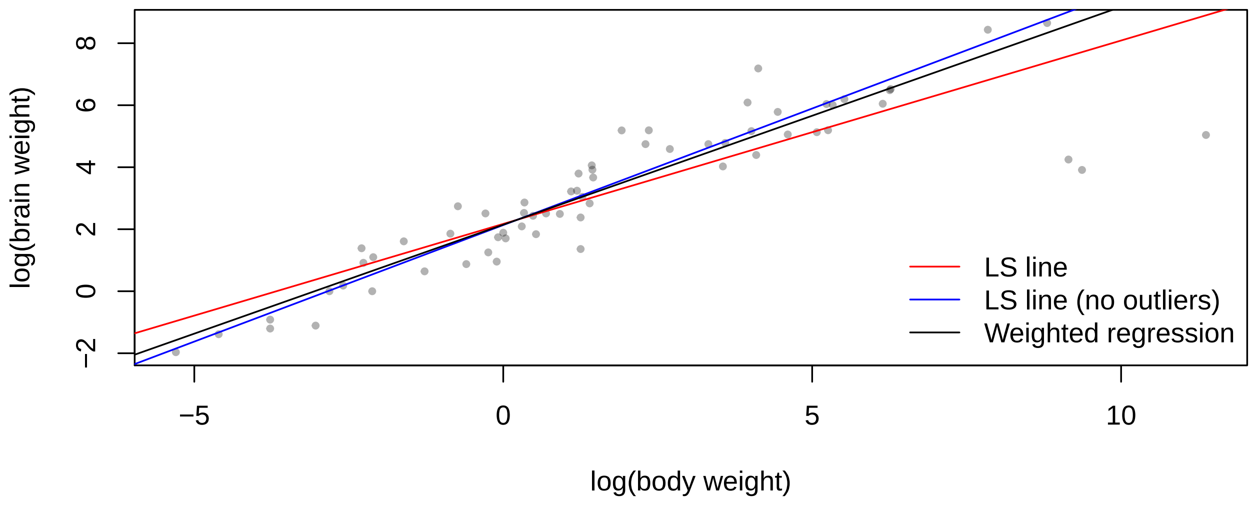
$$\sum_{u \in \mathcal{P}} w_u (y_u - \alpha - \beta(x_u - c)) \begin{pmatrix} 1 \\ \\ x_u - c \end{pmatrix} = \mathbf{0}$$

  - Then the parameter values are given by

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

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

```r
data(Animals2, package = "robustbase")
log.Animals2 = log(Animals2)

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( brain ~ body, data= log.Animals2), col=2 )

indx = which(log(Animals2$body)>9)
W = rep(1,65)
W[indx]=0
abline( lm(brain ~ body, data= log.Animals2, weights=W), col=4)

wt = rep(1,65)
wt[obs] = round(remaining.sd/outlier.sd, 3)
abline( lm(brain ~ body, data= log.Animals2, weights=wt), col=1)

legend( "bottomright", col=c(2,4,1), lty=1,
        legend=c("LS line", "LS line (no outliers)", "Weighted regression"),
        bty='n' )
```



- We used the variation to give less weight to certain points
- Which line represents the relationship in the population?
  - It would be nice to have more mathematical or automatic procedure to do this.

Robust Regression

- The third suggestion next to removing outliers or weighing them is a *robust* procedure to fit the regression line.
- For least squares regression we have that the attribute of interest is

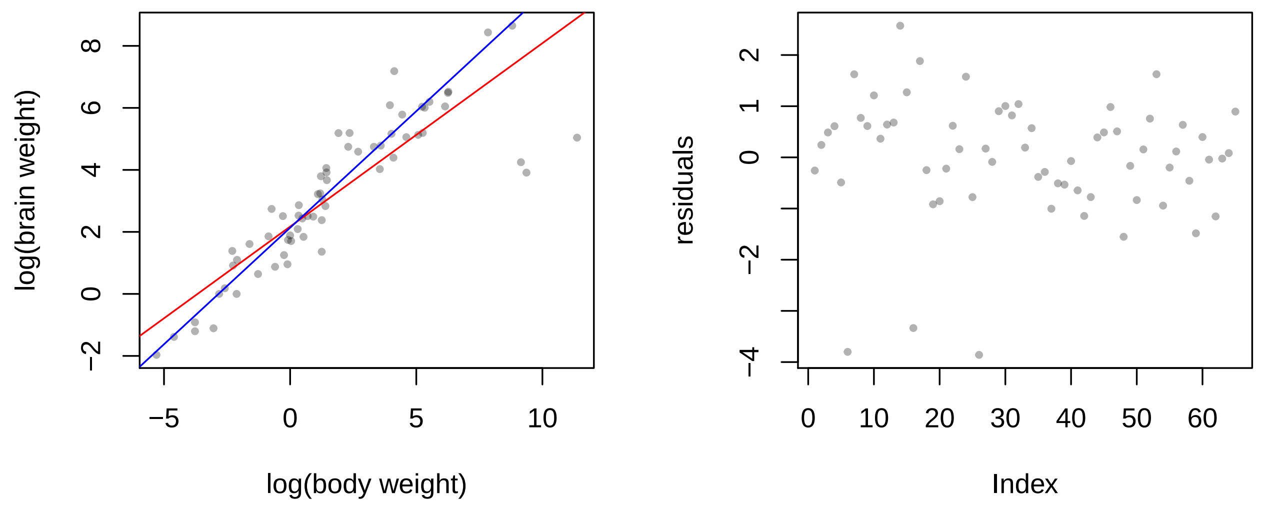
$$\begin{aligned}(\widehat{\alpha}, \widehat{\beta}) &= \arg \min_{(\alpha,\beta) \,\in\, \mathbb{R}^2} \sum_{u \,\in\, \mathcal{P}} (y_u - \alpha - \beta[x-c])^2 \\ &= \arg \min_{(\alpha,\beta) \,\in\, \mathbb{R}^2} \sum_{u \,\in\, \mathcal{P}} \rho\left(y_u - \alpha - \beta[x-c]\right)\end{aligned}$$

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

- We have a population with some unusual units.

```
data(Animals2, package = "robustbase")
log.Animals2 = log(Animals2)

par(mfrow=c(1,2))
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( brain ~ body, data= log.Animals2), col=2 )

indx = which(log(Animals2$body)>9)
W = rep(1,65)
```

```
W[indx]=0
abline( lm(brain ~ body, data= log.Animals2, weights=W), col=4)

ru = residuals(lm( brain ~ body, data= log.Animals2))
plot( ru,
      pch = 19, cex=0.5, col=adjustcolor("black", alpha=0.3),
      ylab="residuals")
```



Another solution to the outlier problem noticed above is to change the $\rho(r)$ function so that

- it gives lower weight than LS to r_u far from 0 so that large residuals have lower weight.
- and $\rho(r)$ should be quadratic near 0 so that it is similar to least squares for small residuals.

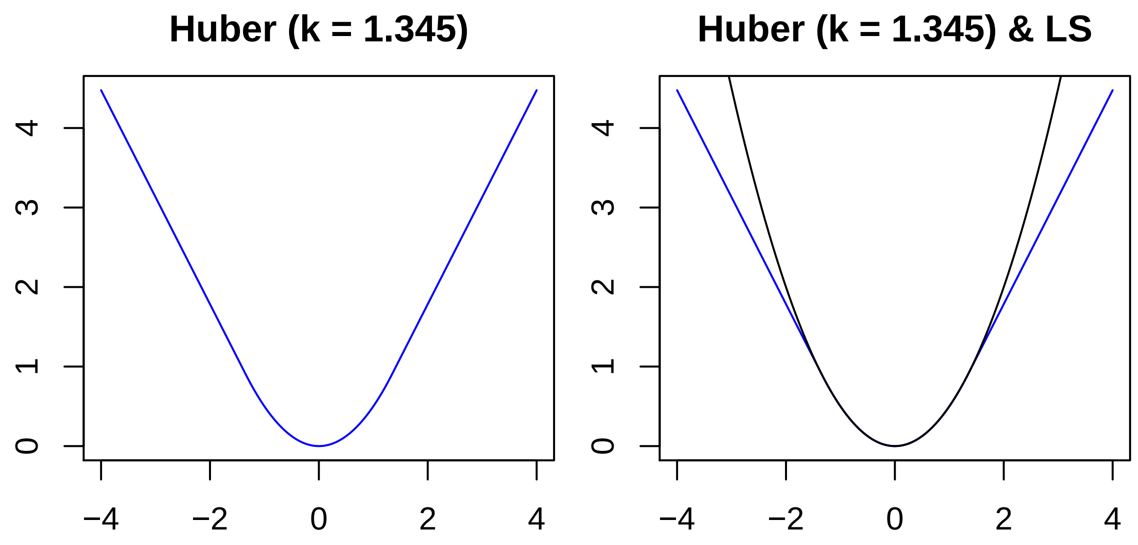
Huber Function

- Mix of the quadratic and absolute value functions.

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

```
huber.fn <- function(r, k = 1.345) {
  val = r^2/2
  subr = abs(r) > k
  val[ subr ] = k*(abs(r[subr]) - k/2)
  return(val)
}

r = seq(-4,4,.01)
par(mfrow=c(1,2), mar=2.5*c(1,1,1,0.1))

plot( r, huber.fn(r), type='l', col=4, main="Huber (k = 1.345)")
```

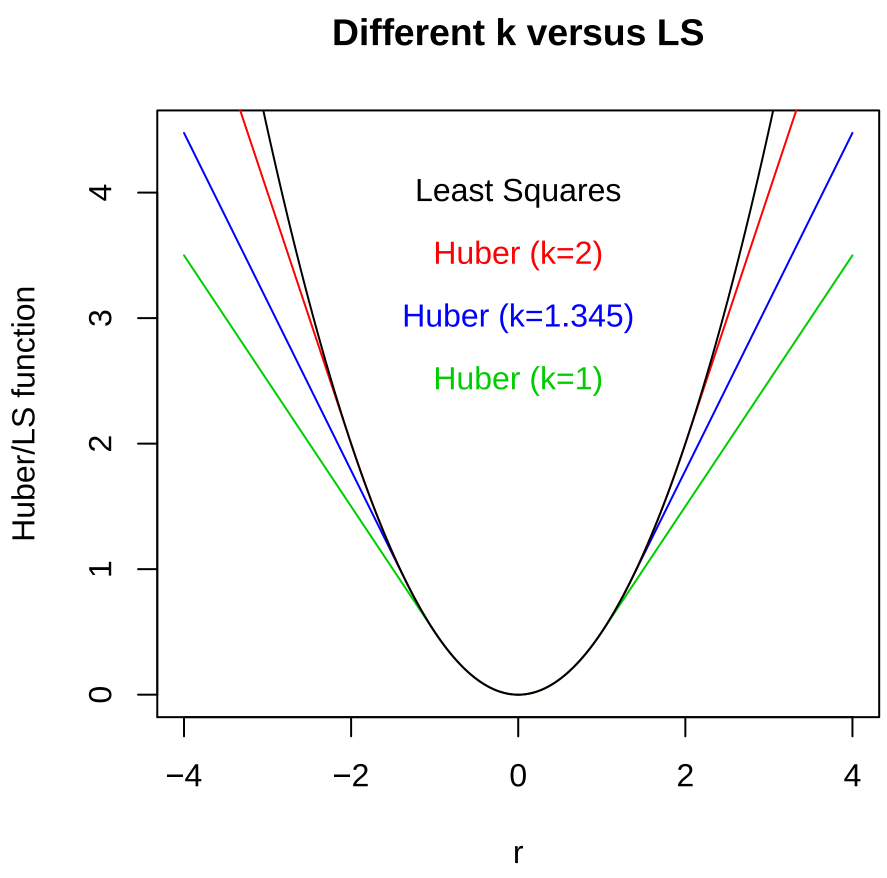
```r
plot( r, huber.fn(r), type='l', col=4, main="Huber (k = 1.345) & LS")
lines( r, r^2/2, type='l')
```



```r
#lines(r, huber.fn(r,k=2), type='l',col=2)
```

- An attribute (regression coefficients) 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.
- We set $k \approx 1.345\ S$ to satisfy a theoretical balance between
  - efficiency and
  - resistance to outliers.
  - Other choices can be $k = 1.5$ or 2.
  - As k increases, the robust regression with Huber function imposes a larger penalty on larger residuals r_u, hence it gets closer to the LS fit.

```r
r = seq(-4,4,.01)
plot( r, huber.fn(r,k=1.345), type='l', col=4, main="Different k versus LS" , ylab = "Huber/LS function
lines(r, huber.fn(r,k=2), type='l',col=2)
lines(r, huber.fn(r,k=1), type='l',col=3)
lines( r, r^2/2, type='l')
text(0,4,"Least Squares", bty = "n")
text(0,3.5,"Huber (k=2)",
       bty = "n",col=2)
text(0,3,"Huber (k=1.345)",
       bty = "n",col=4)
text(0,2.5,"Huber (k=1)",
       bty = "n" ,col=3)
```



Example - Animals (body vs. brain weight)

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

```
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 )

indx = which(log(Animals2$body)>9)
W = rep(1,65)
W[indx]=0
abline( lm(log(Animals2$brain) ~ log(Animals2$body),weights=W), col=4 )

library(MASS)
```

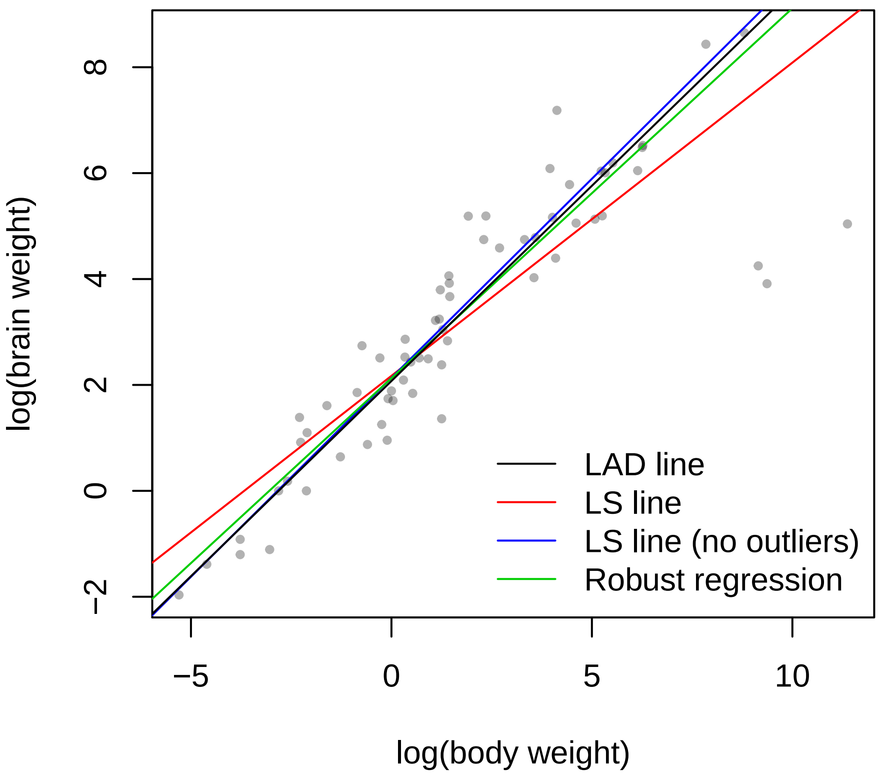
```
## Warning: package 'MASS' was built under R version 3.4.3
```

```
abline( rlm(log(Animals2$brain) ~ log(Animals2$body), psi="psi.huber"), col=3)
```

```r
library(L1pack)
```

```
## Warning: package 'L1pack' was built under R version 3.4.2
```

```r
abline( lad(log(Animals2$brain) ~ log(Animals2$body)), col=1)

legend( "bottomright", col=c(1,2,4,3), lty=1,
        legend=c("LAD line", "LS line", "LS line (no outliers)", "Robust regression"),
      bty='n' )
```



- The LAD (least absolute deviance) line uses the function

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

- But how to find $(\widehat{\alpha}, \widehat{\beta})$ when closed form solutions don't exist due to the choice of $\rho(r)$ function?

Robust Estimation

- Our attribute of interest is

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

- To estimate the parameters for any implicitly defined parameter such as those in Robust Regression, we consider the following optimization methods:
  - Gradient descent,
  - Iteratively reweighted least-squares,
  - Newton-Raphson.

- The algorithms above are employed to handle implicitly defined attributes defined as

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

2.3.1.1 Gradient Descent

The goal is to construct an **iterative procedure** which produces a sequence of **iterates**

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

such that this sequence converges to the solution $\widehat{\boldsymbol{\theta}}$.

- Ideally, each iteration 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 the value of $\rho(\widehat{\boldsymbol{\theta}}_i)$.

Search Direction

- If $\rho(\boldsymbol{\theta})$ is a **differentiable** function of $\boldsymbol{\theta}$ then we can 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 the gradient to unit length provides the **direction** in which $\rho(\boldsymbol{\theta})$ rises, or ascends, fastest.

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

  - The normalized gradient vector $\mathbf{d}_i$ is now the direction of steepest **ascent** and then $-\mathbf{d}_i$ is the direction of **steepest descent**.
  - Recall that $||\mathbf{g}_i|| = \sqrt{\sum_j g_{ij}^2}$, where i denotes calculating $\mathbf{g}$ at θ_i, and $j = 1, ..., k$ is the dimension of $\mathbf{g_i}$.

- We obtain a new estimate or the next iteration by
  - moving in the direction of $-\mathbf{d}_i$ and
  - taking a step of length $\lambda_i > 0$

$$\widehat{\boldsymbol{\theta}}_{i+1} = \widehat{\boldsymbol{\theta}}_i - \lambda_i \ \mathbf{d}_i.$$

- Note we could work with $\mathbf{g}_i$ instead of $\mathbf{d}_i$

Line Search

- The next iteration is based on

$$\widehat{\boldsymbol{\theta}}_{i+1} = \widehat{\boldsymbol{\theta}}_i - \lambda_i \ \mathbf{d}_i.$$

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

$$\rho\left(\widehat{\boldsymbol{\theta}}_i - \lambda_i \ \mathbf{d}_i\right)$$

  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**.

- Alternatively, we could set the step λ_i to be a
  - fixed value such as $\lambda_i = 0.1$
  - fixed sequence such as $\lambda_i = 0.1 + 1/i$

Gradient Descent Algorithm

Given some initial estimate $\widehat{\boldsymbol{\theta}}_0$

1. **Initialize** ; $i \leftarrow 0$;
2. LOOP:

   a. **Gradient**:

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

   b. Gradient **direction**:

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

c. **Line search**: Find the step size $\widehat{\lambda}_i$

$$\widehat{\lambda}_i = \arg\min_{\lambda > 0} \rho\left(\widehat{\boldsymbol{\theta}}_i - \lambda\ \mathbf{d}_i\right)$$

d. **Update** the iterate:

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

e. **Converged?**

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

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

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

R code for Gradient Descent

- For production code we would do more error checking
  - We are avoiding this here to better illustrate the algorithm.

```r
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)
       )
}
```

- Note that the gradient descent algorithm is specified above even though none of the functions it requires exist!
  - These will need to be implemented for each particular function $\rho(\cdots)$ we wish to minimize.

R code for Line Search and 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
}
```

Examples

Quadratic Function (one-dimensional)

Suppose for example that we were finding that θ which minimized

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

Then the gradient is

$$g = 4\theta - 5$$

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}
```

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

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

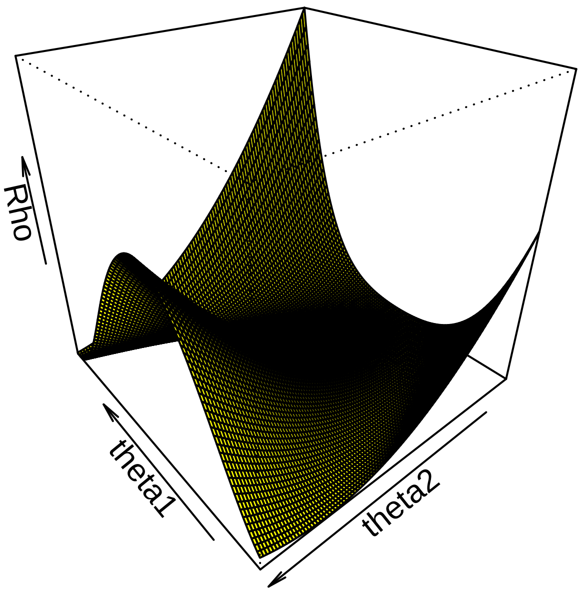
```
## $theta
## [1] 1.25
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 4
##
## $fnValue
## [1] -0.125
```

Rosenbrock function (two-dimensional)

We want to find $\boldsymbol{\theta}$ which minimizes the following Rosenbrock function

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

```
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 is a non-convex function, usually used in performance test problems 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

```r
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

```r
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
```

Least Squares

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

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

and the gradient is

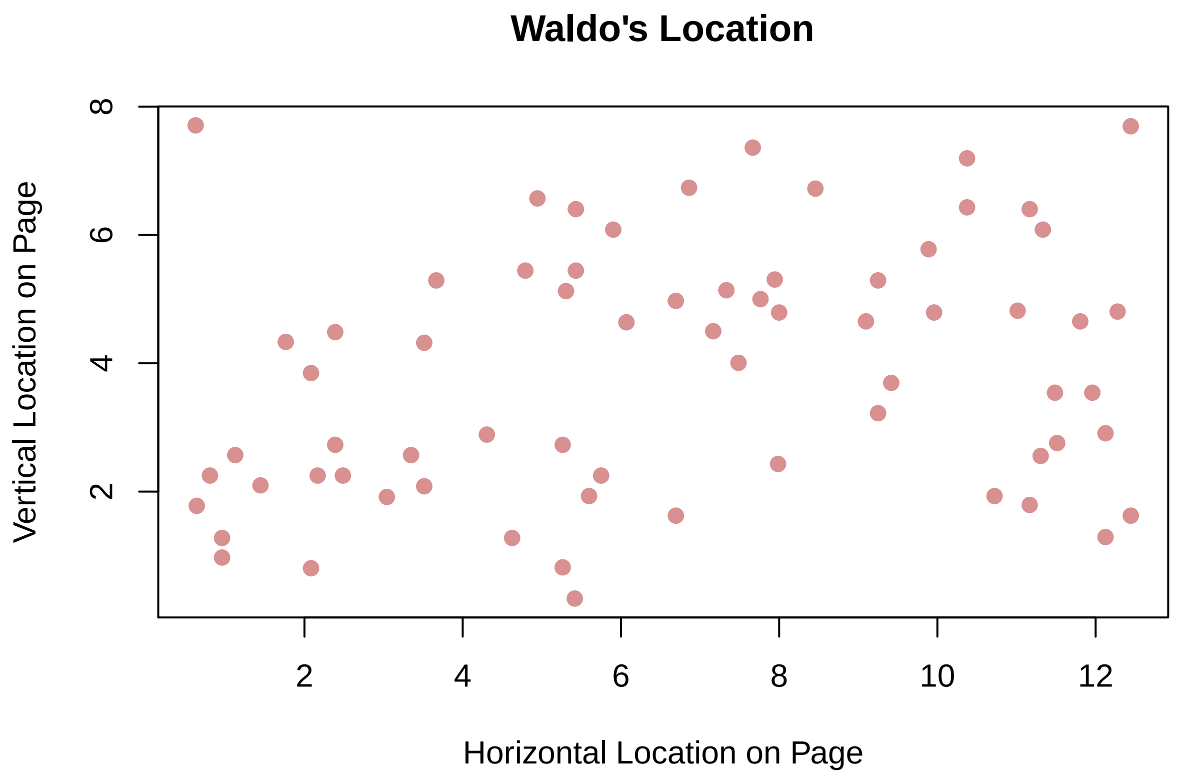
$$\begin{aligned}
\mathbf{g}_i &= \left(\begin{array}{c} \sum_{u \in \mathcal{P}} -2(y_u - \alpha - \beta(x_u - \overline{x})) \\ \\ \sum_{u \in \mathcal{P}} -2(y_u - \alpha - \beta(x_u - \overline{x}))(x_u - \overline{x}) \end{array} \right)_{\alpha = \widehat{\alpha}_i \ ; \ \beta = \widehat{\beta}_i} \\
\\
&= -2 \left(\begin{array}{c} N \ (\overline{y} - \widehat{\alpha}_i) \\ \\ \sum_{u \in \mathcal{P}} (x_u - \overline{x}) \ y_u \ - \ \widehat{\beta}_i \sum_{u \in \mathcal{P}} (x_u - \overline{x})^2 \end{array} \right).
\end{aligned}$$

For a least-squares line, the ρ function depends on the values of the variates x and y. Let's look an example data-set.

LS for Where's Waldo?

For illustration, we can take $\mathcal{P}$ to be the set of `Where's Waldo?` pages and x and y to be the horizontal and vertical locations of `Waldo`.

```
directory = "../Data"
dirsep = "/"
waldofile <- paste(directory, "WheresWaldo", "wheres-waldo-locations.csv", sep=dirsep)
waldo <- read.csv(waldofile)
plot(waldo$X, waldo$Y,
     pch=19, col=adjustcolor("firebrick", 0.5),
     main = "Waldo's Location",
     xlab="Horizontal Location on Page", ylab= "Vertical Location on Page")
```



The 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) {
```

```r
  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**
  - 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 `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

- An obvious solution, and one which generally works, is to pass the needed variables (here `x` and `y`) to the function as arguments.
  - Unfortunately, for our purposes, this would clutter up the code somewhat and necessitate rewriting the nice and very general gradient descent function.
  - A better way to proceed, which still keeps clutter to a minimum, is to write functions which will return the appropriate functions.

- 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.

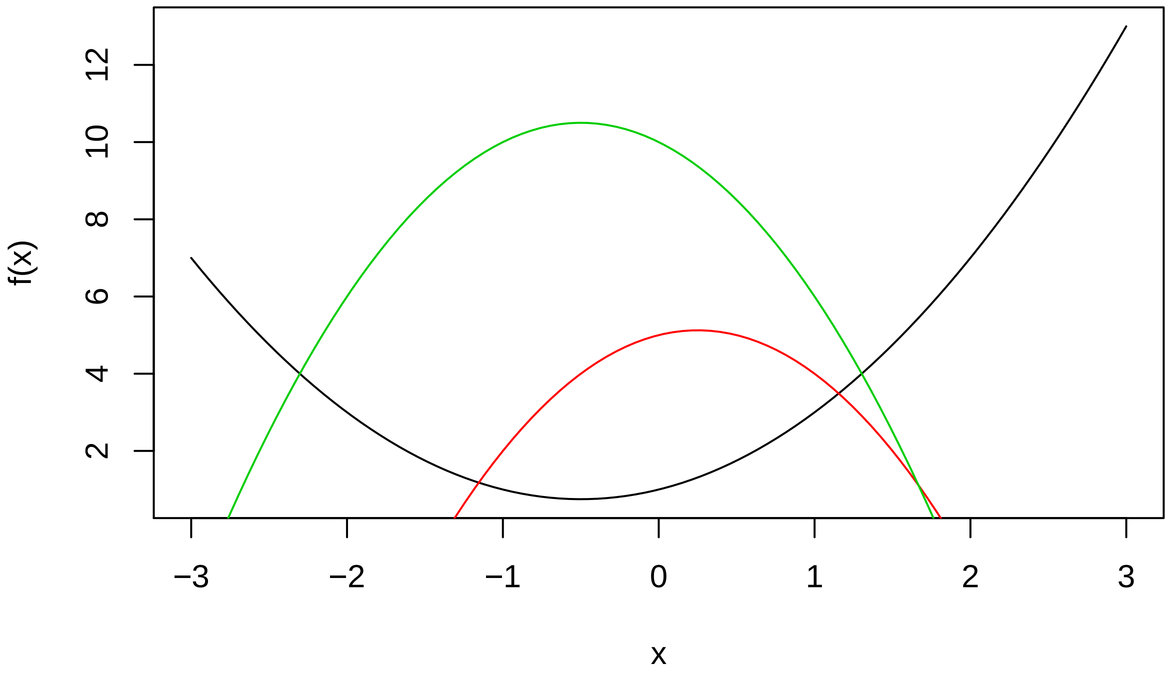
A function that returns a function

A simple example

```
createQuadratic <- function(a=1,b=1,c=1) {
  ## local variable
  xbar <- mean(x)
  ## Return this function
  function(x) {
    fx = a*x^2 + b*x + c
    return(fx)
  }
}
```

Our function creating function in action

```
x = seq(-3,3, length.out=100)

f1 = createQuadratic(a=+1, b=1, c=1)
f2 = createQuadratic(a=-2, b=1, c=5)
f3 = createQuadratic(a=-2, b=-2, c=10)

plot(x, f1(x), type='l', ylab='f(x)')
lines(x, f2(x), col=2)
lines(x, f3(x), col=3)
```



- Note that if one changes the vector `x` in the code above **after** `f1`, `f2`, and `f3` are defined, the result for the plots will be unchanged as these functions do not point to the global variable `x`.

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

```r
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)
```

These functions now have access to their own data as the values of `x` and `y`. The creator functions allow us to simply implement gradient descent for least squares for any `x` and `y`.

Back to `Where's Waldo` Example

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

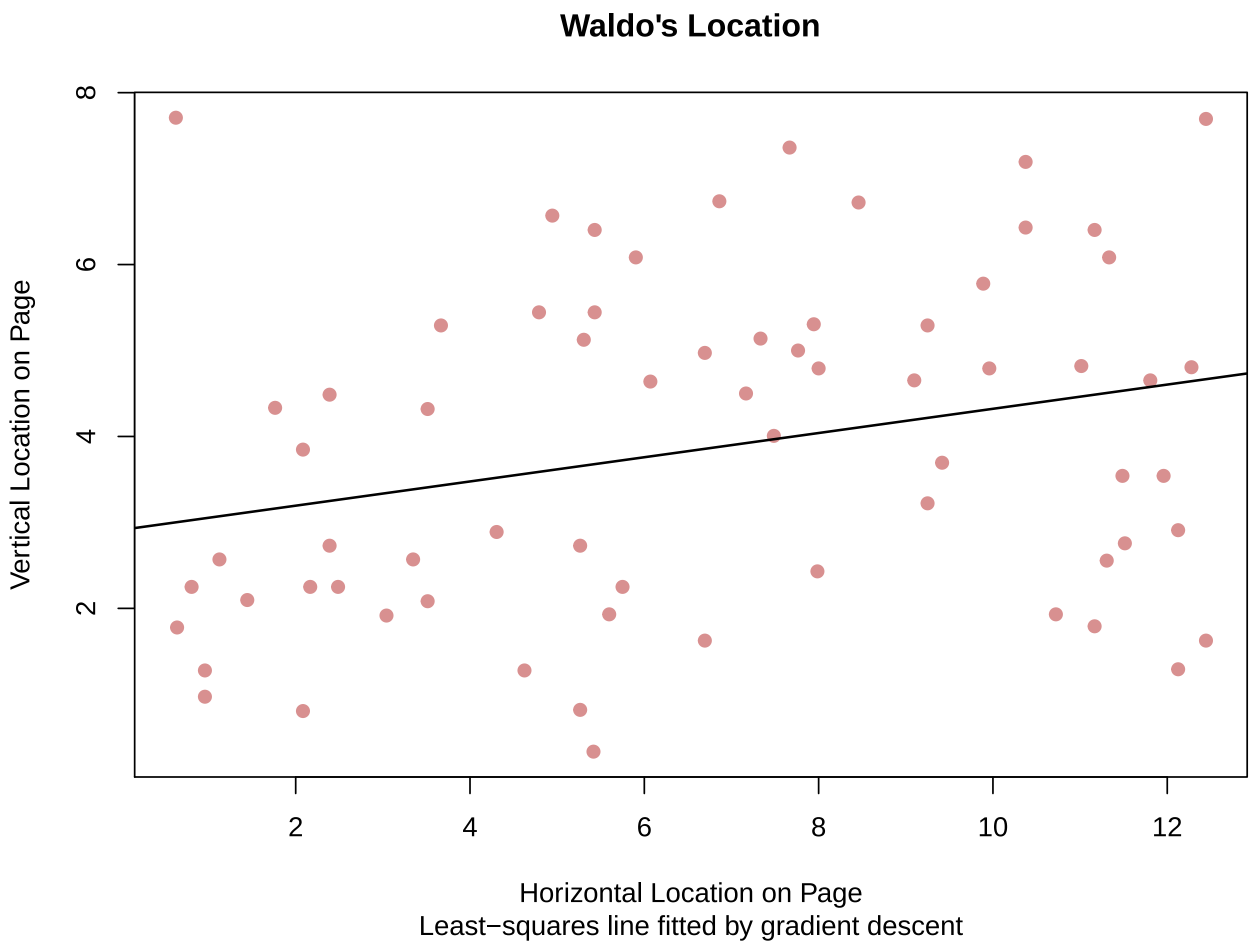
```r
result <- gradientDescent(theta = c(0,0),
                          rhoFn = rho, gradientFn = gradient,
                          lineSearchFn = gridLineSearch,
                          testConvergenceFn = testConvergence)
### Print the results
Map(function(x){if (is.numeric(x)) round(x,3) else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

```
## $theta
## [1] 3.857 0.141
```

```
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 29
##
## $fnValue
## [1] 233.774
```

And we can plot the resulting line for the Where's Waldo population:

```
plot(waldo$X, waldo$Y,
     pch=19, col=adjustcolor("firebrick", 0.5),
     main = "Waldo's Location", sub = "Least-squares line fitted by gradient descent",
     xlab="Horizontal Location on Page", ylab= "Vertical Location on Page")
abline(result$theta[1] - result$theta[2] * mean(waldo$X), result$theta[2], lwd=1.5)
```



The line might be used, for example, to guide the eyes across any two page spread in a *Where's Waldo?* book to help finding him. That said, a more complex summary than a simple line might be better.

Robust Regression Gradient Descent

The attribute of interest is

$$\rho(\boldsymbol{\theta}) = \rho(\alpha, \beta) = \sum_{u \in \mathcal{P}} \rho_k(y_u - \alpha - \beta(x_u - \overline{x})) = \sum_{u \in \mathcal{P}} \rho_k(r_u)$$

where $r_u = y_u - \alpha - \beta(x_u - \overline{x})$ in which

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

- The gradient can be decomposed as

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

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

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

- Now, if we let

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

  - then the gradient can be written as

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

R code for Robust Regression Gradient Descent

The Huber function

```r
huber.fn <- function(r, k = 1.345) {
  val = r^2/2
  subr = abs(r) > k
  val[ subr ] = k*(abs(r[subr]) - k/2)
  return(val)
}
```

A function to create the objective function

```r
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 ) )
  }
}
```

The derivative of the Huber function

```r
huber.fn1 <- function(r, k = 1.345) {
  val = r
  subr = abs(r) > k
  val[ subr ] = k*sign(r[subr])
  return(val)
}
```

A function to create the gradient function

```r
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

R code to do Robust regression using the Huber function

```r
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
```

```
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
```

We can compare our results to the 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
```

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

(Batch) Gradient Descent

When the objective function has the form

$$\rho(\boldsymbol{\theta}, \mathcal{P}) = \sum_{u \in \mathcal{P}} \rho(\boldsymbol{\theta}; u) \qquad \text{or equivalently} \qquad \rho(\boldsymbol{\theta}, \mathcal{P}) = \frac{1}{N} \sum_{u \in \mathcal{P}} \rho(\boldsymbol{\theta}; u)$$

then the gradient descent algorithm is sometimes called **batch gradient descent**. Then the algorithm can be tailored to some important (big data) applications.

- The gradient takes a similar form as the objective function.

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

- In this case the value of λ is sometimes called the **learning rate**.
  - Typically, the step size or learning rate is fixed.
  - However, adaptive learning rates are possible.

Some Questions/Observations:

- How is this different from **plain** gradient descent?

- Why is this called batch gradient descent?

  - If the population is so large that we cannot storage the data in one place. i.e. the population is constructed of H batches (groups) each with M units, $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_H\}$, then the objective function can be written as

$$\rho(\boldsymbol{\theta}, \mathcal{P}) = \frac{1}{N} \sum_{u \in \mathcal{P}} \rho(\boldsymbol{\theta}; u) = \frac{M}{N} \sum_{k=1}^{H} \frac{1}{M} \sum_{u \in \mathcal{P}_k}^{M} \rho(\boldsymbol{\theta}; u)$$

  - In addition the gradient can be written as

$$\mathbf{g}_i = \frac{1}{N} \sum_{u \in \mathcal{P}} \mathbf{g}_i(u) = \frac{M}{N} \sum_{k=1}^{H} \mathbf{g}_{i,k} \qquad \text{where} \qquad \mathbf{g}_{i,k} = \frac{1}{M} \sum_{u \in \mathcal{P}_k} \mathbf{g}_i(u)$$

  - The H subgroups can be formed deterministically or randomly.
  - **Note**, we could form groups of size $M = 1$.

- One issue is that if the population $\mathcal{P}$ is very large (big data), computing the gradient or objective function can be computationally expensive.
  - Since the data is stored separately we could use parallel computation and distribute the M batches to a cloud of computers.

- One possible improvement when the population $\mathcal{P}$ is large is to used Mini-Batch Stochastic Gradient Descent.
  - At every iteration we estimate the gradient
  - using a random sample or few batches.

- Question? At every iteration should we move sequentially through the batches or randomly choose a few batches at every iteration.

- In **Stochastic Gradient Descent**
  - The batches are size 1 and
  - the alogrithm moves sequentially through the population.
  - This also called **on-line** Stochastic Gradient Descent

- **Note** there are many many many verisions of Stochastic Gradient Descent.
  - Unfortunely, a lot of them use the same name.

Batch Gradient Descent Alogrithm

- The objective function and the gradient are sum over the population.

Given some initial estimate $\widehat{\boldsymbol{\theta}}_0$, the batch gradient descent algorithm is:

1. **Initialize** ; $i \leftarrow 0$;

2. LOOP:

   a. **Gradient**: this step is the only change compared to plain GD algorithm.

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

   b. Gradient **direction**:

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

   c. **Line search**: Find the step size $\widehat{\lambda}_i$

   $$\widehat{\lambda}_i = \arg\min_{\lambda > 0} \rho\left(\widehat{\boldsymbol{\theta}}_i - \lambda\ \mathbf{d}_i\right)$$

   d. **Update** the iterate:

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

   e. **Converged?**

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

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

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

Random Sample Stochastic Gradient Descent Alogrithm

- The objective function and the gradient are sum over the population.
- At every iteration we estimate the gradient using a random sample of size n.

Given some initial estimate $\widehat{\boldsymbol{\theta}}_0$ and fixed step size $\lambda^\star$ the algorithm is:

1. **Initialize** ; $i \leftarrow 0$;
2. LOOP:

   a. **Gradient**: Estimate gradient using a random sample $\mathcal{S}$ from the population $\mathcal{P}$

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

   b. Gradient **direction**:

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

   c. **Line search**: Find the step size $\widehat{\lambda}_i$

   $$\widehat{\lambda}_i = \arg\min_{\lambda > 0} \rho\left(\widehat{\boldsymbol{\theta}}_i - \lambda\ \mathbf{d}_i\right)$$

   d. **Update** the iterate:

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

```
    e. **Converged?**
       **if** the iterates are not changing, then **Return**
       **else** $i \leftarrow i+1$ and repeat LOOP.
3. **Return**: $\widehat{\boldsymbol\theta} = \widehat{\boldsymbol\theta}_i$
```

Mini-Batch Stochastic Gradient Descent Alogrithm

- The objective function and the gradient are sum over the population.
- Divide the population of size N to H batches of size M, i.e. $N = H \times M$ and $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_H\}$
- At every sub-iteration we estimate the gradient use the information from a single batch.

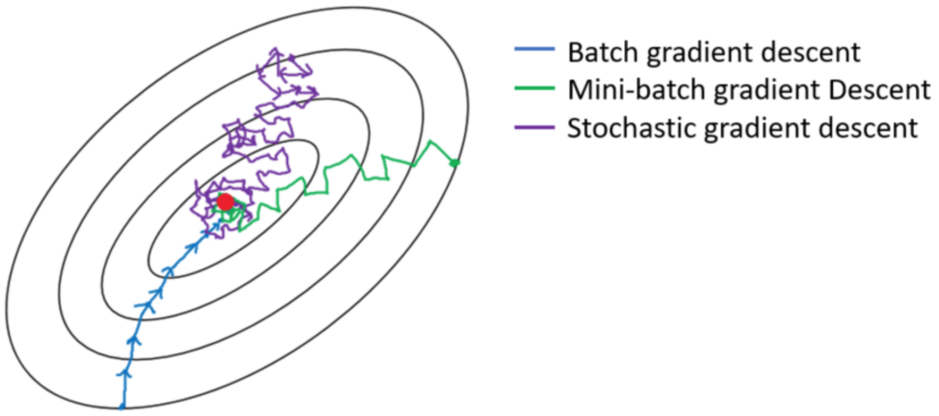
Given some initial estimate $\widehat{\boldsymbol\theta}_0$ and fixed step size $\lambda^\star$ the algorithm is:

```
1. **Initialize** ; $i \leftarrow 0$;
2. LOOP over $i$ ;
    a. Set $\widehat{\boldsymbol\theta}_{i,1} = \widehat{\boldsymbol\theta}_i$
    b. Repeat for $k = 1, \dots, H$;
        i) **Gradient** for group $\mathcal{P}_k$:
               $\mathbf{g}_{i,k} = \left.\frac{\partial\ \rho(\boldsymbol\theta; \mathcal{P}_k)}{\partial \boldsymbol\theta}\right|_{\boldsymbol\theta = \widehat{\boldsymbol\theta}_{i,k}}$
        ii) Gradient **direction** for group $\mathcal{P}_k$:
               $\mathbf{d}_{i,k} \leftarrow \frac{\mathbf{g}_{i,k}}{||\mathbf{g}_{i,k}||}$
        iii) **Update** using $\mathcal{P}_k$
               $\widehat{\boldsymbol\theta}_{i,k+1} \leftarrow \widehat{\boldsymbol\theta}_{i,k} - \lambda^\star\ \mathbf{d}_{i,k}$
    c. **Update** the iterate:
               $\widehat{\boldsymbol\theta}_{i+1} \leftarrow \widehat{\boldsymbol\theta}_{i,H}$
    d. **Converged?**
       **if** the iterates are not changing, then **Return**
       **else** $i \leftarrow i+1$ and repeat LOOP.
3. **Return**: $\widehat{\boldsymbol\theta} = \widehat{\boldsymbol\theta}_i$
```

- When a constant learning rate $\lambda^\star$
- **Note**; If $H = N$ or the batch size is $M = 1$ then is known as
    - stochastic gradient descent or
    - **on-line** gradient descent.

Comparing the 3 Algorithms

- Stochastic gradient descent is **on-line** gradient descent or mini-batch with batch size equal to $M = 1$.



Stochastic Gradient Descent - R code

- Do we need to change any code in the `gradientDescent` function?

```
## Nope
```

The `gradientDescent` function

```
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)
  )
}
```

- We have to edit the gradient function

```
createRobustHuberGradient.stochastic <- function(x,y, kval=1.5, nsize=1) {
  ## 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)

    subset = sample(x=length(rhok), size=nsize)

    -1*c( sum( rhok[subset] ), sum( (rhok*(x-xbar))[subset] ) )
  }
}
```

We can use fixed line search.

```
fixedLineSearch <- function(theta, rhoFn, g,
                           lambdaStepsize = 0.01,
                           lambdaMax = 1) {
  lambdaStepsize
}
```

Test of convergence using difference in the parameters

```
testConvergence <- function(thetaNew, thetaOld, tolerance = 1E-10, relative=FALSE) {
   sum(abs(thetaNew - thetaOld)) < if (relative) tolerance * sum(abs(thetaOld)) else tolerance
}
```

Test of convergence using difference in the gradient

```
createGradient.Convergence <- function(gradFn) {
  ## local variables
  function(thetaNew=NULL, thetaOld=NULL, tolerance = 1E-10, relative=FALSE) {
    sum(abs( gradFn(thetaNew) )) < tolerance
  }
}
```

- Putting all together.
- Load the data and set the hyper-parameter for the Huber function.

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

Create the objective, stochastic gradient and gradient function.

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

grad.stochatic <- createRobustHuberGradient.stochastic(
  log(Animals2$body), log(Animals2$brain),
  kval=animal.kval, nsize=1
  )

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

Create the convergence criteria function.

```
testFn <- createGradient.Convergence(gradient)
```

Perform stochastic gradient descent.

```
set.seed(341)
result2 <- gradientDescent(theta = c(1,0), rhoFn = rho,
      gradientFn = grad.stochatic,
      lineSearchFn = fixedLineSearch,
      testConvergenceFn = testFn,
      maxIterations=10^5, lambdaStepsize = 0.01, tolerance = 1 )
### Print the results
result2
```

```
## $theta
## [1] 3.3481444 0.6956716
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 36451
##
## $fnValue
## [1] 28.81961
```

From Batch Gradient Descent

```
result
```

```
## $theta
## [1] 3.3308999 0.6909751
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 20
##
## $fnValue
## [1] 28.82349
```

- **Excerise** examine the effect of changing the value of `nsize`. Any value `nsize>1` results in mini-batch SGD.

2.3.2 Solving a system of equations

- A population attribute might also be defined implicitly as the solution $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ that satisfies a system of equations
$$\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P}) = \mathbf{0}$$
  - 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 $\surd$ and
  - the attribute of interest $\widehat{\boldsymbol{\theta}}$ is the value $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ that solves
$$\sum_{u \in \mathcal{P}} \psi(\boldsymbol{\theta}; u) = \mathbf{0}.$$

Examples

- Our familiar examples for implicitly defined parameters
  - as solutions to mimimazation problems can also be defined as
  - as solution to systems of equations
- The average
$$\sum_{u \in \mathcal{P}} y_u - n\theta = 0$$
- The weighted sum
$$\sum_{u \in \mathcal{P}} w_u(y_u - \theta) = 0$$

- The quantiles for some $q \in (0,1)$ which involve finding the value of y such that

$$\frac{1}{N}\sum_{u\in\mathcal{P}} I\left(y_u \leq y\right) = q$$

  - We might also ask for the smallest y which satisfies this.

- Least squares

$$\sum_{u\in\mathcal{P}} (y_u - \alpha - \beta(x_u - c)) \begin{pmatrix} 1 \\ \\ x_u - c \end{pmatrix} = \mathbf{0}$$

- Weighted least squares

$$\sum_{u\in\mathcal{P}} w_u(y_u - \alpha - \beta(x_u - c)) \begin{pmatrix} 1 \\ \\ x_u - c \end{pmatrix} = \mathbf{0}$$

- Robust regression

$$\sum_{u\in\mathcal{P}} \psi(y_u - \alpha - \beta(x_u - c)) \begin{pmatrix} 1 \\ \\ x_u - c \end{pmatrix} = \mathbf{0}$$

  where $\psi(\cdot) = \rho'(\cdot)$

- Name another population attribute which is similar and related to regression.

```
### Correlation
```

- There are numerous ways in which the solution to an equation of *implicitly defined attribute* might be found.
  - Many fall under the category of **root finding** methods.

Newton-Raphson

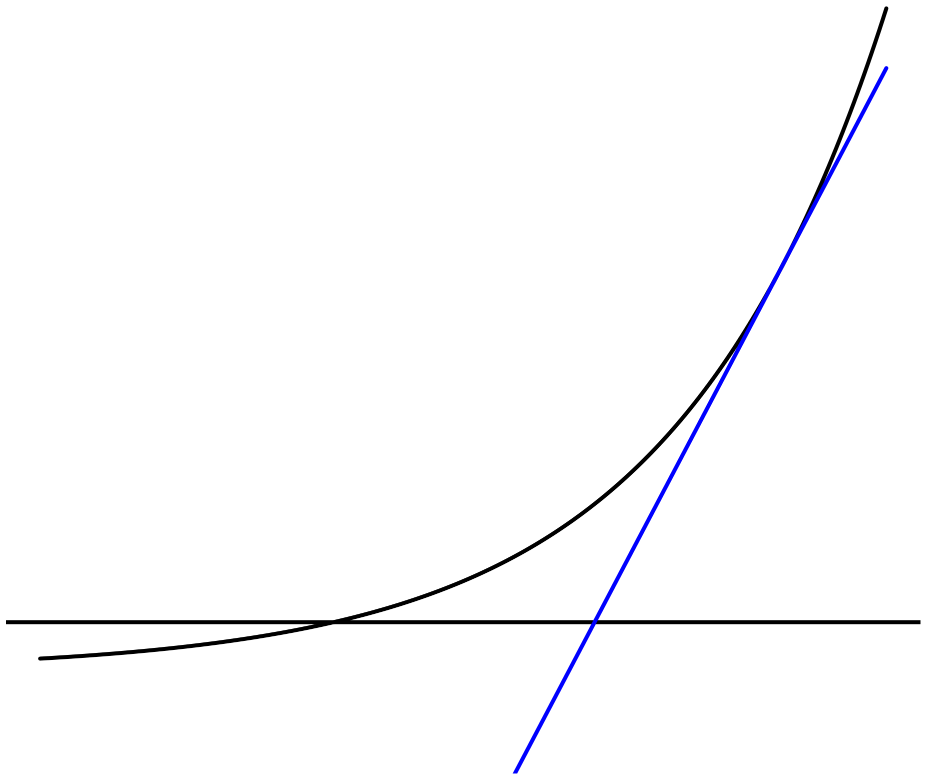
- Given x_0 and if the function $f(x)$ is differentiable
  - we can use a linear function to approximate the function,

$$f(x) \approx f(x_0) + f'(x_0)\,(x - x_0)$$

  - find the root of the linear approximation to get the next iteration, i.e. solve

$$0 = f(x_0) + f'(x_0)\,(x - x_0) \qquad \Rightarrow \qquad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

```r
x.seq = seq(0, 4, length.out = 1000)
plot( x.seq, exp(x.seq) -4 , type='l', ylab='', lwd=2, xaxt='n',
      yaxt='n', xlab='', bty='n', main="",
      mar=0* c(2, 0.1, 0.1, 0.1), xlim=c(0, 4), ylim=c(-10, 51)  )
abline(h=0, lwd=2)

x0 = 3.5
f0 = exp(x0) -4
f1 = exp(x0)
gx = f0 + f1*(x.seq-x0)
lines( x.seq, gx, type='l', lwd=2, col=4)
```



- For our implicitly defined scalar parameter, we want to find the value such that

$$\psi\left(\theta;\mathcal{P}\right)=0$$

  - Given the current parameter guess θ_i a first order approximation is

$$\psi(\theta;\mathcal{P})\approx\psi(\theta_i;\mathcal{P})+\psi'(\theta_i;\mathcal{P})\times(\theta-\theta_i)\,.$$

  - then the update is the root of the linear approximation and is given by

$$\theta_{i+1}=\theta_i-\frac{\psi(\theta_i;\mathcal{P})}{\psi'(\theta_i;\mathcal{P})}$$

- For a vector of parameter defined by a system of equations. i.e. we want the vector which solves

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

- Given a current guess $\boldsymbol{\theta}_i$ we can use a linear function to approximate the function
  - Since, $\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P})$ is a differentiable vector, we let

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

    be the $p \times p$ *matrix* of partial derivatives.
  - Then, a first order approximation can be written as

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

  - then the vector such that the linear approximation is equal to zero which is given by

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

    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:
    a. **Update** the iterate:
         $\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}_i - \left[\boldsymbol{\psi}'(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})\right]^{-1}\ \boldsymbol{\psi}(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})$
    b. **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**.

- For example, here is a simple implementation of Newton's method:

```r
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

The solution to the weighted sum

$$\psi(\theta) = \sum_{u \in \mathcal{P}} w_u (y_u - \theta) = 0$$

can now be found via Newton's method by defining the appropriate `psi(...)` and `psiPrime(...)`. For this function

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

For example, suppose we look at the `facebook` data set and the number of `likes` a posting receives.

```r
facebookfile <- paste(directory, "FacebookMetrics",
                      "facebook.csv", sep=dirsep)
facebook <- read.csv(facebookfile)
### Remove all rows with missing data
fb <- na.omit(facebook)
head(fb[,c(1,2,3,4,6)])
```

```
##   All.interactions share like comment Impressions
## 1              100    17   79       4        5091
## 2              164    29  130       5       19057
## 3               80    14   66       0        4373
## 4             1777   147 1572      58       87991
## 5              393    49  325      19       13594
## 6              186    33  152       1       20849
```

Recall

Variate	Value
`share`	the total (lifetime) number of times the post was shared
`like`	the total (lifetime) number of times the post "liked"
`comment`	the total (lifetime) number of comments attached to the post
`All.interactions`	the sum of `share`, `like`, and `comment`
`Impressions`	the total (lifetime) number of times the post has been displayed, whether the post is clicked or not. The same post may be seen by a Facebook user several times (e.g. via a page update in their News Feed once, whenever a friend shares it, etc.).

It might make sense to 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.

The ψ function for the weighted sum

```r
### 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

```r
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

This is the value theta which should be the same as the weighted average (closed form solution).

```r
y <- fb$like
wt <- 1/fb$Impressions
```

```r
c(theta = result$theta, weightedAve = sum(wt*y) / sum(wt) )
```

```
##       theta weightedAve
##    78.46845    78.46845
```

- **Exercise** Why does this converge in two iterations?
- **Exercise** Try out some other ψ functions for location estimation.
  - e.g. Andrews's sine function, the bisquare weight function, Huber's psi, or Hampel's psi.

General Newton-Raphson R code

```r
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's method 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 back-solving twice).

Example - Rosenbrock function

Suppose we are interested in finding $\boldsymbol{\theta}$ which minimized the Rosenbrock function

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

In optimization, the roots of the gradient are of importance

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

for Newton-Raphson we need the matrix of partial derivatives

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

```
  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)
  }
```

- 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

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

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

- Let's try same example (gradient descent) with different staring value
  - An algorithm's performance depends on the initial guess.

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

```r
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

```r
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

2.3.2.2 Iteratively reweighted least-squares

- For this algorithm we need to
  - rewrite objective function into the form of weighted least squares problem.
  - have the ability to solve the weighted least squares problem.

- The weighted least squares problem is

$$\sum_{u\in\mathcal{P}} w_u \left(y_u - \alpha - \beta\left[x_u - c\right]\right)^2$$

  - When we set c to the weighted average of the x_i's the solution is given by

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

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

- Suppose that the function to be minimized has the form

$$\sum_{u\in\mathcal{P}} \rho(y_u - \alpha - \beta(x_u - c))$$

  - Differentiating this with respect to α and β gives the minimum as a solution to

$$\sum_{u\in\mathcal{P}} \psi(y_u - \alpha - \beta(x_u - c)) \begin{pmatrix} 1 \\ \\ x_u - c \end{pmatrix} = \mathbf{0}$$

    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

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

- **Aside:** For weighted least-squares, $\rho(r) = wr^2$, $\psi(r) = 2wr$ and the equation reduces to

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

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

$$\begin{aligned} \mathbf{0} &= \textstyle\sum_{u\in\mathcal{P}} \psi(r_u)\ \mathbf{z}_u \\ &= \textstyle\sum_{u\in\mathcal{P}} \left(\frac{\psi(r_u)}{r_u}\right)\ r_u\ \mathbf{z}_u \\ &= \textstyle\sum_{u\in\mathcal{P}} w_u r_u\ \mathbf{z}_u \end{aligned}$$

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

- We can put minimizing $\sum_{u\in\mathcal{P}} \rho(y_u - \alpha - \beta(x_u - c))$ into a weighted least squares equation.
  - This suggests another possible algorithm to find the unknown parameters.

- Given some initial α and β.
  - Use this estimate to construct the residuals r_u and

  - Construct weights $w_u = \psi(r_u)/r_u$.
  - Then find estimates of α and β by solving this weighted least squares problem.

- Use these to get new r_u values and new w_u.
  - Solve the weighted least squares problem to get new α and β.
  - Repeat until the estimates of α and β converge.

- This algorithm is called **iteratively reweighted least squares**.

- We can generalize this algorithm by expressing the problem in terms of the vector parameter $\boldsymbol{\theta} = (\alpha,\ \beta)^{\mathsf{T}}$. Then $r_u = y_u - \mathbf{z}_u{}^{\mathsf{T}}\boldsymbol{\theta}$.

The Algorithm

```
1. Initialize ; $i \leftarrow 0$; determine an initial estimate $\widehat{\boldsymbol{\theta}}_0$
2. LOOP:
   a. Construct residuals and weights for all $u \in \mathcal{P}$
          $r_u = y_u - \mathbf{z}_u{}^{\mathsf{T}}\boldsymbol{\theta}_i$   and   $w_u = \frac{\psi(r_u)}{r_u}$
   b. Solve the weighted least squares problem
          $\sum_{u \in \mathcal{P}} w_u (y_u - \mathbf{z}_u{}^{\mathsf{T}}\boldsymbol{\theta})\ \mathbf{z}_u$
   c. Update the parameter:
          $\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}$
   d. Converged?
      if the iterates are not changing, then return
      else $i \leftarrow i+1$ and repeat LOOP.
3. Return: $\widehat{\boldsymbol{\theta}} = \widehat{\boldsymbol{\theta}}_i$
```

- The algorithm above is for any attribute that can be expressed as

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

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

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

R code for Iteratively reweighted least-squares

```
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,
       c0 = sum(wt*x)/sum(wt)
       )
}
```

Example - Least Squares

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

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

```
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-12) {
  ## 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
}
```

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	c0
3.8753064	TRUE	2	6.700776
0.1429041	TRUE	2	6.700776

Compared to `lm` in R

```
lm(waldo$Y ~ I(waldo$X - mean(waldo$X)) )$coef
```

```
##                (Intercept) I(waldo$X - mean(waldo$X))
##                  3.8753064                  0.1429041
```

Example - Robust Regression

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

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

  - and now we have that

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

```r
huber.psi <- function(resid, k = 1.345) {
  val = resid
  subr = abs(resid) > k
  val[ subr ] = k*sign(resid[subr])
  return(val)
}
```

- Using the Animals data as an example

```r
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	c0
3.1282452	TRUE	8	1.438996
0.6885898	TRUE	8	1.438996

Compared to the `rlm` function in R.

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

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

- **Note** these two intercept terms 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

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

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

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

```
## (Intercept)
##    1.452423
```

and refit the model with this offset term.

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

```
##                (Intercept) I(log(Animals2$body) - c0)
##                  3.1427093                  0.6985483
```

- From our previous analysis, the parameter values were 3.128, 0.689.
- Note that centralization based on the mean, i.e. using $x_u - \overline{x}$ in the code above gives similar (but not the same) result:

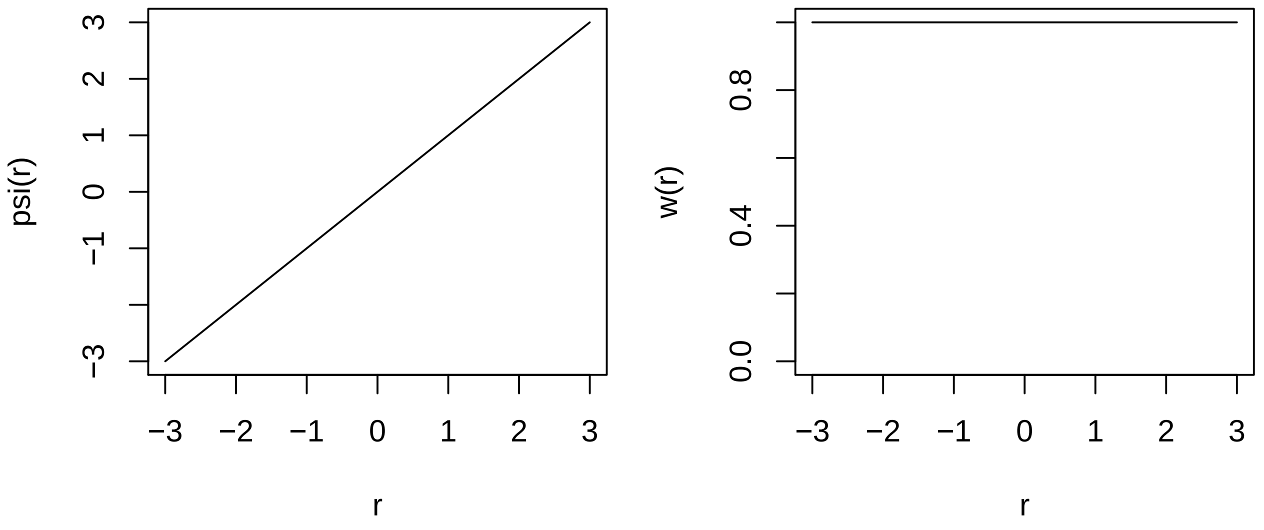
```r
temp3 = rlm(log(Animals2$brain) ~ I(log(Animals2$body) - mean(log(Animals2$body)))  , psi="psi.huber")
temp3$coef
```

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

Robust Regression Review

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

```
rseq = seq(-3,3, 0.01)

par(mfrow=c(1,2), mar=c(5, 5, 0.1, 0.1))
plot(rseq, rseq,type='l', ylab='psi(r)', xlab='r')
plot(rseq, rep(1,length(rseq)),type='l',ylim=c(0,1), ylab='w(r)', xlab='r')
```



Huber Function

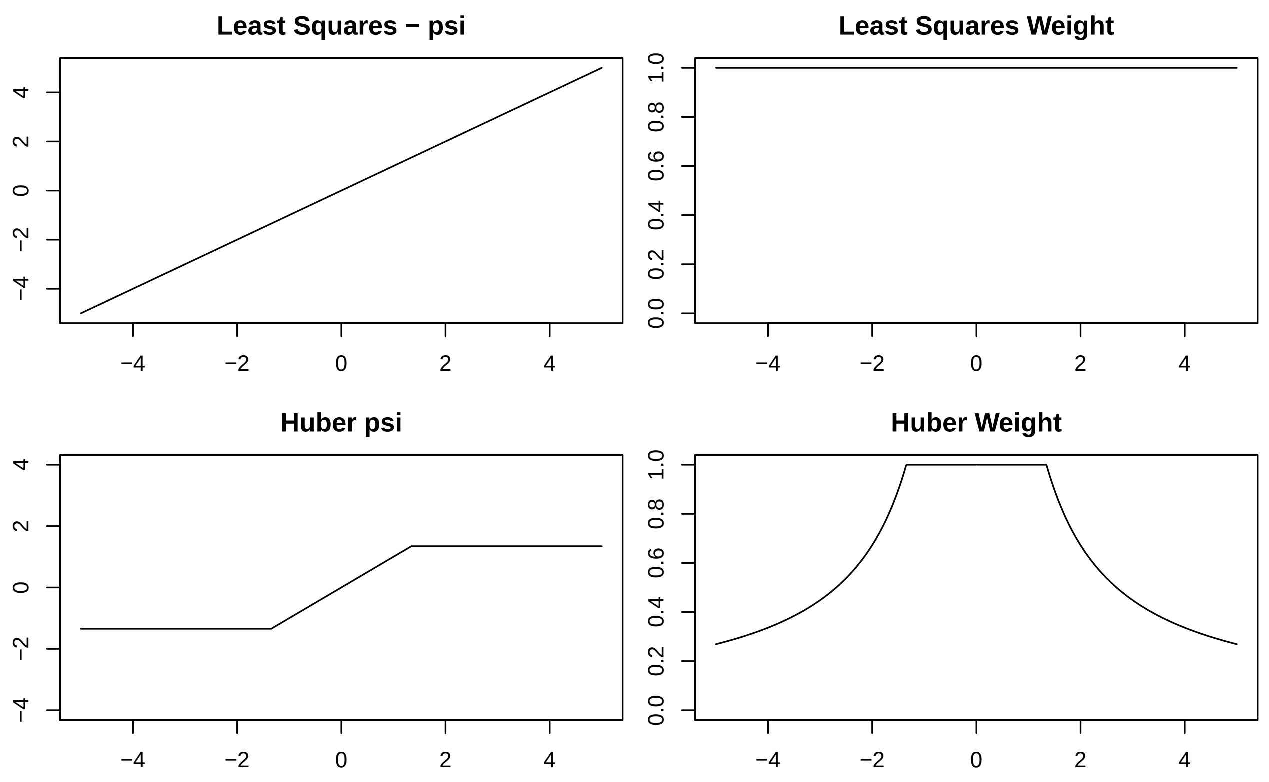
- What does $\psi(\cdot)$ look like for the Huber function?

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

- Compared to least squares we have

```
rseq = seq(-5,5, 0.01)

par(mfrow=c(2,2), mar=2.5*c(1,1,1,0.1) )
```

```r
plot(rseq, rseq,type='l', ylab='psi(r)', xlab='r', main="Least Squares - psi")
plot(rseq, rep(1,length(rseq)),type='l',ylim=c(0,1), ylab='w(r)', xlab='r', main="Least Squares Weight"

plot(rseq, huber.psi(rseq),type='l', ylab='psi(r)', xlab='r', main="Huber psi", ylim=c(-4,4))
plot(rseq, huber.psi(rseq)/rseq,type='l', ylim=c(0,1), ylab='w(r)', xlab='r', main="Huber Weight")
```



- What do we notice?

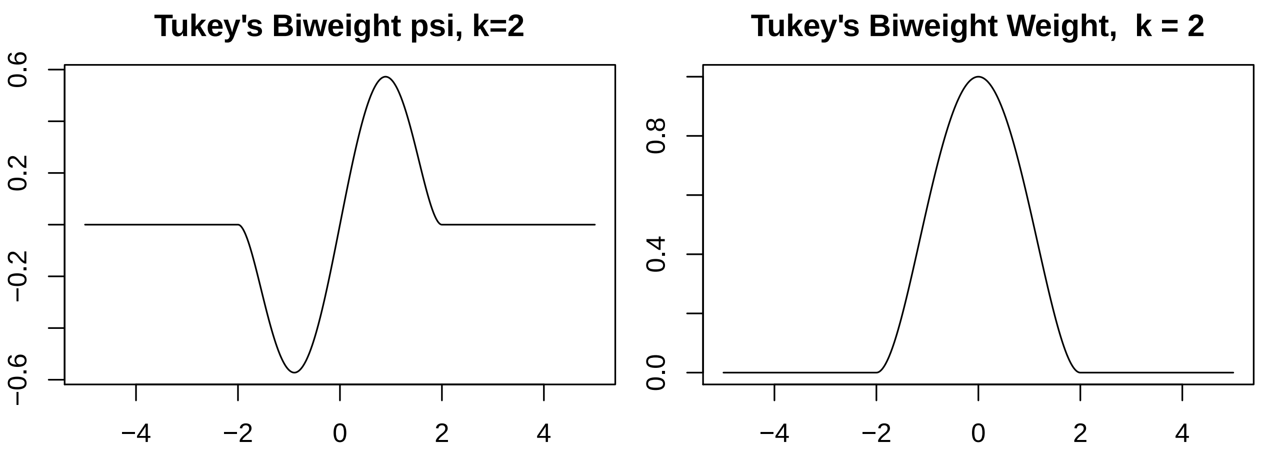
```
## The psi function is bounded
##
## The weight function is 1 and then decreases
```

Tukey's biweight

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

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

```r
r <- seq(-10, 10, length.out=200)

TukeyBiweightpsi <- function (r, k=2) {
  psivals <- rep(0, length(r))
  middlevals  <- {abs(r) <= k}
  psivals[middlevals] <- r[middlevals] * (1 - (r[middlevals]/k)^2)^2
  psivals
}

TukeyBiweight <- function(r, k=2){
  zeros <- r == 0
  wt <- r
  wt[zeros] <- 1
  wt[!zeros] <- TukeyBiweightpsi(r[!zeros],k)/r[!zeros]
  wt
  }

par(mfrow=c(1,2), mar=2.5*c(1,1,1,0.1) )
plot(rseq, TukeyBiweightpsi(rseq),type='l', ylab='psi(r)', xlab='r', main="Tukey's Biweight psi, k=2")
plot(rseq, TukeyBiweight(rseq),type='l',ylim=c(0,1), ylab='w(r)', xlab='r', main="Tukey's Biweight Weig
```



- What is the value k for?

ψ-function

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

- ψ is a piecewise continuous function $\psi : \Re \rightarrow \Re$
- 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.

Scale Parameter

- During the robust regression, we fixed the scale parameter.
    - We can construct an algorithm to estimate the scale as well.

- But which estimate of the scale should we use?

```
MAD <- function(r, k=1.4826) { k*median( abs(r -median(r)) ) }
```

- What is the constant 1.4826 for?

```
## Ensures a consistency when estimating the
## standard deviation of a normal distribution
```

Writing another Huber function to include a measure of scale and change the weights function to calculate the scale.

```
huber.psi2 <- function(resid,  k = 1.345, scale=1) {
  k = k * scale
  val = resid
  subr = abs(resid) > k
  val[ subr ] = k*sign(resid[subr])
  return(val)
}
getWeights <- function(resids, psiFn, delta = 1e-12) {
  smallResids <- abs(resids) <= delta
  resids[smallResids] <- delta * sign(resids[smallResids])

  scale.est = mad( resids )
  psiFn(resids, scale=scale.est)/resids
}
```

After running iteratively reweighted least-squares the result is

```
result2 <- irls(log(Animals2$brain), log(Animals2$body),
                theta = c(1,1),
                psiFn = huber.psi2,
                maxIterations = 100,
                tolerance = 1e-8)
kable(as.data.frame(result2))
```

theta	converged	iteration	c0
3.1086764	TRUE	10	1.404087
0.6981795	TRUE	10	1.404087

Our initial result was

```
kable(as.data.frame(result))
```

theta	converged	iteration	c0
3.1282452	TRUE	8	1.438996
0.6885898	TRUE	8	1.438996

Using the `rlm` command we obtain

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

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

Note that the intercepts has to do with appropriate centralization of the data.

- with the effect of scale and the c value being counted for, our result agrees that of the `rlm` function.

```
cbind( c(result2$theta[1] - result2$theta[2]*result2$c0,  result2$theta[2]), temp$coef)
```

```
##                         [,1]      [,2]
## (Intercept)        2.1283718 2.1281218
## log(Animals2$body) 0.6981795 0.6985483
```