2.3 Implicitly Defined Population Attributes

Contents

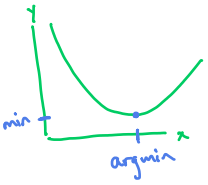
We have defined an attribute $a(\mathcal{P})$ to be a summary of the population $\mathcal{P}$. The numeric and graphical attributes that we have discussed so far are called **explicit** attributes because they are defined *explicitly* by the population variates. In this section we will discuss **implicitly** defined attributes, which also summarize the population $\mathcal{P}$, but which are defined only implicitly by the population variates. In particular, such attributes are solutions to an optimization problem (i.e., the solution to an equation or a set of equations).

"argument that minimizes"

2.3.1 The Minimum of a Function

attribute

In most practical situations we are interested in a (possibly vector-valued) attribute $\boldsymbol{\theta}$ which minimizes some function $\rho(\boldsymbol{\theta}; \mathcal{P})$ of the variates in the population.



- That is, we want the value $\widehat{\boldsymbol{\theta}}$ which satisfies

$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\operatorname{argmin}} \rho(\boldsymbol{\theta}; \mathcal{P})$$

objective function (aka: loss function)

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

"parameter space"

minimum

- Note that maximizing a function is the same as minimizing its negation:

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

and so

$$\underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\operatorname{argmax}} \rho(\boldsymbol{\theta}; \mathcal{P}) = \underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\operatorname{argmin}} -\rho(\boldsymbol{\theta}; \mathcal{P})$$

Therefore, we only need to consider minimization here.

- The most common form for $\rho(\boldsymbol{\theta}, \mathcal{P})$ is a sum of functions $\rho(\boldsymbol{\theta}, u)$ evaluated at each unit $u \in \mathcal{P}$:

$$\rho(\boldsymbol{\theta}, \mathcal{P}) = \sum_{u \in \mathcal{P}} \rho(\boldsymbol{\theta}, u)$$

The following examples include attributes defined implictly in this way. You will be familiar with them already, but you probably have not defined them in this way.

All of these examples seek to determine the value of θ that best represents the population $P = \{y_1, y_2, \ldots, y_N\}$. We want to find $\hat{\theta}$ which is "as close" as possible to the observed variate values. Each of the objective functions below measure distance (i.e., "closeness") differently

Examples (scalar valued attributes)

Some familiar examples for a **scalar valued** attribute $\theta \in \mathbb{R}$ and $u \in \mathcal{P}$ include:

Least-squares: If $\rho(\theta; u) = (y_u - \theta)^2$ then

"squared error loss"

$$\widehat{\theta} = \underset{\theta \in \mathbb{R}}{\operatorname{argmin}} \rho(\theta, \mathcal{P}) \tag{1}$$

$$= \underset{\theta \in \mathbb{R}}{\operatorname{argmin}} \sum_{u \in \mathcal{P}} \rho(\theta, u) \tag{2}$$

$$= \underset{\theta \in \mathbb{R}}{\operatorname{argmin}} \sum_{u \in \mathcal{P}} (y_u - \theta)^2 \tag{3}$$

$$= \bar{y} \tag{4}$$

Try to prove these

← This is a special case of this when $w_u = w \ \forall u$

- **Weighted least-squares**: If $\rho(\theta; u) = w_u(y_u - \theta)^2$ then

$$\widehat{\theta} = \underset{\theta \in \mathbb{R}}{\text{argmin}}\, \rho(\theta, \mathcal{P}) \tag{5}$$
$$= \underset{\theta \in \mathbb{R}}{\text{argmin}} \sum_{u \in \mathcal{P}} \rho(\theta, u) \tag{6}$$
$$= \underset{\theta \in \mathbb{R}}{\text{argmin}} \sum_{u \in \mathcal{P}} w_u(y_u - \theta)^2 \tag{7}$$
$$= \tag{8}$$

- **Least absolute deviations**: If $\rho(\theta; u) = |y_u - \theta|$ then

$$\widehat{\theta} = \underset{\theta \in \mathbb{R}}{\text{argmin}}\, \rho(\theta, \mathcal{P}) \tag{9}$$
$$= \underset{\theta \in \mathbb{R}}{\text{argmin}} \sum_{u \in \mathcal{P}} \rho(\theta, u) \tag{10}$$
$$= \underset{\theta \in \mathbb{R}}{\text{argmin}} \sum_{u \in \mathcal{P}} |y_u - \theta| \tag{11}$$
$$= \tag{12}$$

- **Least generalized-absolute deviations**: If for some $q \in (0, 1)$ we define the vee function

$$\rho_q(\theta; u) = \begin{cases} q(y_u - \theta) & \text{if } y_u \geq \theta \\ (q-1)(y_u - \theta) & \text{if } y_u < \theta \end{cases}$$

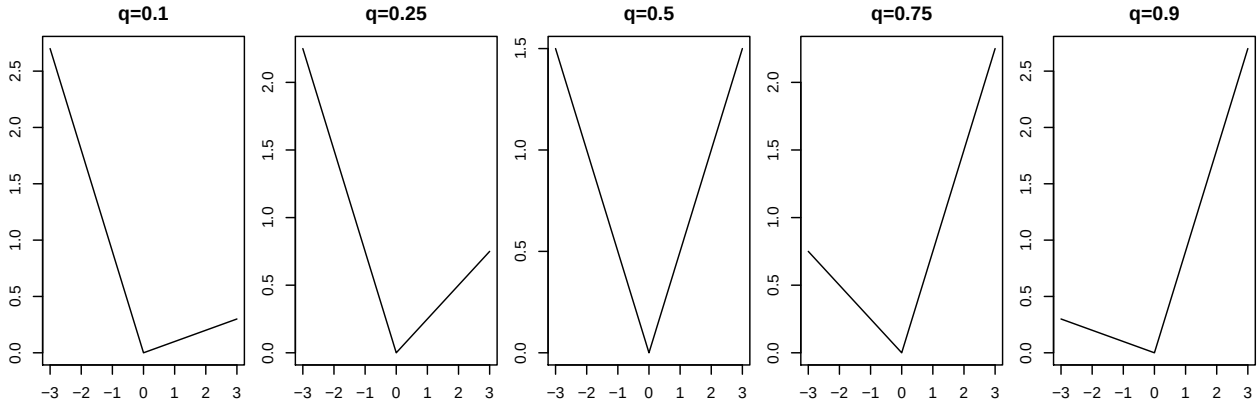
then

$$\widehat{\theta} = \underset{\theta \in \mathbb{R}}{\text{argmin}}\, \rho(\theta, \mathcal{P}) \tag{13}$$
$$= \underset{\theta \in \mathbb{R}}{\text{argmin}} \sum_{u \in \mathcal{P}} \rho(\theta, u) \tag{14}$$
$$= \underset{\theta \in \mathbb{R}}{\text{argmin}} \sum_{u \in \mathcal{P}} \rho_q(\theta; u) \tag{15}$$
$$= \tag{16}$$

Here are several examples of the vee function for different values of q.

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

```
z = seq(-3,3,.01)
par(mfrow=c(1,5), mar=2.5*c(1,1,1,0.1))
plot(z, vee(z, q=1/10), main="q=0.1", type='l')
plot(z, vee(z, q=1/4), main="q=0.25", type='l')
plot(z, vee(z, q=2/4), main="q=0.5", type='l')
plot(z, vee(z, q=3/4), main="q=0.75", type='l')
plot(z, vee(z, q=9/10), main="q=0.9", type='l')
```



Below we illustrate that the 30^{th} quantile, $Q_y(0.3)$, can be determined by minimizing a sum of appropriate `vee` functions. Note that in principle this method can be used to calculate *any* quantile.

1. Set the randomization seed using the commdand: `set.seed(341)`. Doing so ensures that all simulations give the same answer when you rerun them.

2. Simulate a population.

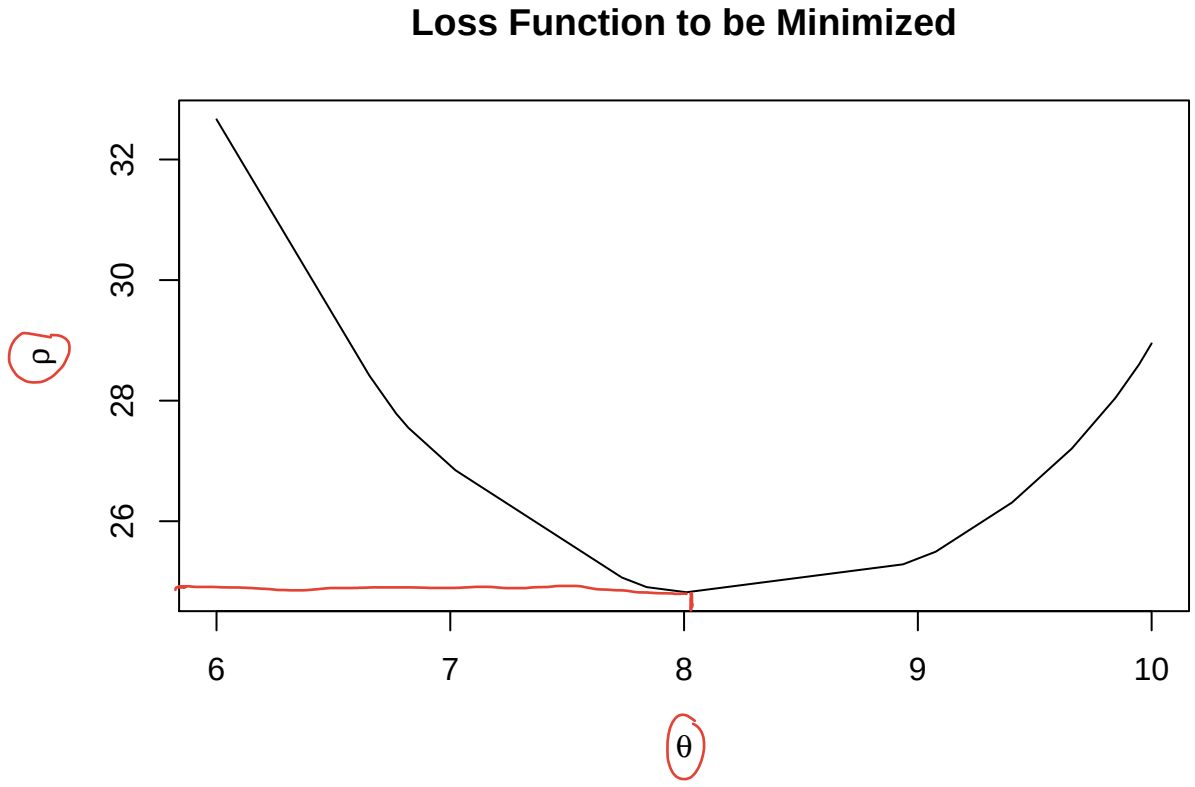
```
set.seed(341)
y.pop <- rnorm(25, 10, 3)
```

3. Define the loss function (sum of `vee` functions) that is to be minimized.

```
rho.fun <- function(theta, q.val){
  return(sum(sapply(X = y.pop-theta, FUN = vee, q = q.val)))
}
```

4. Plot the function.

```
theta = seq(6,10,length.out=1000)
rho.vals = numeric(length(theta))
for(i in 1:length(theta)){
  rho.vals[i] = rho.fun(theta[i], q.val=0.3)
}
plot(theta, rho.vals, type='l',
     xlab = bquote(theta) , ylab = bquote(rho[]),
     main = "Loss Function to be Minimized")
```



5. Use a numerical minimizer in R to find the argument that minimizes this function. We will use all three of `nlminb`, `optimize`, and `optim` for purposes of illustration.

```
nlminb(start = 0.5, objective = rho.fun, 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(f = rho.fun, interval = range(y.pop), q.val = 0.3)
```

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

```
optim(par = 0.5, fn = rho.fun, q.val = 0.3)
```

```
## $par
## [1] 8.010168
##
## $value
## [1] 24.82169
##
## $counts
## function gradient
##       52       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Compare these results to the built-in `quantile` function:

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

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

* this agrees with the minimizers, but it's not super close. However, if we changed the method R uses to calculate a quantile, agreement may have been better.

9 options

Example (vector valued attributes): Simple Linear Regression

A familiar **vector valued** attribute is the vector of coefficients associated with the following simple linear regression:

$$y_u = \alpha + \beta\,(x_u - c) + r_u \qquad \forall\, u \in \mathcal{P} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$$

The attribute of interest is $\boldsymbol{\theta} = (\alpha, \beta)$.

Note that a re-centering of the x_u values in a linear regression is not uncommon. Typically c is chosen to be a meaningful value in the data set such as the average x_u value (i.e., $c = \overline{x}$), for example. Different choices of c give rise to different interpreations for α. Not all such interpretations have practical relevance.

$c = 0$: α is the expected response when $x_u = 0$
$c = \overline{x}$: α is the expected response when $x_u = \overline{x}$

- These coefficients are determined implicitly by

$$\widehat{\boldsymbol{\theta}} = \left(\widehat{\alpha}, \widehat{\beta}\right) = \underset{(\alpha,\beta)\,\in\,\mathbb{R}^2}{\operatorname{argmin}} \sum_{u\,\in\,\mathcal{P}} \left(y_u - \alpha - \beta(x_u - c)\right)^2$$

objective (loss) function

r_u

* The question being answered by this attribute: What line best describes the relationship between x and y?

- It can be shown (show this) that

Least Squares Estimates →

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

- The resulting estimates determine the **least-squares** fitted line:

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

- The equation of the fitted values, defined for all $u \in \mathcal{P}$, is:

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

- The residuals are the difference between y_u and $\hat{y}_u$

$$\widehat{r}_u = y_u - \widehat{\alpha} - \widehat{\beta}\,(x_u - c)$$

  Each residual is the signed vertical distance between the point (x_u, y_u) and the point $(x_u, \widehat{y}_u) = \left(x_u, \widehat{\alpha} + \widehat{\beta}\,(x_u - c)\right)$. The latter point is the value of the fitted line, defined by $\widehat{\boldsymbol{\theta}} = \left(\widehat{\alpha}, \widehat{\beta}\right)$, calculated at $x = x_u$.

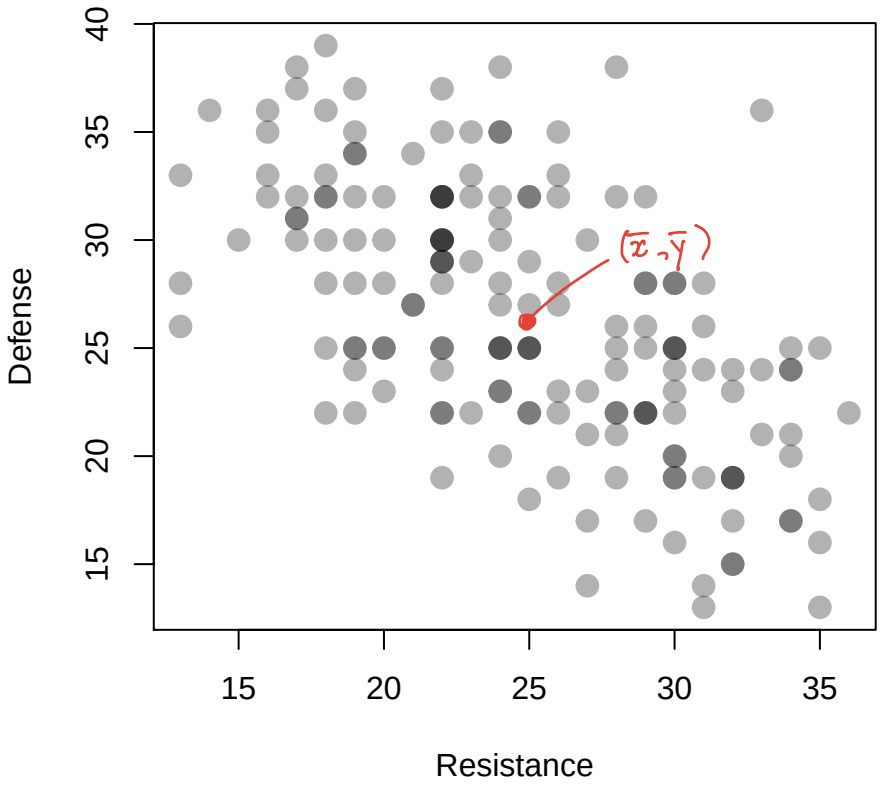
Fire Emblem Heroes Example (Calculating Influence)

- `RES`: resistance, ability to absorb magical attack (use as x)
- `DEF`: defense, ability to absorb physical attacks (use as y)

```
feh = read.csv("../Data/feh.csv", header=TRUE)
head(feh)
```

```
##      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
## 6  Arthur  Green Axe Infantry 43  32  29  30  24   158
```

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



- The averages for defense and resistance, respectively, are $\bar{y} = 26.4$ and $\bar{x} = 24.9$
- This relationship can be summarized as

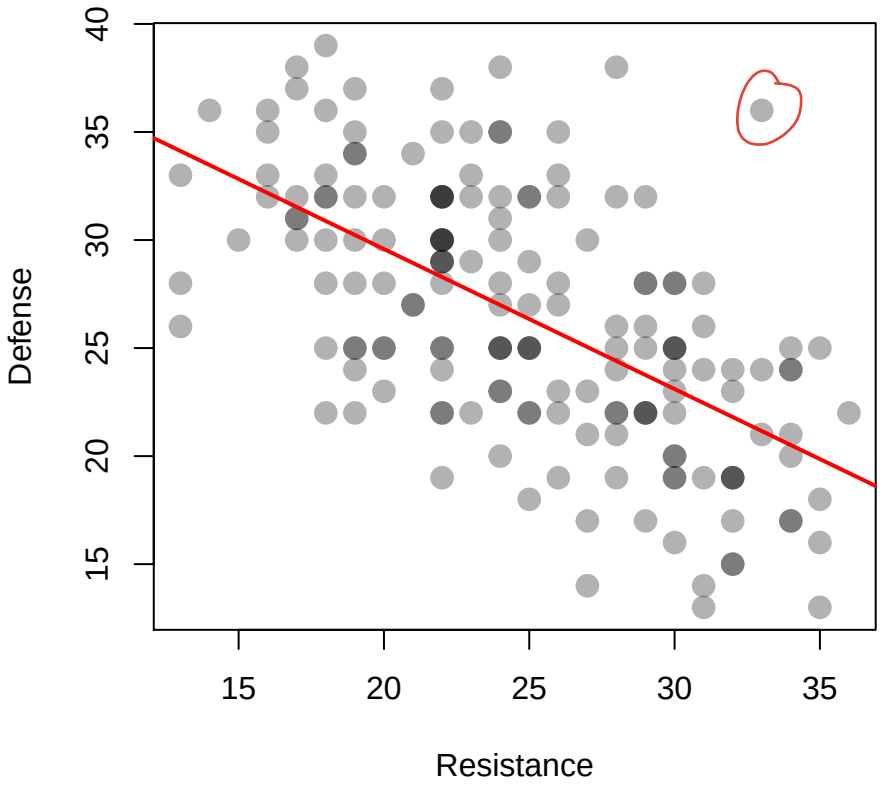
$$\text{Defense} = 42.5 - 0.65 \times \text{Resistance} + \text{error}$$

  - or equivalently as

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

  - both equations (non-centred and centered, respectively) yield the same line and fitted values.

```
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" )

fit = lm(DEF ~ RES, data=feh)
abline(fit, col="red", lwd=2)
```



- What have we done here?

  We've summarized the population with a straight line.

- Does it make sense to use a regression line here?

  Yes, the relationship between DEF and RES appears linear.

- In terms of the game is the regression line important?

  Yes, this illustrates that as resistance increases defense decreases (and vice versa) and that generally characters don't have both high defense and high resistence.

- Is the notion of **influence** still relevant with implicitly defined attributes? YES!

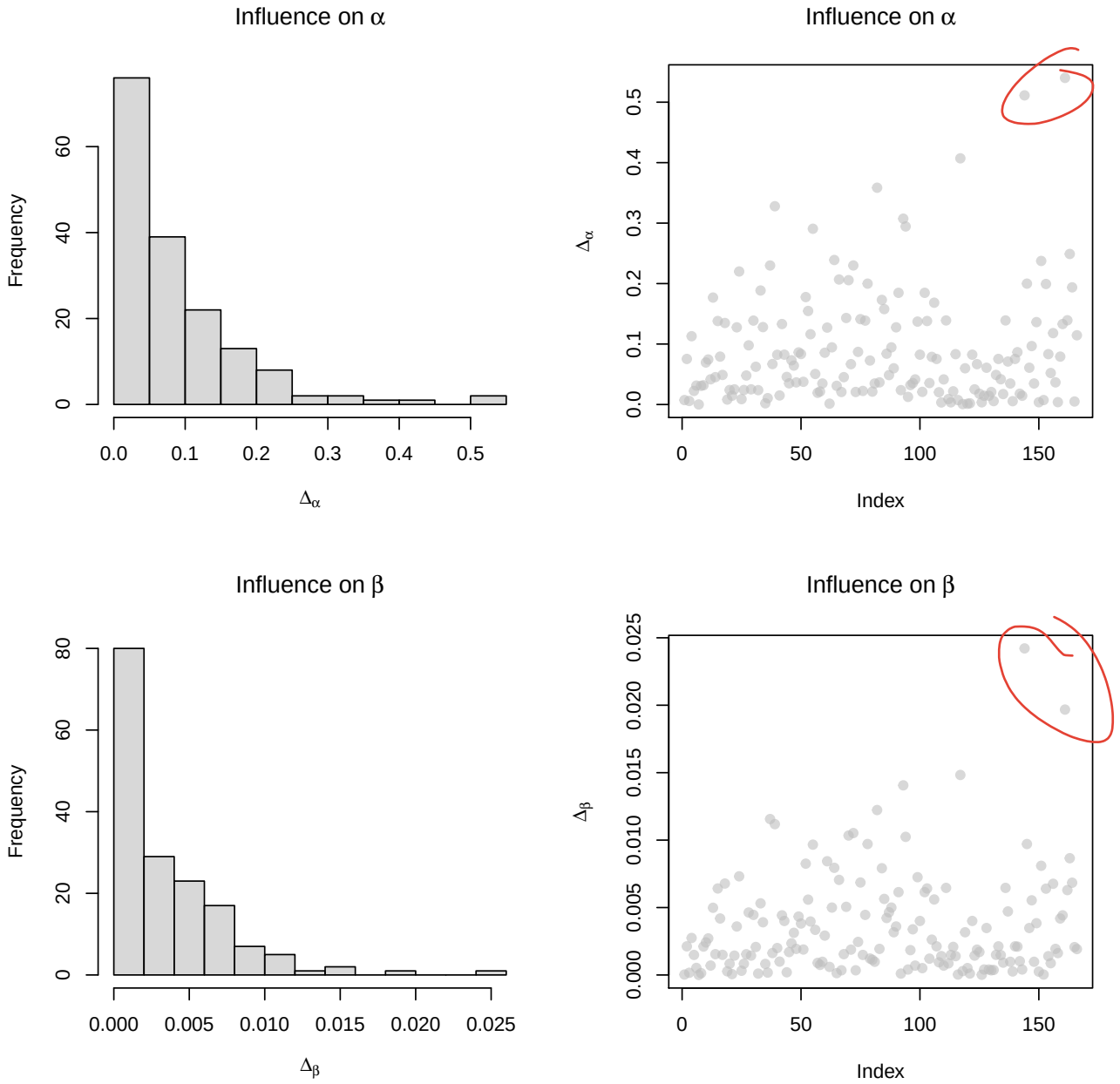
```
N = nrow(feh)
delta = matrix(0, nrow=N, ncol=2)
for (i in 1:N) {
  ## feh[-i] removes the ith row from a vector
  fit.no.i = lm(DEF ~ RES, data=feh[-i,])
  delta[i,] = abs(fit$coef - fit.no.i$coef)
}
```

Scalar-valued Influence: $\Delta(\theta, u) = |\hat{\theta} - \hat{\theta}_{[-u]}|$ where $\hat{\theta}_{[-u]}$ is calculated without unit u

Vector-valued Influence: $\Delta(\hat{\boldsymbol{\theta}}, u) = \|\hat{\boldsymbol{\theta}} - \hat{\boldsymbol{\theta}}_{[-u]}\|_2$ ← Euclidean (L_2) norm

Then plot the Δ's associated with each of the parameters (i.e., each component of the attribute $\boldsymbol{\theta}$)

```
par(mfrow=c(2,2))

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

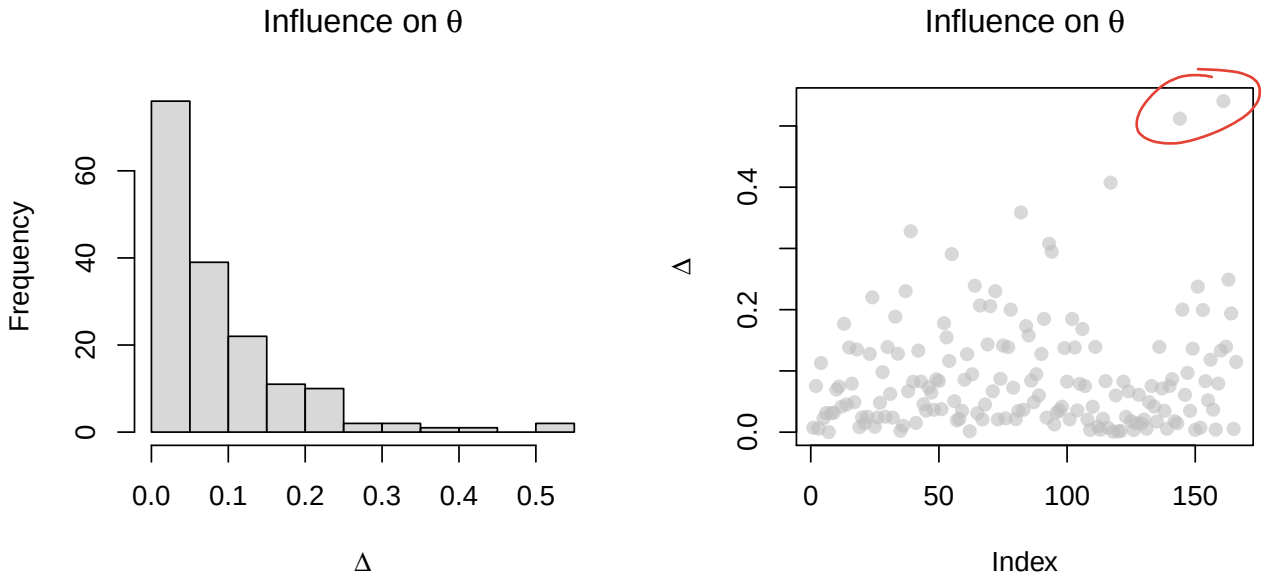


These plots illustrate the inluence each unit has on the indvidual coefficient estimates, but they don't illustrate

the influence each unit has on the regression line as a whole. To do this we define influence for a vector valued attribute using the vector's Euclidean norm.

$$\|z\|_2 = \sqrt{z_1^2 + z_2^2 + \cdots + z_k^2}$$

```
par(mfrow=c(1,2))

delta2 = apply(X = delta, MARGIN = 1, FUN = function(z) { sqrt(sum(z^2)) })
hist(delta2, breaks = "FD", main=bquote("Influence on" ~ theta),
     xlab = bquote(Delta), col=adjustcolor("grey", 0.6))
plot(delta2, main=bquote("Influence on" ~ theta),
     ylab = bquote(Delta), pch=19, col=adjustcolor("grey", 0.6) )
```



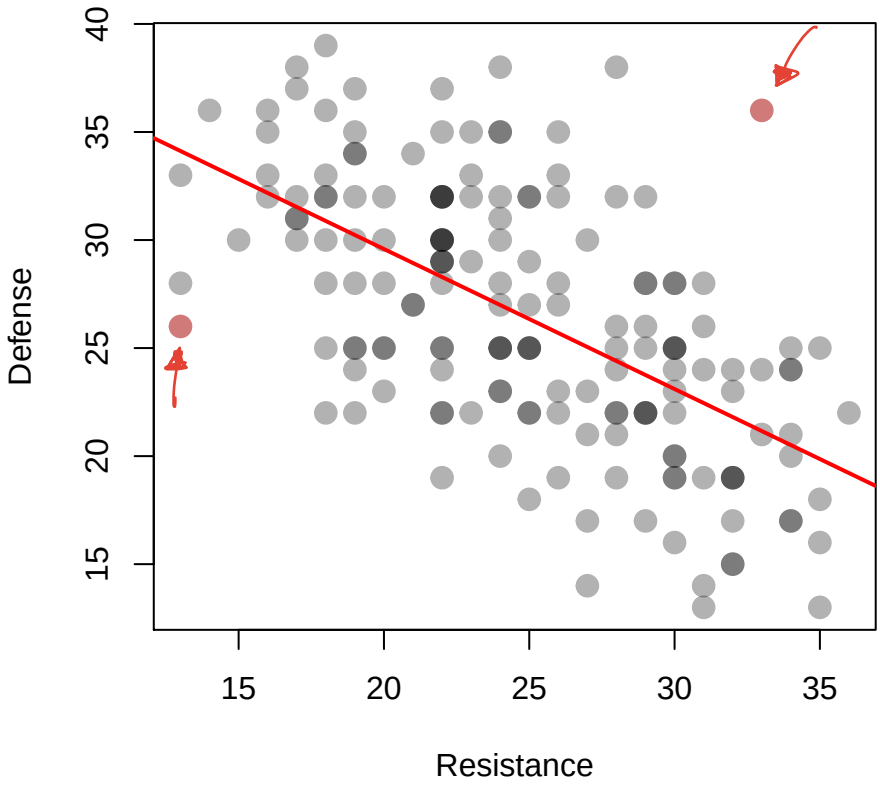
The two units with the largest influence are:

```
feh[delta2 > 0.5,]
```

```
##       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. Let's highlight them on the scatterplot.

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

plot(feh$RES, feh$DEF, main= "",
     pch = 19,  cex=1.5,
     col= col.nam[(delta2 > 0.5) + 1],
     xlab="Resistance", ylab="Defense" )

# fit = lm(DEF ~ RES, data=feh)
abline(fit, col=2, lwd=2)
```



- How else could we define influence for vector valued attributes?

  - Could change measure of distance (i.e., a different norm)

  - Account for uncertainty in our line of best fit calculation (i.e., Cook's-D statistic)

Animals Example (Removing Influential Units)

In this section we discuss the idenfication of units that have a large influence on regression coeffcients and illustrate how to design attributes (i.e., alternative methods of estimating regression coefficients) that are resistant to the effect of influential observations.

The data we consider here is the `Animals2` dataset from the `robustbase` package in `R`. The data frame contains average brain and body weights for 62 species of land animals:

- `body` represents body weight in `kg`
- `brain` represents brain weight in `grams`

An excerpt of the data as well as numerical and graphical summaries are shown below:

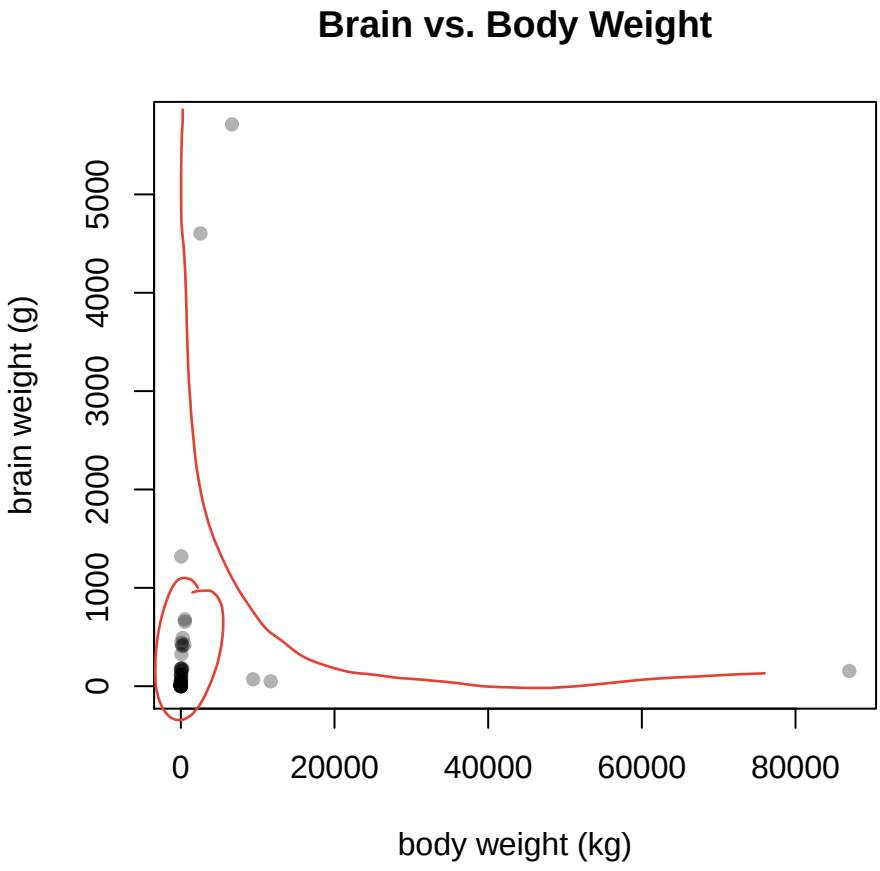
```
library(robustbase)
head(Animals2, n = 5)
```

```
##                   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
```

```
summary(Animals2)
```

```
##       body              brain
##  Min.   :    0.00   Min.   :   0.14
##  1st Qu.:    0.75   1st Qu.:   5.00
##  Median :    3.50   Median :  21.00
##  Mean   : 1852.69   Mean   : 274.29
##  3rd Qu.:   60.00   3rd Qu.: 157.00
##  Max.   :87000.00   Max.   :5712.00
```

```
plot(Animals2, main = "Brain vs. Body Weight",
     xlab = "body weight (kg)", ylab = "brain weight (g)",
     pch = 16, col = adjustcolor("black", alpha.f = 0.3))
```



- Given the scatterplot above, can you see any relationship between body weight and brain weight?

Not really, the data is too skewed

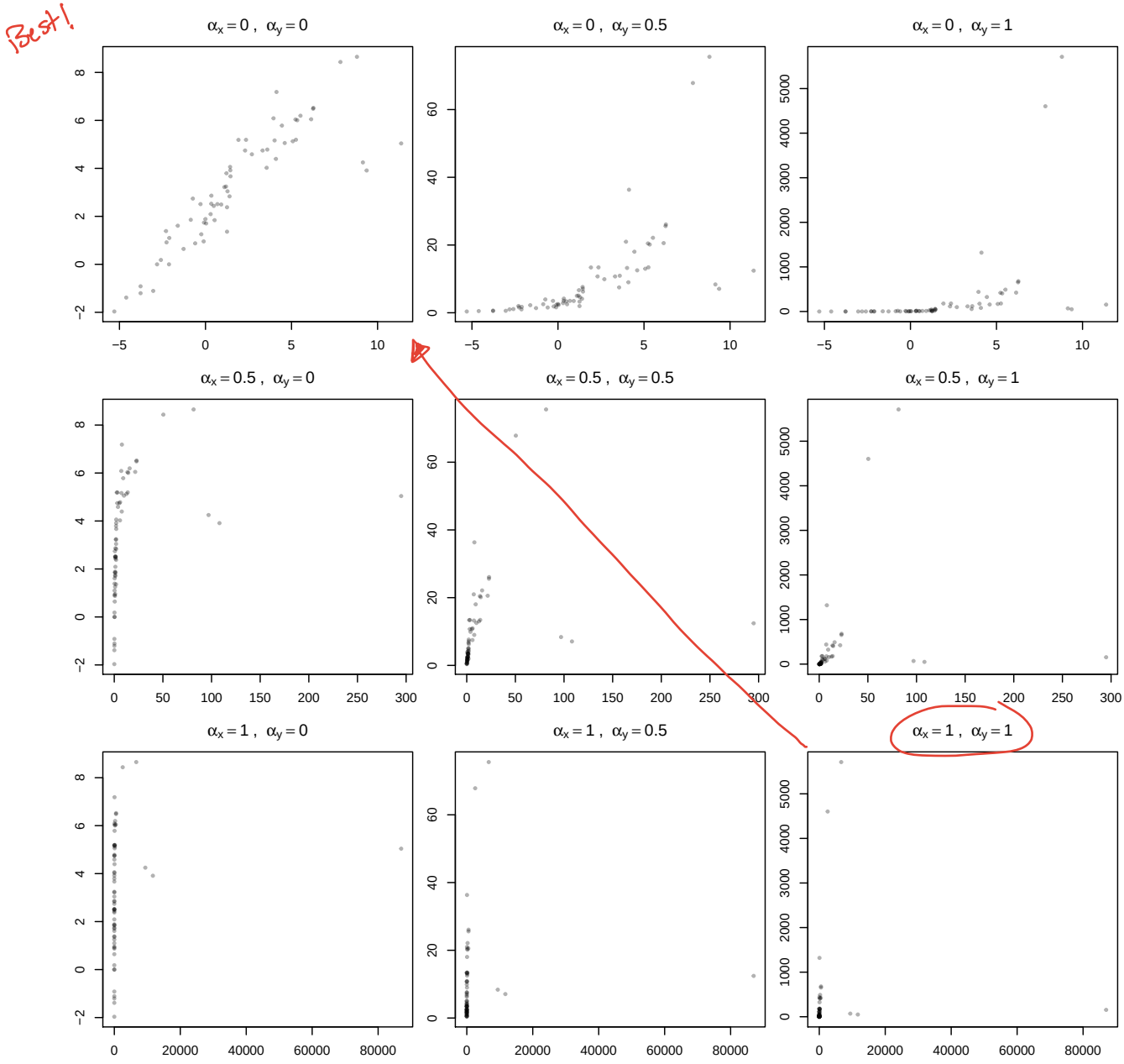
- Both variates are very skewed, so a power transformation might help – which direction should we go with the powers?

Both α's should go down

- Let's look at the power-transformed data under several different transformations (choices of the α's):

```
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 = bquote(alpha[x] == .(a[i]) ~ ", " ~ alpha[y] == .(b[i]) ) )
     # main = paste('alpha_x = ', a[i] ,' & alpha_y=', b[i] ) )
}
```

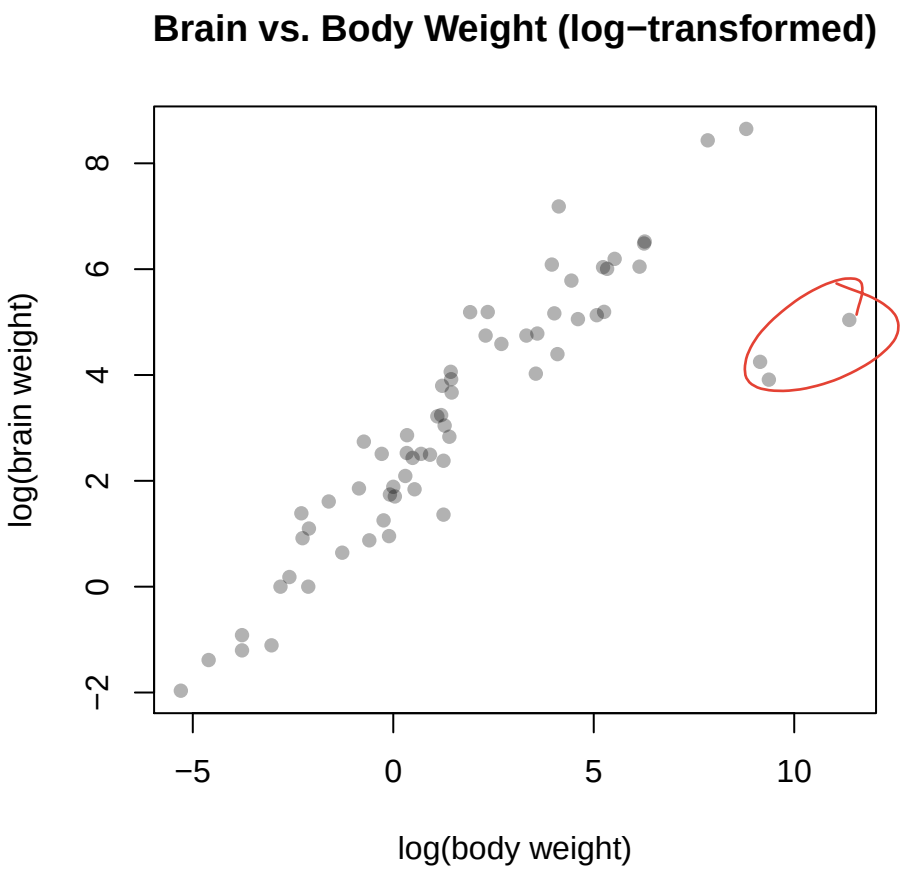


- Do our "ladder rules" for power transformations seem applicable here?

- Which transformation do you prefer?

```
plot(x = log(Animals2$body), y = log(Animals2$brain),
     main = "Brain vs. Body Weight (log-transformed)",
     xlab = "log(body weight)", ylab = "log(brain weight)",
     pch = 16, col = adjustcolor("black", alpha.f = 0.3))
```

Question: What line best summarizes this relationship?



There appear to be 3 units that do not behave as the others do. It is likely that these will have a large **influence** on the fitted regression line. Let's fit the model and study the effect of these extreme units.

- In what follows it will be convenient to define the objective function very generally:

$$(\widehat{\alpha}, \widehat{\beta}) = \underset{(\alpha,\beta) \in \mathbb{R}^2}{\operatorname{argmin}} \sum_{u \in \mathcal{P}} \rho\left(y_u - \alpha - \beta(x_u - c)\right)$$

  where different forms of the function $\rho(\cdot)$ give rise to different fitted lines:

  - Least Squares Regression

$$\rho\left(y_u - \alpha - \beta(x_u - c)\right) = \left[y_u - \alpha - \beta(x_u - c)\right]^2$$

  - Weighted Least Squares Line:

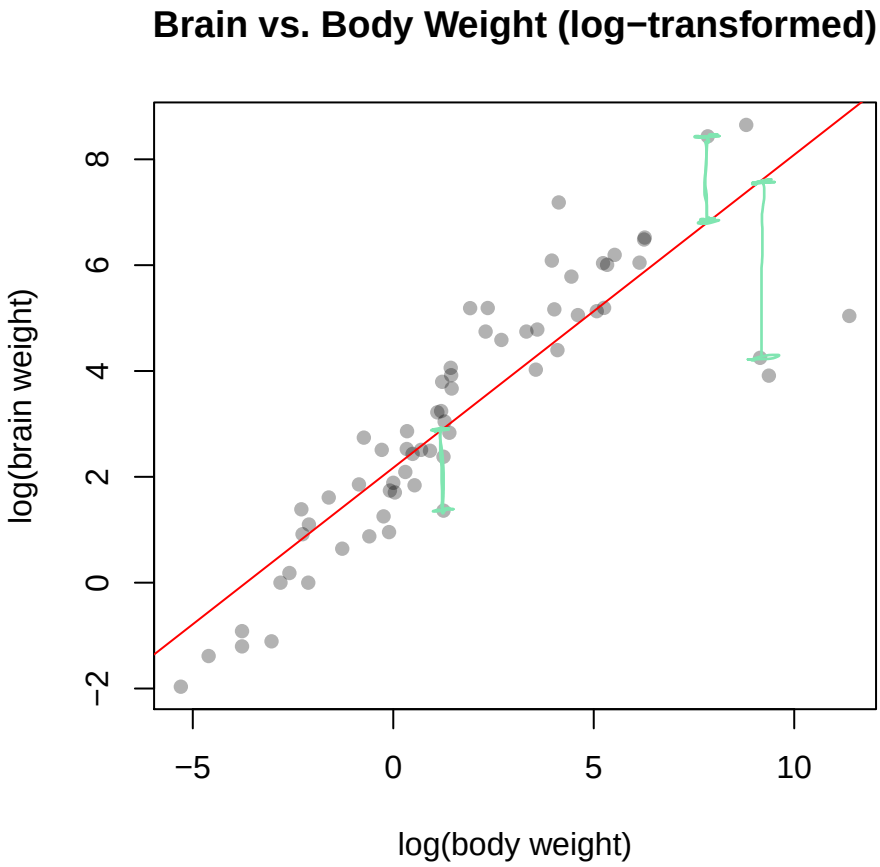
$$\rho\left(y_u - \alpha - \beta(x_u - c)\right) = w_u \left[y_u - \alpha - \beta(x_u - c)\right]^2$$

  - Least Absolute Deviations Line:

$$\rho\left(y_u - \alpha - \beta(x_u - c)\right) = \left|y_u - \alpha - \beta(x_u - c)\right|$$

- Calculate and plot the least squares (LS) line of best fit for the transformed variates:

```
plot(x = log(Animals2$body), y = log(Animals2$brain),
     main = "Brain vs. Body Weight (log-transformed)",
     xlab = "log(body weight)", ylab = "log(brain weight)",
     pch = 16, col = adjustcolor("black", alpha.f = 0.3))
abline(lm(log(Animals2$brain) ~ log(Animals2$body)), col = "red")
```

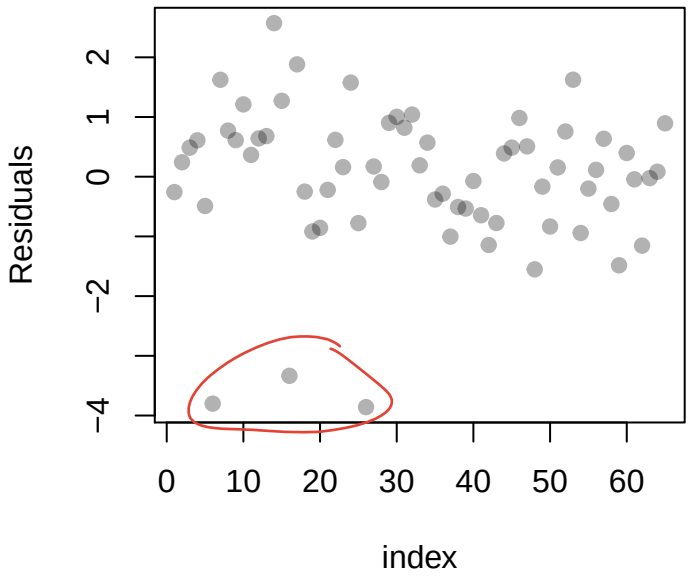


- Does the LS regression line do a good job at representing this relationship?

Kind of. It captures the general relationship, but doesn't do as good a job of summarizing the actual relationship that exists between the majority of the points.

- Let's analyze the residuals

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

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



- These confirm that there are three units that are somehow different from the others.
- 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.

- Let's begin by removing them:
  - `Red line`: LS regression using all the data
  - `Blue line`: LS regression while removing those three `outlier` points

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

plot(x = log(Animals2$body), y = log(Animals2$brain),
     main = "Brain vs. Body Weight (log-transformed)",
     xlab = "log(body weight)", ylab = "log(brain weight)",
     pch = 16, col = adjustcolor("black", alpha.f = 0.3))

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 )
```

- Although we have not formally calculated an influence statistic, the difference between the red and blue lines is a visual representation of the influence those three units have on the LS line.
- Which line seems like a better representation of the population?

The blue line does a better job at summarizing the majority of the population

So what are these units? Why are they so different from the others?

```
indx <- which(log(Animals2$body)>9)
indx
```

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

```
Animals2[indx,]
```

```
##                 body brain
## Dipliodocus    11700  50.0
## Triceratops     9400  70.0
## Brachiosaurus 87000 154.5
```

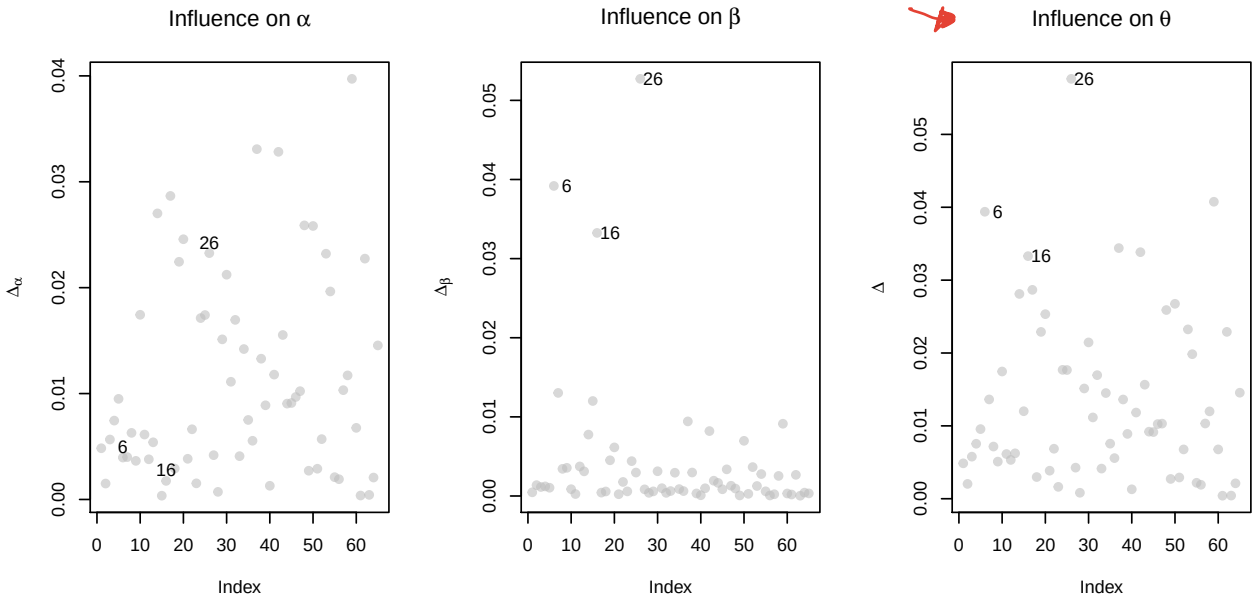
- These three data points are dinosaurs!
  - The rest of the data-set are land mammals.
  - Relative to other land mammals, dinosaurs have significantly larger bodies relative to their brain weights!

- 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, ] = abs(theta.hat-temp.model$coef)
}

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

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

plot(delta[,2], ylab=bquote(Delta[beta]), main=bquote("Influence on" ~ beta),
     pch=19, col=adjustcolor("grey", 0.6) )

obs = c(6,16,26)
text(obs+3, delta[obs,2] , obs)
```

```
delta2 = apply(X = delta, MARGIN = 1, FUN = function(z) { sqrt(sum(z^2)) } )
plot(delta2, ylab = bquote(Delta), main=bquote("Influence on" ~ theta),
      pch=19, col=adjustcolor("grey", 0.6) )

text(obs+3, delta2[obs], obs)
```

this one is most appropriate



- Note these units (dinosaurs) have a large influential on the slope but not on the intercept.

Rather than deleting the problematic units, let's next consider re-weighting all of the units.

Example (vector valued attributes): Weighted Linear Regression

In Weighted Least Squares (WLS), the fitted line minimizes the following objective function

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

- It is assumed the weights w_u are known but, as we will see, the residuals from an ordinary LS regression model can help us determine sensible values.
- As in ordinary LS regression a choice for c needs to be made. In this setting it is common to either set $c = 0$ or define c to be the weighted average of the x_u values:

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

- Given the values of the w_u's and c, we determine $\left(\widehat{\alpha}, \widehat{\beta}\right)$ by taking derivatives of $\rho(\boldsymbol{\theta}; \mathcal{P})$ with respect to each parameter and then setting the resulting gradient equal to zero and solving the system of

equations.

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

- Doing so yields the following parameter values (show this):

WLS Estimates → $\widehat{\alpha} = \overline{y}_w - \beta\,(\overline{x}_w - c)$ and $\widehat{\beta} = \dfrac{\sum_{u\in\mathcal{P}} w_u(x_u - \overline{x}_w)(y_u - \overline{y}_w)}{\sum_{u\in\mathcal{P}} w_u(x_u - \overline{x}_w)^2}$

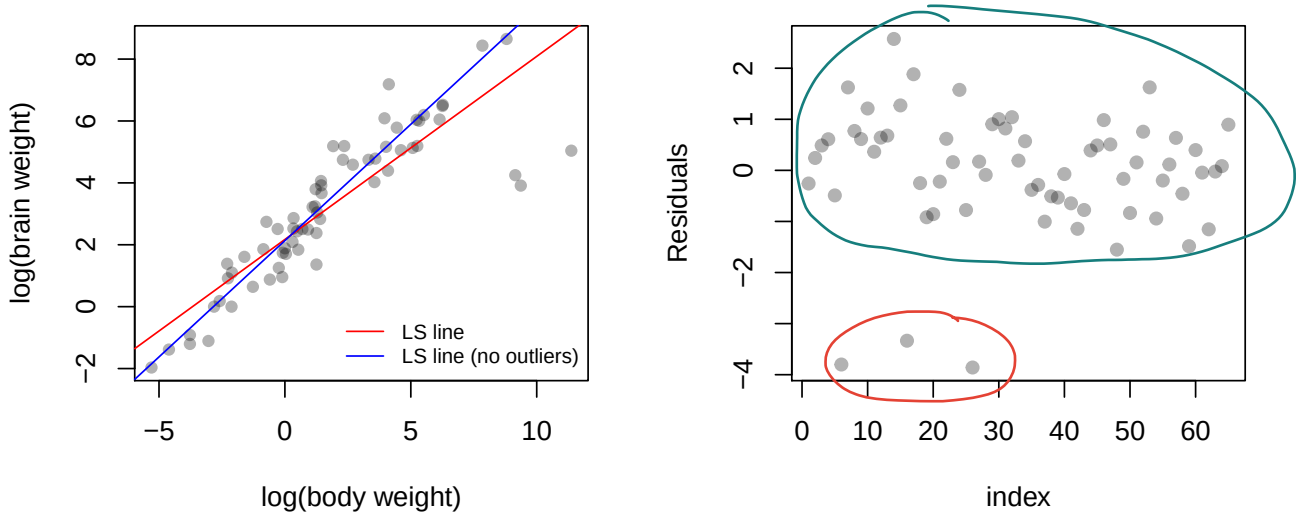
where $\overline{y}_w$ and $\overline{x}_w$ are respectively the weighted averages of the y and x values.

Animals Example (Weighted Least Squares)

We've seen previously that working with the log-transformed variates `log(brain weight)` and `log(body weight)` improved our ability to model the relationship between brain and body weight. We also saw that there were 3 unusual units (they were dinosaurs) that behaved differently than the others.

Recall:

```
par(mfrow = c(1,2))
plot(x = log(Animals2$body), y = log(Animals2$brain),
     main = "",
     xlab = "log(body weight)", ylab = "log(brain weight)",
     pch = 16, col = adjustcolor("black", alpha.f = 0.3))

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

legend("bottomright", legend=c("LS line", "LS line (no outliers)"), col = c("red", "blue"),
       cex = 0.75, bty = "n", lty = 1)

plot( residuals(lm(log(Animals2$brain) ~ log(Animals2$body))),
      col=adjustcolor("black", alpha = 0.3),
      ylab='Residuals', xlab='index', pch=19 )
```



Rather than removing these units from the dataset and fitting an ordinary LS regression model, here we will fit a WLS regression model which includes all of the data, but that differentially weights the data points.

- Notice that some units have different residual variation than others.
- The residual standard deviations for both the outliers and the remaining points are:

```
res =  residuals(lm(log(Animals2$brain) ~ log(Animals2$body)))

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

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

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

The relative variation (the ratio between these) is

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

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

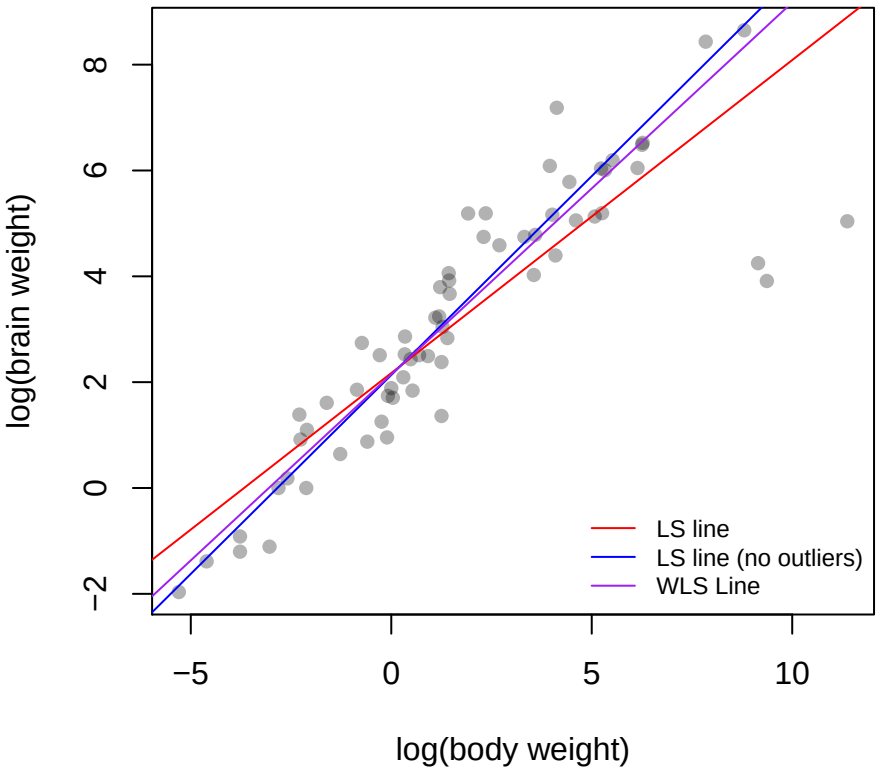
- Since the dinosaurs have larger residual variation we should give them less weight.
- Since the relative variation is roughly 0.2349 we could consider letting

$$w_u = \begin{cases} 1 & \text{for mammals} \\ 0.2349 & \text{for dinosaurs} \end{cases}$$

and then solve

$$(\widehat{\alpha}, \widehat{\beta}) = \underset{(\alpha,\beta) \in \mathbb{R}^2}{\operatorname{argmin}} \sum_{u \in \mathcal{P}} w_u \big[y_u - \alpha - \beta(x_u - c)\big]^2$$

```
plot(x = log(Animals2$body), y = log(Animals2$brain),
     main = "",
     xlab = "log(body weight)", ylab = "log(brain weight)",
     pch = 16, col = adjustcolor("black", alpha.f = 0.3))

abline( lm(log(Animals2$brain) ~ log(Animals2$body)), col = "red" )
W = rep(1,65)
W[which(log(Animals2$body)>9)]=0
abline( lm(log(Animals2$brain) ~ log(Animals2$body),weights=W), col = "blue" )
wt = rep(1,65)
wt[obs] = remaining.sd/outlier.sd
abline( lm(log(Animals2$brain) ~ log(Animals2$body), weights=wt), col = "purple")

legend("bottomright",
       legend=c("LS line", "LS line (no outliers)", "WLS Line"),
       col = c("red", "blue", "purple"),
       cex = 0.75, bty = "n", lty = 1)
```



- Which line best represents the relationship in the population?

Definitely blue or purple. Your choice

Both of the procedures for dealing with outliers discussed so far (deletion and re-weighting) have been very manual. It would be nice to have a more automatic procedure to do this.

Example (vector valued attributes): Robust Regression

In the presence of influential units we would like to define an attribute that is **robust** (resistent) to their inlfuence. We have seen that different definitions of the loss function $\rho(y_u - \alpha - \beta\ (x_u - c))$ impact the nature of this influence. We've seen that

- LS places equal weight on every single unit, whereas
- WLS provides a method for giving less weight to units with LS residuals that are large (in magnitude)

Another alternative regression technique that is robust to outliers involves changing the loss function $\rho(y_u - \alpha - \beta\ (x_u - c))$ function so that

- it gives lower weight than LS to units with large residuals (i.e., u such that $|r_u| >> 0$), and that
- it is quadratic near 0 and hence behaves similarly to LS for units with small residuals (i.e., u such that $|r_u| \approx 0$)

We achieve these goals with the **Huber Loss Function** which is a mixture of the quadratic and absolute value functions:

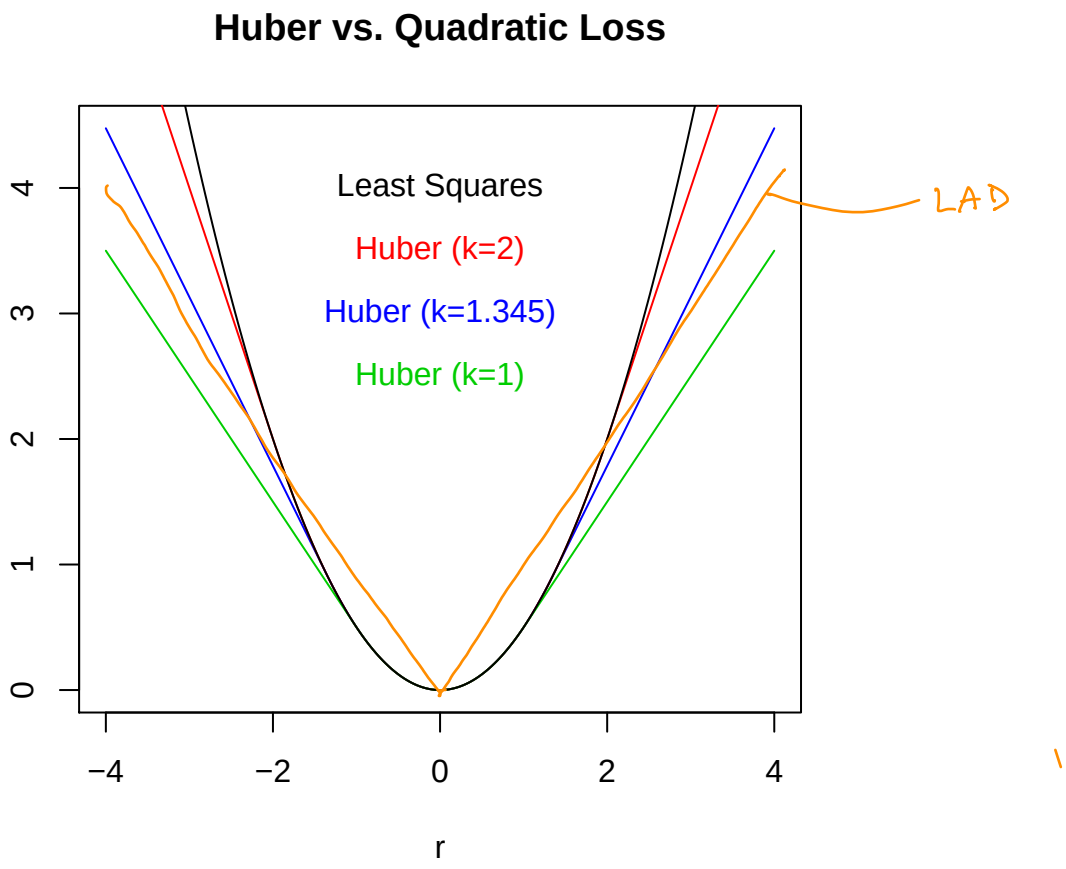
$$\rho_k(r) = \begin{cases} \frac{1}{2}r^2 & \text{for } |r| \leq k \\ k\,|r| - \frac{1}{2}k^2 & \text{for } |r| > k \end{cases}$$

Here is a general-purpose R implementation of the Huber Function:

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

And here is a plot of the Huber Function for different choices of the threshold k:

```
r = seq(-4,4,.01)
plot( r, huber.fn(r,k=1.345), type='l', col=4, main="Huber vs. Quadratic Loss" , ylab = "" )
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)
```



- An attribute (e.g., regression coefficients) based on this function will be affected by the scale of r, and

so...

  - we might let $k = cS$ where S is a (possibly robust) measure of scale.

- In practice it is common to set $k \approx 1.345S$ to satisfy a theoretical balance between efficiency and resistance to outliers.
  - Other common choices include $k = 1.5$ or 2.
  - Note that as k increases, the robust regression with Huber function imposes a larger penalty on larger residuals, hence approaches the LS fit.

In fact, $\lim_{k \to \infty} \rho_k(r) = \frac{1}{2} r^2$

Another form of **robust regression** involves defining the loss function in terms of **least absolute deviations (LAD)**:

$$(\widehat{\alpha}, \widehat{\beta}) = \underset{(\alpha,\beta) \in \mathbb{R}^2}{\operatorname{argmin}} \sum_{u \in \mathcal{P}} |y_u - \alpha - \beta(x_u - c)|$$

- However, in both Huber and LAD-based regression the attribute $(\widehat{\alpha}, \widehat{\beta})$ cannot be solved for in closed form.
- When the attribute of interest is

$$(\widehat{\alpha}, \widehat{\beta}) = \underset{(\alpha,\beta) \in \mathbb{R}^2}{\operatorname{argmin}} \sum_{u \in \mathcal{P}} \rho\left(y_u - \alpha - \beta(x_u - c)\right)$$

  but the definition of $\rho(\cdot)$ precludes straightforward calculation, we consider the following optimization methods:
  - Gradient descent
  - Newton-Raphson
  - Iteratively reweighted least-squares

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

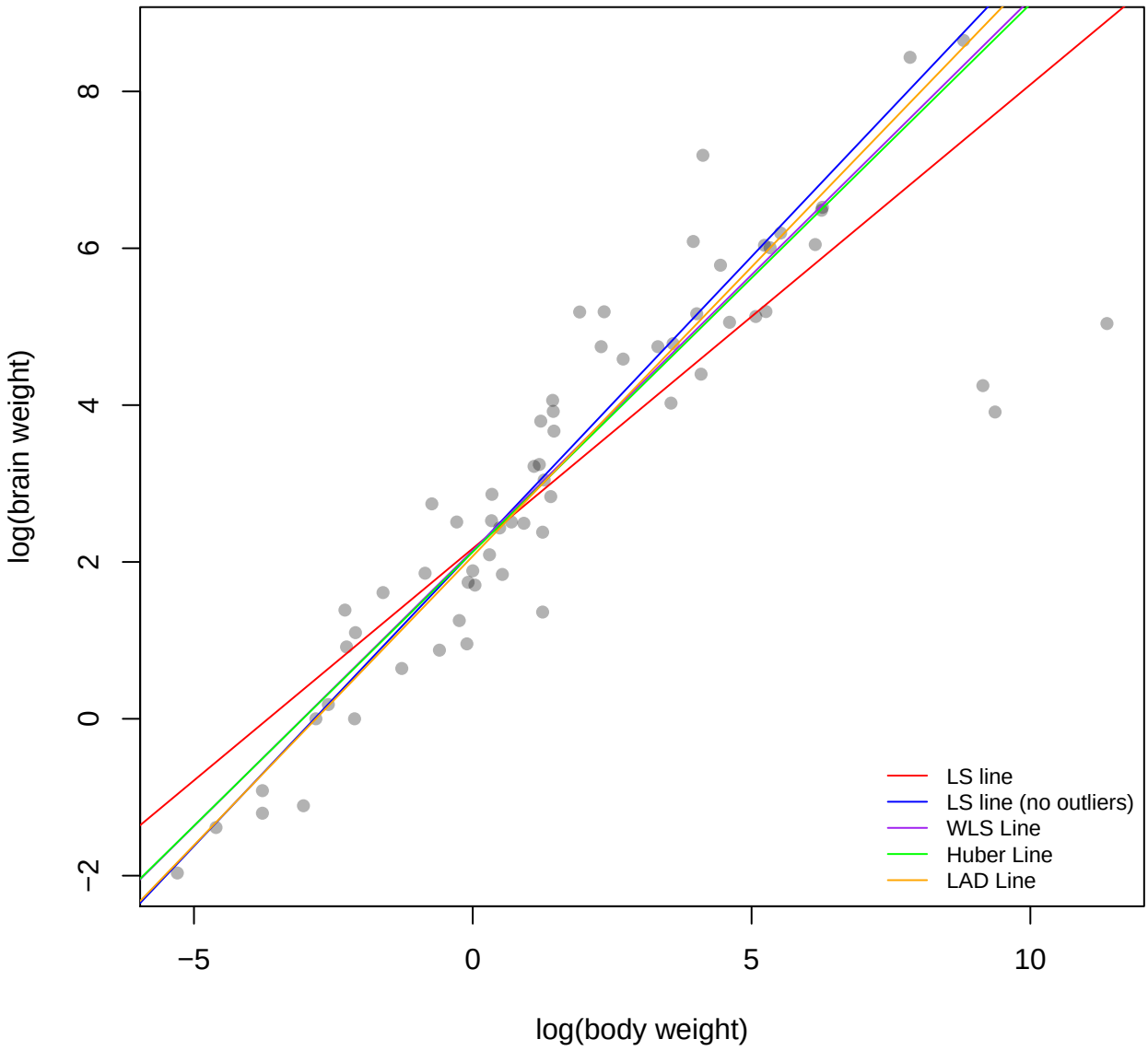
$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\operatorname{argmin}} \rho(\boldsymbol{\theta}; \mathcal{P})$$

Animals Example (Robust Regression)

Without going through the details of the aforementioned optimization methods, here we simply implement Huber and LAD-based linear regression in the context of the `Animals` data to illustrate their robustness to the dinosaurs. We do so using existing `R` functions without explaining their implementation (yet).

```
plot(x = log(Animals2$body), y = log(Animals2$brain),
     main = "",
     xlab = "log(body weight)", ylab = "log(brain weight)",
     pch = 16, col = adjustcolor("black", alpha.f = 0.3))

abline( lm(log(Animals2$brain) ~ log(Animals2$body)), col = "red" )
```

```r
W = rep(1,65)
W[which(log(Animals2$body)>9)]=0
abline( lm(log(Animals2$brain) ~ log(Animals2$body),weights=W), col = "blue" )
wt = rep(1,65)
wt[obs] = remaining.sd/outlier.sd
abline( lm(log(Animals2$brain) ~ log(Animals2$body), weights=wt), col = "purple")

library(MASS)
abline( rlm(log(Animals2$brain) ~ log(Animals2$body), psi="psi.huber"), col = "green")

library(L1pack)
abline( lad(log(Animals2$brain) ~ log(Animals2$body)), col = "orange")

legend("bottomright",
       legend=c("LS line", "LS line (no outliers)", "WLS Line", "Huber Line", "LAD Line"),
       col = c("red", "blue", "purple", "green", "orange"),
       cex = 0.75, bty = "n", lty = 1)
```



- Which line best represents the relationship in the population? Not red, otherwise it's up to you.

Review so far

- Populations: collections of units we want to study
  - assume we observe the value of one or more variates on every unit.
- Attributes: summaries of some characteristic of the population
  - numeric summaries
  - graphical summaries
  - solutions to some optimization problem

* evaluating attributes → sensitivity and invariance / equivariance
* sometimes re-expressing the data via a power transformation can be helpful

Next we'll spend time discussing various methods for finding solutions to optimization problems numerically.

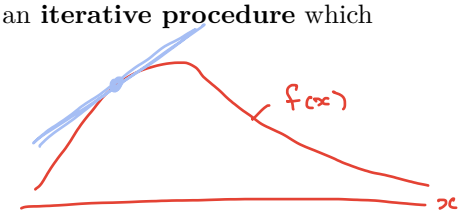
2.3.2 Gradient Descent

Given an implictly defined attribute of interest $\boldsymbol{\theta}$, 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

$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}\ \in\ \boldsymbol{\Theta}}{\operatorname{argmin}}\, \rho(\boldsymbol{\theta}; \mathcal{P})$$



.

- Ideally, each iterate is closer to the solution $\widehat{\boldsymbol{\theta}}$ than the one before it.
- Assuming we are at some point $\widehat{\boldsymbol{\theta}}_i$ on the surface of $\rho(\boldsymbol{\theta}; \mathcal{P})$, we achieve this by moving away from $\widehat{\boldsymbol{\theta}}_i$ in a direction which takes us to a point on the surface that is lower than at our current position, i.e., $\rho(\widehat{\boldsymbol{\theta}}_{i+1}; \mathcal{P}) < \rho(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})$.
- The direct of this movement is defined by the **gradient** of the surface.

Direction and Step Size

If $\rho(\boldsymbol{\theta}; \mathcal{P})$ is a **differentiable** function of $\boldsymbol{\theta} \in \mathbb{R}^k$ then we can calculate the **gradient** for any value of $\boldsymbol{\theta}$:

$$\boldsymbol{g} = \boldsymbol{g}(\boldsymbol{\theta}) = \nabla \rho(\boldsymbol{\theta}; \mathcal{P}) = \begin{bmatrix} \frac{\partial \rho(\boldsymbol{\theta}; \mathcal{P})}{\partial \theta_1} \\ \frac{\partial \rho(\boldsymbol{\theta}; \mathcal{P})}{\partial \theta_2} \\ \vdots \\ \frac{\partial \rho(\boldsymbol{\theta}; \mathcal{P})}{\partial \theta_k} \end{bmatrix}$$

- Note that we will typically distinguish among the gradient calculations at each iteration
- At iteration i, when $\widehat{\boldsymbol{\theta}}_i$ is our best guess at the solution, we denote the gradient by $\boldsymbol{g}_i = \boldsymbol{g}(\widehat{\boldsymbol{\theta}}_i)$.

By definition, the normalized gradient

$$\boldsymbol{d}_i = \frac{\boldsymbol{g}_i}{||\boldsymbol{g}_i||}$$

provides the **direction** in which $\rho(\boldsymbol{\theta}; \mathcal{P})$ increases or decreases fastest. (Recall that $||\boldsymbol{x}|| = \sqrt{x_1^2 + x_2^2 + \cdots + x_k^2}$ for $\boldsymbol{x} \in \mathbb{R}^k$). In particular:

- $\boldsymbol{d}_i$ indicates the direction of **steepest ascent**, and
- $-\boldsymbol{d}_i$ indicates the direction of **steepest descent**

We iterate and obtain a new estimate of $\boldsymbol{\theta}$ by

- moving in the direction of $-\boldsymbol{d}_i$ and
- taking a step of size $\lambda_i > 0$

start

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

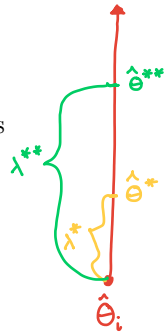
Note that the step size λ at each iteration can been chosen in a variety of ways:

* we could use g_i in our updating formula, but then λ_i would have to account for this.

1. We could choose a fixed value for all i such as $\lambda_i = 0.1$
2. We could define a fixed sequence such as $\lambda_i = 0.1 + 1/i$
3. We could perform a **line search** and algorithmically choose the value of λ_i that minimizes

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

In other words, which step size in the direction $-\boldsymbol{d}_i$ away from $\widehat{\boldsymbol{\theta}}_i$, minimizes $\rho\left(\widehat{\boldsymbol{\theta}}_{i+1}; \mathcal{P}\right)$?



The Gradient Descent Algorithm

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

1. **Initialize** ; $i \leftarrow 0$;
2. LOOP:
   a. **Gradient**:
   $$\boldsymbol{g}_i = \nabla \rho(\boldsymbol{\theta}; \mathcal{P})|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}_i}$$
   b. Gradient **direction**:
   $$\boldsymbol{d}_i \leftarrow \frac{\boldsymbol{g}_i}{||\boldsymbol{g}_i||}$$
   c. **Line search**: Find the step size $\widehat{\lambda}_i$
   $$\widehat{\lambda}_i = \arg\min_{\lambda > 0} \rho\left(\widehat{\boldsymbol{\theta}}_i - \lambda \, \boldsymbol{d}_i\right)$$
   d. **Update** the iterate:
   $$\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}_i - \widehat{\lambda}_i \, \boldsymbol{d}_i$$
   e. **Converged?** $\hat{\theta}_i$ vs. $\hat{\theta}_{i+1}$
   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

```
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))
}
```

Notes:

- Notice the gradient descent algorithm was defined even though none of the functions it requires exist!
  - These will need to be implemented for each particular function $\rho(\boldsymbol{\theta}; \mathcal{P})$ we wish to minimize.
- For production code we would do more error checking
  - We are avoiding this here to better illustrate the algorithm.

R code for Line Search and Test of 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 <- sapply(lambdas, function(lambda) {rhoFn(theta - lambda * g)})
  ## 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
}
```

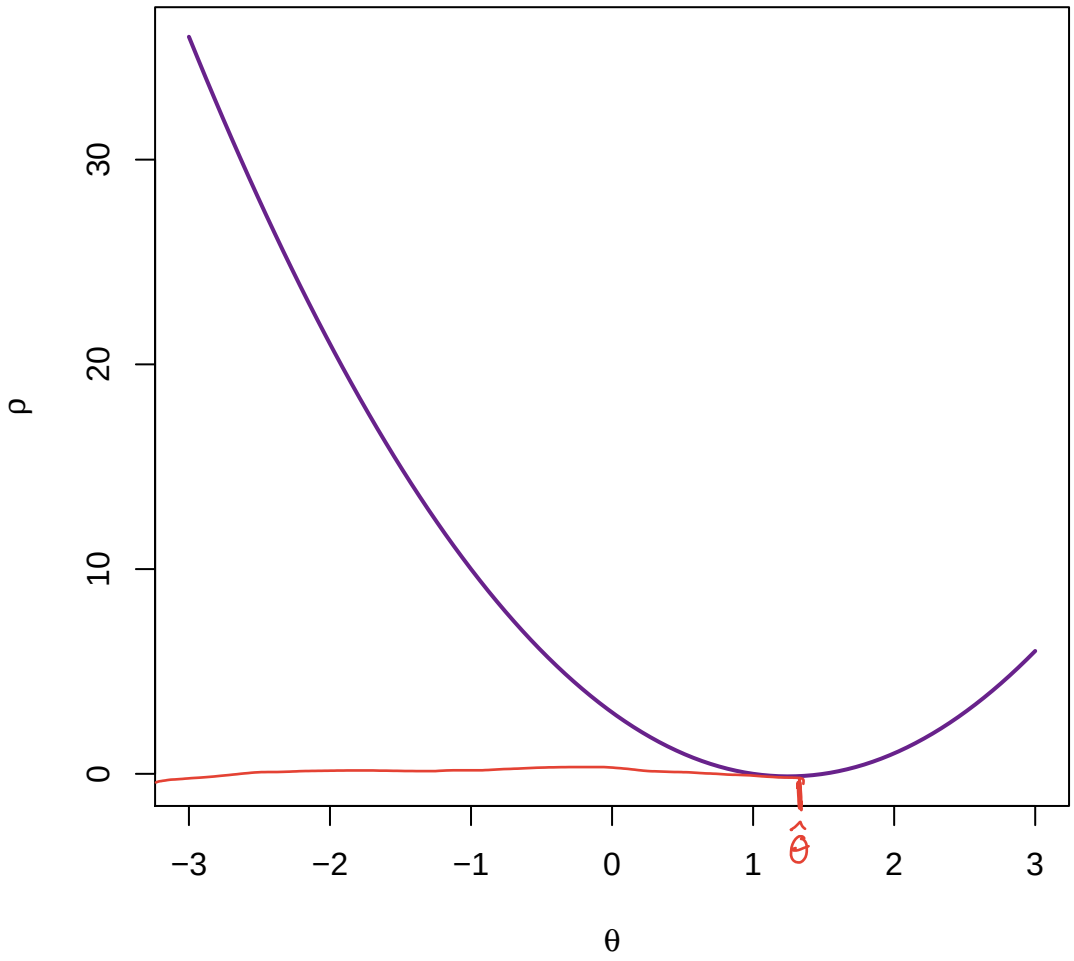
relative = FALSE $\longrightarrow$ $\|\hat{\theta}_i - \hat{\theta}_{i+1}\| < \varepsilon$ where here $\|\cdot\| = L_1$ norm

relative = TRUE $\longrightarrow$ $\dfrac{\|\hat{\theta}_i - \hat{\theta}_{i+1}\|}{\|\hat{\theta}_i\|} < \varepsilon$ $\quad \|z\| = \sum_{j=1}^{k} |z_j|$

Example 1 (one-dimensional quadratic function)

Suppose for example we're interested in finding the value of θ which minimizes

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



The gradient is

$$g = \rho'(\theta) = 4\theta - 5$$

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 using our `gradientDescent` function to find the value $\widehat{\theta}$ which minimizes $\rho(\theta)$.

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

```
## $theta
## [1] 1.25
```

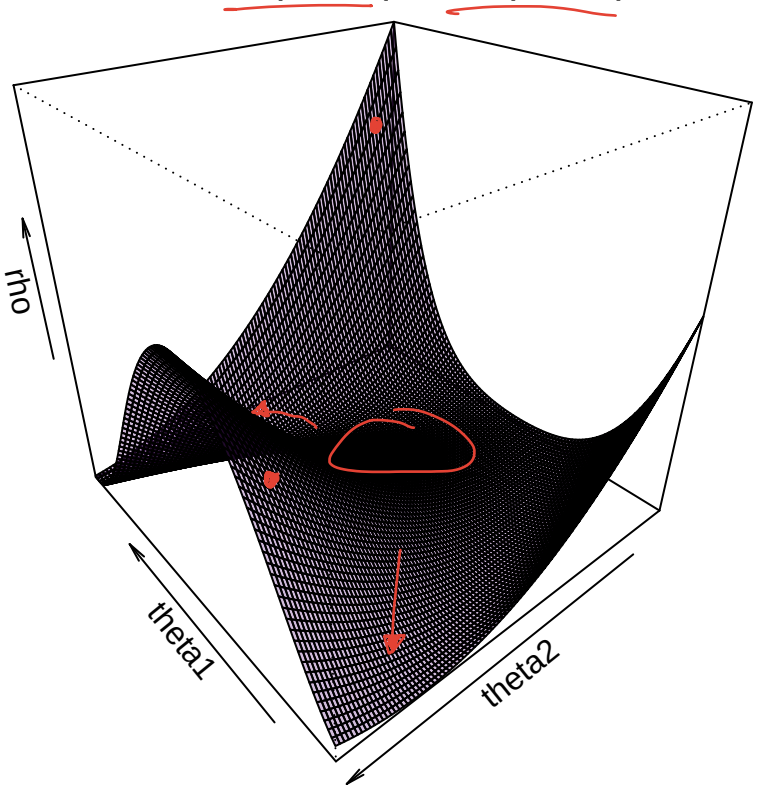
```
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 4
##
## $fnValue
## [1] -0.125
```

Example 2 (two-dimensional Rosenbrock function)

We want to find $\boldsymbol{\theta} = (\theta_1, \theta_2)^{\mathsf{T}}$ which minimizes the following Rosenbrock function

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

which is a **non-convex** function, usually used in performance test problems for optimization algorithms. A plot of this function is shown below for $\theta_1 \in [-1.5, 1.8]$ and $\theta_2 \in [-0.5, 3.0]$.



The gradient for this function is

$$\boldsymbol{g} = \nabla\rho(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial\rho}{\partial\theta_1} \\ \frac{\partial\rho}{\partial\theta_2} \end{bmatrix} = \begin{bmatrix} 400\theta_1^3 - 400\theta_1\theta_2 + 2\theta_1 - 2 \\ \\ -200\theta_1^2 + 200\theta_2 \end{bmatrix}$$

To use our `gradientDescent` function let's define $\rho(\boldsymbol{\theta})$ and $\nabla\rho(\boldsymbol{\theta})$ in `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] )
}
```

Use the code below to find $\widehat{\theta}$ which minimizes $\rho(\boldsymbol{\theta})$.

```
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
```

Example 3 (Least Squares Regression)

Suppose we have a population $\mathcal{P}$ which consists of pairs (x_u, y_u) and we wish to fit the following linear regression model to these data

$$y_u = \alpha + \beta(x_u - \overline{x}) + r_u$$

The attribute of interest is $\boldsymbol{\theta} = (\alpha, \beta)^{\mathsf{T}}$ and the objective function we need to minimize is

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

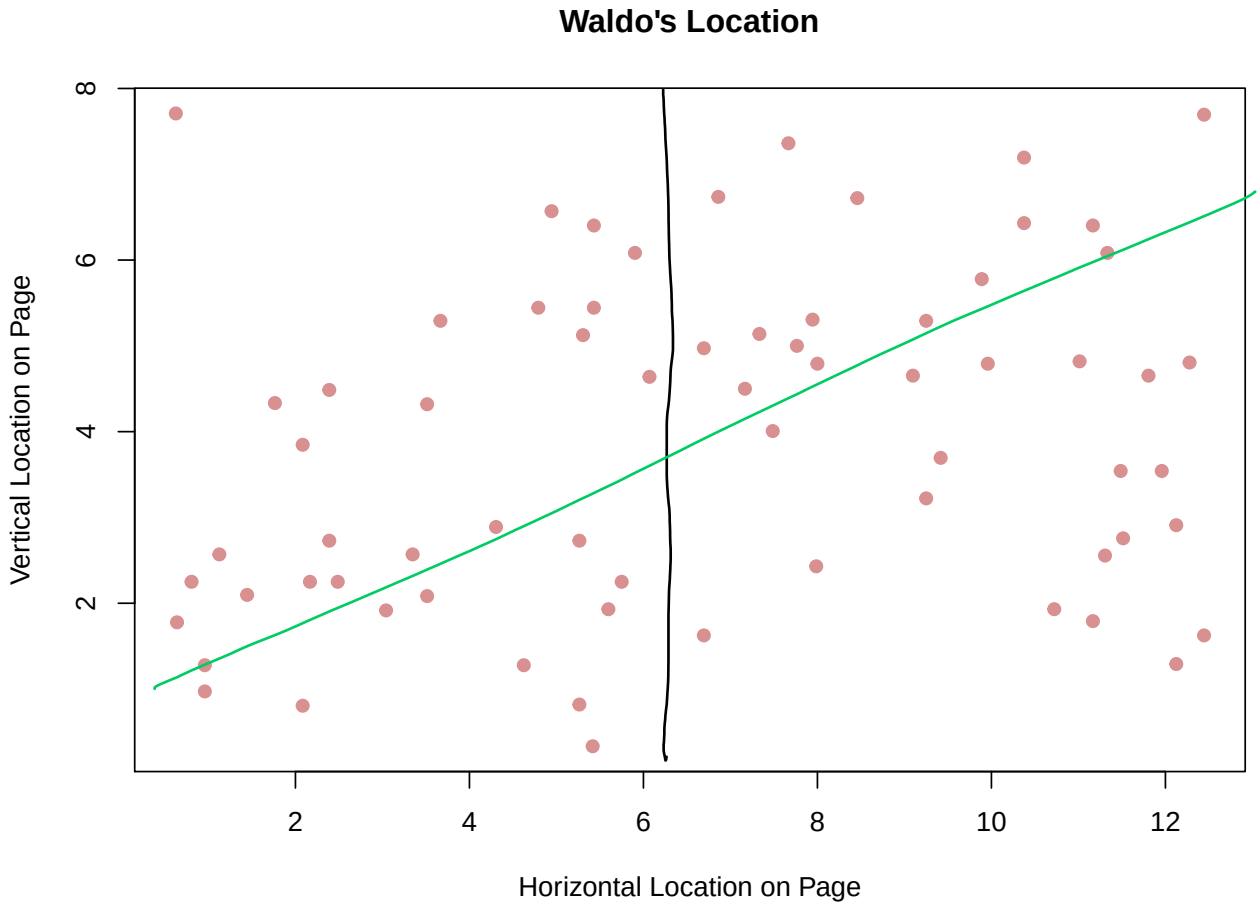
The corresponding gradient is

$$\boldsymbol{g} = \nabla\rho(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial \rho}{\partial \alpha} \\ \frac{\partial \rho}{\partial \beta} \end{bmatrix} = \begin{bmatrix} \sum_{u \in \mathcal{P}} -2\left(y_u - \alpha - \beta(x_u - \overline{x})\right) \\ \sum_{u \in \mathcal{P}} -2\left(y_u - \alpha - \beta(x_u - \overline{x})\right)(x_u - \overline{x}) \end{bmatrix}$$

Where's Waldo?

For this example we will take $\mathcal{P}$ to be the `Where's Waldo?` population for which x and y respectively represent the horizontal and vertical locations of `Waldo` in a given page in a given book.

```
waldo <- read.csv("../Data/wheres-waldo-locations.csv", header = TRUE)
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")
```



As with the previous two examples we need to write the corresponding `R` functions `rho` and `gradient`. However, unlike the previous two examples, these functions will now rely on data in addition to $\boldsymbol{\theta}$.

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

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

```
                sum((y - alpha - beta * (x - xbar)) * (x - xbar)))
  # Return g
  g
}
```

Notes:

- In the R code 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.

INTERLUDE: Avoiding Reliance on the Global Environment

- An obvious solution, and one which generally works, is to pass the needed variables (here `x` and `y`) to the relevant functions 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 exploit this property by creating functions that themselves define and return a function.
  - 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.

"Factory Function"

Functions that make Functions

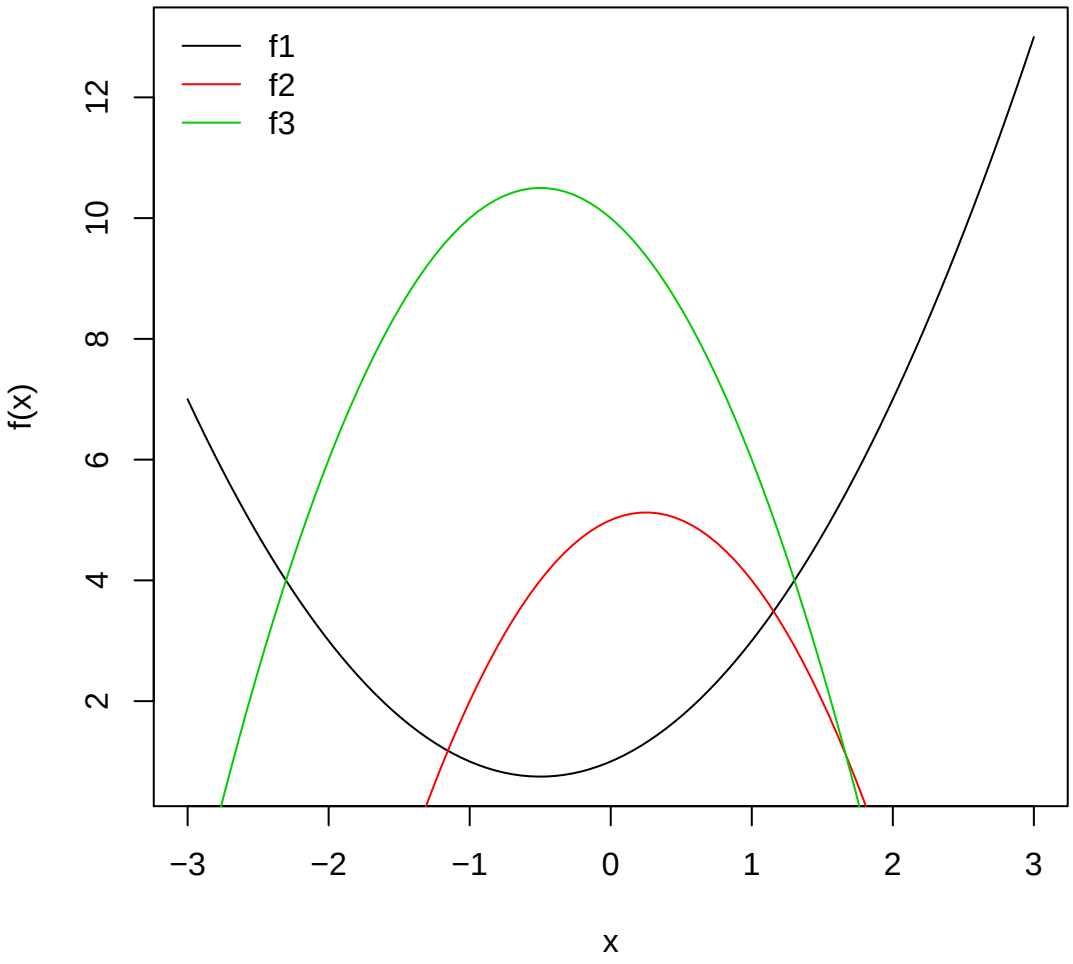
Here is a simple example of a function that defines and returns a quadratic function.

```
createQuadratic <- function(a, b, c) {
  ## Return this function
  function(x) {
```

```
    fx = a*x^2 + b*x + c
    return(fx)
  }
}
```

Here is 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)

legend("topleft", lty = 1, col = 1:3, legend = c("f1", "f2", "f3"), bty = "n")
```



In a similar fashion we can define functions that will take in the data (`x` and `y`) and return appropriate `rho` and `gradient` functions for least squares regression with any data:

```
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(sum(y - alpha - beta * (x - xbar)),
           sum((y - alpha - beta * (x - xbar)) * (x - xbar)))
  }
}
gradient <- createLeastSquaresGradient(waldo$X, waldo$Y)
```

Back to Waldo

The functions `rho` and `gradient` can now be passed as arguments to `gradientDescent(...)` to find the value $\widehat{\boldsymbol{\theta}} = \left(\widehat{\alpha}, \widehat{\beta}\right)^{\mathsf{T}}$ that minimizes

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

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

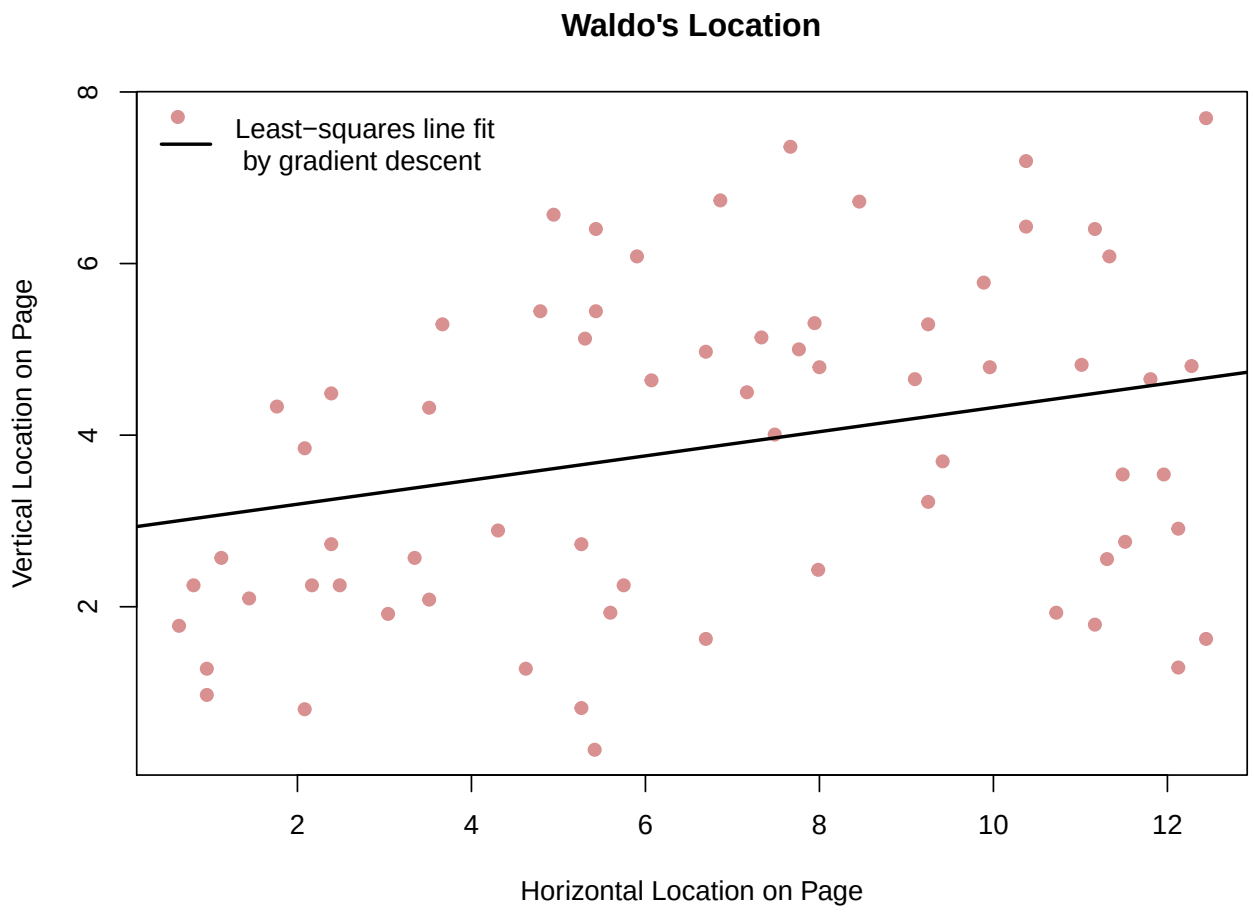
```
## $theta
## [1] 3.857138 0.140886
##
## $converged
## [1] TRUE
##
```

```
## $iteration
## [1] 29
##
## $fnValue
## [1] 233.7737
```

min

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",
     xlab="Horizontal Location on Page", ylab= "Vertical Location on Page")
abline(result$theta[1] - result$theta[2] * mean(waldo$X), result$theta[2], lwd=2)
legend("topleft", lty = 1, col = 1, lwd = 2,
       legend = "Least-squares line fit \n by gradient descent", bty = "n")
```



Why do we care about this line?

This line summarizes the population which gives us a strategy for finding Waldo. We can move our eyes across this line and this should show us where Waldo is most likely to be.

Example 4 (Robust Regression)

As in the previous example, let us suppose we have a population $\mathcal{P}$ which consists of pairs (x_u, y_u) and we wish to fit the following linear regression model to these data

$$y_u = \alpha + \beta(x_u - \overline{x}) + r_u$$

The attribute of interest is again $\boldsymbol{\theta} = (\alpha, \beta)^\mathsf{T}$ but now let's assume we wish to perform **robust regression** in which case the objective function to be minimized is

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

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

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

is the Huber function.

The corresponding gradient is

$$\boldsymbol{g} = \nabla \rho(\boldsymbol{\theta})$$

which can be decomposed using the Chain Rule as follows:

$$\nabla \rho(\boldsymbol{\theta}) = \frac{\partial}{\partial \boldsymbol{\theta}} \sum_{u \in \mathcal{P}} \rho_k(r_u) = \sum_{u \in \mathcal{P}} \frac{\partial \rho_k(r_u)}{\partial \boldsymbol{\theta}} = \sum_{u \in \mathcal{P}} \frac{\partial \rho_k(r_u)}{\partial r_u} \times \frac{\partial r_u}{\partial \boldsymbol{\theta}}$$

where

$$\frac{\partial \rho_k(r)}{\partial r} = \begin{cases} r & \text{for } |r| \leq k \\ \text{sign}(r) \times k & \text{for } |r| > k \end{cases}$$

and

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

Thus the gradient can be written as

$$\boldsymbol{g} = - \begin{bmatrix} \sum_{u \in \mathcal{P}} \frac{\partial \rho_k(r_u)}{\partial r_u} \\ \sum_{u \in \mathcal{P}} \frac{\partial \rho_k(r_u)}{\partial r_u} (x_u - \overline{x}) \end{bmatrix}$$

The following are `R` functions for the Huber function and its derivative as well as *creator* functions which, given data, create `rho(...)` and `gradient(...)` functions for simple linear robust regression.

The Huber function

```
huber.fn <- function(r, k) {
  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

```
createRobustHuberRho <- function(x, y, kval) {
  ## 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

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

A function to create the gradient function

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

Animals

Here we will use gradient descent with the functions defined above to perform robust regression on the `Animals` data that we've worked with previously. The necessary R code to do so is provided below:

```
rho  <- createRobustHuberRho(x = log(Animals2$body), y = log(Animals2$brain),
                             kval=0.766*1.5)

gradient <- createRobustHuberGradient(x = log(Animals2$body),
                                      y = log(Animals2$brain),
                                      kval=0.766*1.5)

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

print(result)
```

```
## $theta
## [1] 3.3308999 0.6909751
##
## $converged
```

argmin

```
## [1] TRUE
##
## $iteration
## [1] 20
##
## $fnValue
## [1] 28.82349
```

min

Let's compare these results to what we found previously (which relied on the `rlm` function in the `MASS` package):

```
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

In practice, many of the objective functions minimized during statistical analyses have the following form:

$$\rho(\boldsymbol{\theta}, \mathcal{P}) = \sum_{u \in \mathcal{P}} \rho(\boldsymbol{\theta}; u)$$

in which case the gradient $\boldsymbol{g}$ can simply be written as the sum of the unit-specific contributions to the objective function:

$$\boldsymbol{g} = \boldsymbol{g}(\boldsymbol{\theta}) = \nabla\rho(\boldsymbol{\theta}; \mathcal{P}) = \sum_{u \in \mathcal{P}} \nabla\rho(\boldsymbol{\theta}; u) \equiv \sum_{u \in \mathcal{P}} \boldsymbol{g}(\boldsymbol{\theta}; u)$$

Thus when $\rho(\cdot)$ is a sum over $u \in \mathcal{P}$:

- the gradient $\boldsymbol{g}$ is composed of N 'smaller' independent gradient calculations
- these individual gradient calculations $\boldsymbol{g}(\boldsymbol{\theta}; u)$ can be done in any order
  - this can be very handy when N is large and the individual gradients are expensive to calculate
  - we may wish to perform the gradient computations in several batches which are distributed across different machines
  - this is important in many "Big Data" applications
- the terms **batch gradient descent** and **gradient descent** are used interchangeably ← *
  - The appropriateness of the term **batch** becomes clear when we explicitly partition the population $\mathcal{P}$ into H non-overlapping groups (batches) $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \cdots \cup \mathcal{P}_H$ each containing M_k units $(k = 1, 2, \ldots, H)$:

$$\boldsymbol{g} = \sum_{u \in \mathcal{P}} \boldsymbol{g}(\boldsymbol{\theta}; u) = \sum_{k=1}^{H} \sum_{u \in \mathcal{P}_k} \boldsymbol{g}(\boldsymbol{\theta}; u)$$

* the concepts of map and reduce are relevant

sum of gradients in batch k

sum across batches

* Batch gradient descent lends itself well to parallel computation, but what if the gradient calculations are sufficiently complex and even parallelization isn't fast enough?

What if we didn't use all of the data?

Gradient Descent Using Subsets of the Population

When computing the gradient $\boldsymbol{g}$ is computationally expensive we could consider using only a subset of the available data – as opposed to using all of it.

- If the run time for batch gradient descent based on all N units is too long, consider *estimating* the gradient using just $M < N$ units
- In such situations we typically do not optimize for the step size λ and instead use a fixed step size $\lambda^\star$ (Why?) Gradients in this case are only approximate, so we typically don't optimize the step size when our direction is not exactly correct.
  - $\lambda^\star$ is often referred to as the **learning rate**
- Two common approaches to do this are **batch-sequential** and **batch-stochastic** gradient descent
  - these approaches differ only in the manner in which the the subsets of size M are chosen.

Batch-Sequential Gradient Descent

$P = \{P_1, P_2, P_3, \ldots, P_H\}$

In this approach we sequentially move through the H batches and update our estimate $\widehat{\boldsymbol{\theta}}$ after *each* batch. Note that this is different from ordinary batch gradient descent; in that case the gradients are still calculated in batches, but the $\widehat{\boldsymbol{\theta}}$ is only updated *after observing all batches*. If convergence takes more than H iterations then the batches are iteratively sequenced through until convergence.

- The **batch-sequential gradient descent algorithm** is the following:

  Given some initial value $\widehat{\boldsymbol{\theta}}_0$ and a fixed step size $\lambda^\star$

1. **Initialize** ; $i \leftarrow 0$;
2. LOOP:
   a. **Gradient**:
   $$\boldsymbol{g}_i = \nabla \rho(\boldsymbol{\theta}; \mathcal{P}_{i \bmod H})|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}_i}$$
   for this notation to work, define $P_0 \equiv P_H$
   b. Gradient **direction**:
   $$\boldsymbol{d}_i \leftarrow \frac{\boldsymbol{g}_i}{||\boldsymbol{g}_i||}$$
   c. **Update** the iterate:
   $$\widehat{\boldsymbol{\theta}}_{i+1} \leftarrow \widehat{\boldsymbol{\theta}}_i - \lambda^\star \, \boldsymbol{d}_i$$
   d. **Converged?**

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

   else $i \leftarrow i + 1$ and repeat LOOP.
3. **Return**: $\widehat{\boldsymbol{\theta}} = \widehat{\boldsymbol{\theta}}_i$

* here we use all of the data but it can be faster than ordinary GD because our batches are much smaller than the full population.

much more common in practice than batch-sequential

Batch-Stochastic Gradient Descent

In this approach, each iteration of the gradient is calculated from a sample (batch) $\mathcal{S}$ selected randomly from the population $\mathcal{P}$. Like batch-sequential gradient descent, the estimate $\widehat{\boldsymbol{\theta}}$ is updated after *each* batch (sample). Denoting the sample size by M, note that setting $M = 1$ gives rise to what is often simply referred to as **stochastic gradient descent**.

- The **batch-stochastic gradient descent** algorithm is the following:

  Given some initial value $\widehat{\boldsymbol{\theta}}_0$ and a fixed step size $\lambda^\star$

1. **Initialize** ; $i \leftarrow 0$;
2. LOOP:
   a. **Gradient**: Given a **new** random sample $\mathcal{S} \in \mathcal{P}$

   $$\boldsymbol{g}_i = \nabla \rho(\boldsymbol{\theta}; \mathcal{S})|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}_i}$$

   b. Gradient **direction**:

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

   c. **Update** the iterate:

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

   d. **Converged?**

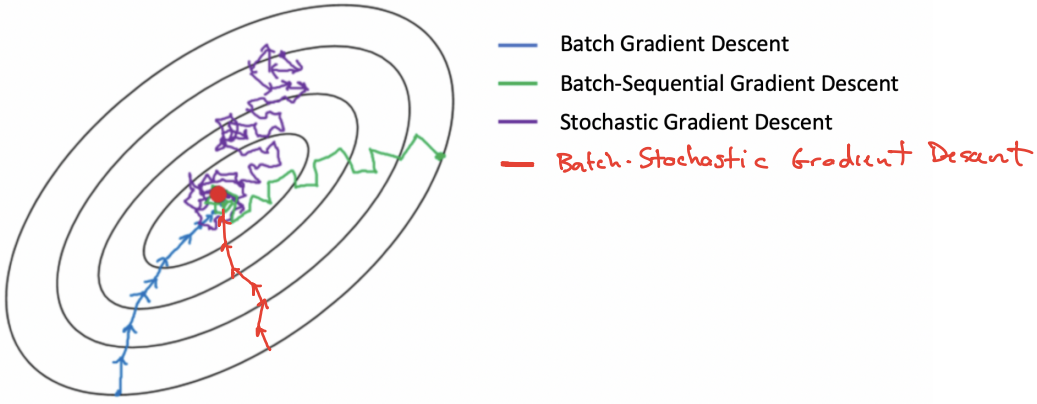
   if the iterates are not changing, then **Return**

   else $i \leftarrow i + 1$ and repeat LOOP.
3. **Return**: $\widehat{\boldsymbol{\theta}} = \widehat{\boldsymbol{\theta}}_i$

* outliers are less likely to have an influence here

* a random sample estimate of the gradient is probably going to be better than an estimate based on a systematically chosen batch.

Comparing the Algorithms



* this picture communicates efficiency in terms of number of steps, but not in terms of time.

R code for Batch-Stochastic Gradient Descent

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

No! But we need to amend our gradient functions.

- We have to edit the `gradient` function. For example, here is a creator function that will create the batch-stochastic gradient function for robust regression with the Huber function.

factory

```
createRobustHuberGradient.stochastic <- function(x,y, kval, M) {
  ## local variables
  xbar <- mean(x)
  ybar <- mean(y)
  function(theta) {
    alpha <- theta[1]
    beta <- theta[2]
    ru = y - alpha - beta*( x - xbar)
    rhok = huber.fn.prime(ru, k=kval)

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

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

- And if we want to use fixed step sizes we need to write a function that does this

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

- Let's put this together and perform stochastic gradient descent when performing robust regression on the `Animals` data:

```
rho  <- createRobustHuberRho(x = log(Animals2$body), y = log(Animals2$brain), kval = 0.766*1.5)

gradient.stochastic <- createRobustHuberGradient.stochastic(x = log(Animals2$body),
                                                            y = log(Animals2$brain),
                                                            kval = 0.766*1.5, M = 10)

set.seed(341)
result2 <- gradientDescent(theta = c(1,0), rhoFn = rho, gradientFn = gradient.stochastic,
      lineSearchFn = fixedLineSearch, testConvergenceFn = testConvergence,
      maxIterations=10^5, tolerance = 1E-3, lambdaStepsize = 0.01, relative = TRUE)

print(result2)
```

λ^*

```
## $theta
## [1] 3.3669859 0.7136107
##
## $converged
## [1] FALSE
##
## $iteration
## [1] 100001
##
## $fnValue
## [1] 28.92222
```

arg min

min

Compare this to the result with ordinary (batch) gradient descent

```
print(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 `M`. Any value `M>1` results in batch-stochastic gradient descent.

2.3.3 Solving a System of Equations

We have defined an **implicit attribute** $\boldsymbol{\theta}$ as the solution to an optimization problem. Until now we have cast such an optimization problem as the minimization of some appropriately defined objective function $\rho(\boldsymbol{\theta}; \mathcal{P})$:

$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\operatorname{argmin}}\, \rho(\boldsymbol{\theta}; \mathcal{P})$$

In all situations we saw that such a solution was obtained by solving (either algorithmically or in closed form) the equation

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

When the dimension of the vector valued attribute $\boldsymbol{\theta}$ is k, such an equation is actually a system of k independent equations with k unknowns. Thus an implicit attribute $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ can be defined slightly more generally as the solution to a system of equations

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

Thus we work directly with $\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P})$ which, in most cases, equals $\nabla \rho(\boldsymbol{\theta}; \mathcal{P})$.

Unsurpsingly, $\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P})$ is often defined as a sum over $u \in \mathcal{P}$:

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

in which case the attribute of interest $\widehat{\boldsymbol{\theta}}$ is the value $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ that solves

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

Examples (scalar valued attributes)

Recall our familiar examples for implicitly defined scalar attributes θ

- Previously these were defined as solutions to minimization problems
- Now we defne them as solutions to systems of equations

The average is the value of θ which solves

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

The weighted average is the value of θ which solves

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

The q^{th} quantile: is the smallest value of θ which solves

$$\Psi(\theta; P) = \sum_{u \in \mathcal{P}} \psi(\theta; u) = \sum_{u \in \mathcal{P}} \frac{1}{N} I\left(y_u \leq \theta\right) - q = 0$$

Examples (vector valued attributes)

Recall our interest in the vector valued attribute $\boldsymbol{\theta} = (\alpha, \beta)$ in the context of a simple linear regression. Different forms of linear regression arose by determining the vector $\widehat{\boldsymbol{\theta}} = (\widehat{\alpha}, \widehat{\beta})$ which was the solution to a variety of different systems of equations.

Least squares

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

Weighted least squares

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

Robust regression

$$\Psi(G; P) = \sum_{u \in \mathcal{P}} \boldsymbol{\psi}(\boldsymbol{\theta}; u) = \sum_{u \in \mathcal{P}} \rho_k'(y_u - \alpha - \beta(x_u - c)) \begin{pmatrix} 1 \\ \\ x_u - c \end{pmatrix} = \mathbf{0}$$

The class of methods used to find such solutions to systems of equations are generally referred to as **root finding** methods.

Newton's Method

Suppose we have a differentiable function $f(x)$ and we wish to find x^* which solves

$$f(x) = 0$$

Given an initial value x_0

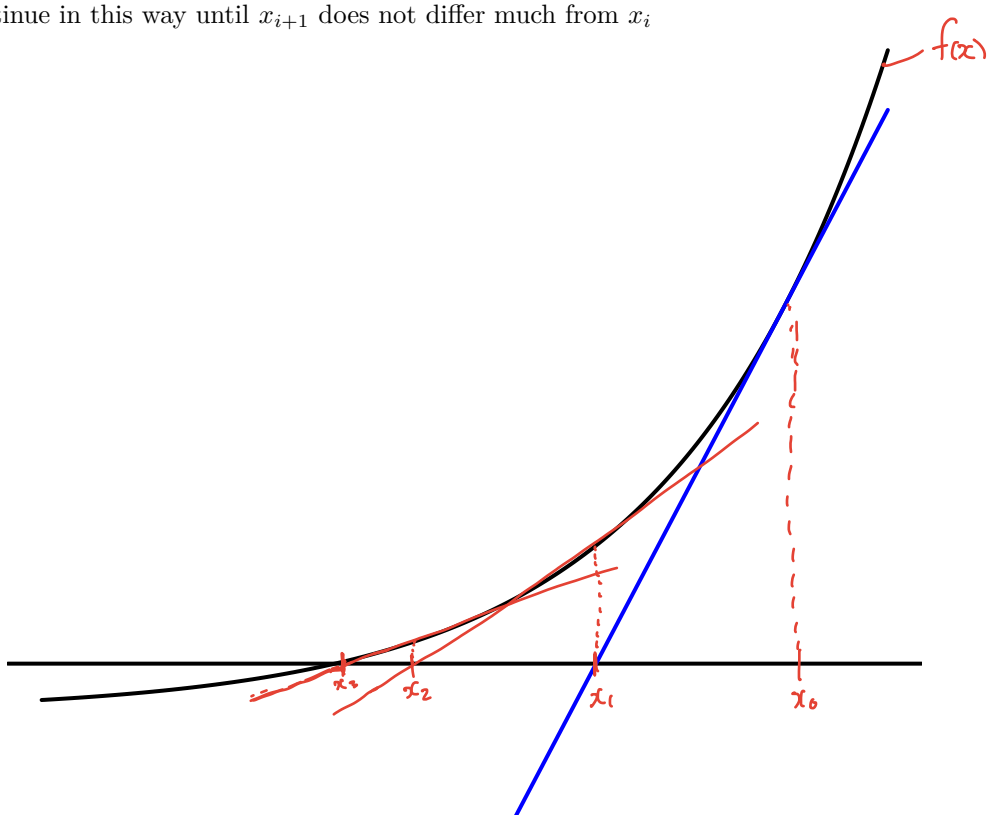
- we can use a linear function to approximate $f(x)$ in the vicinity of x_0

$$f(x) \approx f(x_0) + f'(x_0)\,(x - x_0)$$ ← First order Taylor Approx.

- then we find the root of the linear approximation to iterate to the next value of x:

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

- We continue in this way until x_{i+1} does not differ much from x_i



For our implicitly defined scalar attribute θ, we want to find $\theta \in \Theta$ such that

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

- Given the current guess $\widehat{\theta} = \widehat{\theta}_i$ a first order approximation is

$$\psi(\theta; \mathcal{P}) \approx \psi(\widehat{\theta}_i; \mathcal{P}) + \psi'(\widehat{\theta}_i; \mathcal{P}) \times \left(\theta - \widehat{\theta}_i\right).$$

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

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

The Newton Algorithm

Given an initial value $\widehat{\theta}_0$

1. **Initialize** ; $i \leftarrow 0$;
2. LOOP:
   a. **Update** the iterate:

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

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

R Code for Newton's Method

Here is a basic `R` implementation of Newton's method:

```
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)
```

```
    )
}
```

- Similar to our `gradientDescent` function we will need to write context-specific `psiFn(...)` and `psiPrimeFn(...)` functions if we want to use this in practice.
- We can reuse the `testConvergence` function that we wrote for use with `gradientDescent` in this context

Example (Weighted Average)

Let's consider the `facebook` dataset. 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.).

```
facebook <- read.csv("/Users/nstevens/Dropbox/Teaching/STAT_341/Lectures/Data/facebook.csv",
                     header = TRUE)
### Remove all rows with missing data
fb <- na.omit(facebook)
head(fb[,c(2,3,4,1,6)])
```

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

- Interest lies in determining θ, the typical number of `likes` a post receives
  - Let's account for the `Impressions` for each post since a post that is seen a lot will tend to have more likes
  - Let's do this by weighting `like` by the inverse of `Impressions` so that the more often a post is seen, the lower the weight we assign to its number of `likes`
- The solution to the weighted sum

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

can now be found via Newton's Method by defining the appropriate `psi(...)` and `psiPrime(...)` functions.

- In this case $\psi'(\theta; \mathcal{P})$ is given by

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

- And the `R` implementations are as follows:

```
### 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
```

- Passing these functions into `Newton(...)` yields the following:

```
result <- Newton(theta = mean(fb$like),
                 psiFn = psi, psiPrimeFn = psiPrime)
print(result)
```

```
## $theta
## [1] 78.46845
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 2
##
## $fnValue
## [1] -4.346092e-16
```

- **Note:** we have a closed form expression for the weighted average

$$\frac{\sum_{u \in \mathcal{P}} w_u y_u}{\sum_{u \in \mathcal{P}} w_u}$$

and so the result above should agree with this closed form solution:

```
y <- fb$like
wt <- 1/fb$Impressions
c(NewtonSol = result$theta, weightedAve = sum(wt*y) / sum(wt) )
```

```
##   NewtonSol weightedAve
##    78.46845    78.46845
```

- **Question:** Why did this converge in two iterations?

Because $\psi(\theta; \mathcal{P})$ is a linear function of θ.

- **Exercise:** Try out some other ψ functions for location estimation.

  + e.g. Andrew's sine function, the bisquare weight function, Huber's psi, or Hampel's psi

The Newton-Raphson Method

When $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_k)^\mathsf{T}$ is a vector-valued attribute, the multivariate analog to Newton's Method is referred to as the **Newton-Raphson Method**. In this case we want the vector which solves

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

Since $\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P})$ is a differentiable $k \times 1$ *vector* $(\psi_1, \psi_2, \ldots, \psi_k)^\mathsf{T}$ we let

$$\boldsymbol{\psi}'(\boldsymbol{\theta}; \mathcal{P}) = \frac{\partial\ \boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P})}{\partial \boldsymbol{\theta}} = \begin{bmatrix} \frac{\partial \psi_1}{\partial \theta_1} & \frac{\partial \psi_1}{\partial \theta_2} & \cdots & \frac{\partial \psi_1}{\partial \theta_k} \\ \frac{\partial \psi_2}{\partial \theta_1} & \frac{\partial \psi_2}{\partial \theta_2} & \cdots & \frac{\partial \psi_2}{\partial \theta_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \psi_k}{\partial \theta_1} & \frac{\partial \psi_k}{\partial \theta_2} & \cdots & \frac{\partial \psi_k}{\partial \theta_k} \end{bmatrix}$$

be the $k \times k$ *matrix* of partial derivatives. This matrix is called the *Jacobian* matrix of $\boldsymbol{\psi}$

- Given a current guess $\widehat{\boldsymbol{\theta}}_i$ we can use a linear function to approximate the function $\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P})$
  - A first-order approximation of $\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P})$ in the vicinity of $\widehat{\boldsymbol{\theta}}_i$ can be written as

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

  - Then the vector at which the linear approximation is equal to zero is

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

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

The Newton-Raphson Algorithm

Given an initial value $\widehat{\boldsymbol{\theta}}_0$

1. **Initialize** ; $i \leftarrow 0$;
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**: $\widehat{\boldsymbol{\theta}} = \widehat{\boldsymbol{\theta}}_i$

R Code for the Newton-Raphson Algorithm

```
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))
}
```

Note that `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
   $$\left[\psi'(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})\right]^{-1}$$
   is the value of $\boldsymbol{x}$ that solves the system of equations
   $$\left[\psi'(\widehat{\boldsymbol{\theta}}_i; \mathcal{P})\right] \boldsymbol{x} = \boldsymbol{I}_k$$
   which is precisely what the function `solve(...)` does.

Example (Rosenbrock Function)

Suppose we are interested in finding $\boldsymbol{\theta} = (\theta_1, \theta_2)$ which minimizes the Rosenbrock function

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

With the problem framed as *solving a system of equations*, the roots of the gradient are of importance:

$$\boldsymbol{\psi}(\boldsymbol{\theta}) = \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix} \equiv \boldsymbol{g} = \nabla\rho(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial\rho}{\partial\theta_1} \\ \frac{\partial\rho}{\partial\theta_2} \end{bmatrix} = \begin{bmatrix} 400\theta_1^3 - 400\theta_1\theta_2 + 2\theta_1 - 2 \\ -200\theta_1^2 + 200\theta_2 \end{bmatrix}$$

And for Newton-Raphson we need the matrix of partial derivatives of $\boldsymbol{\psi}(\boldsymbol{\theta})$ (i.e., the Jacobian of $\boldsymbol{\psi}(\boldsymbol{\theta})$):

$$\boldsymbol{\psi}'(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial\psi_1}{\partial\theta_1} & \frac{\partial\psi_1}{\partial\theta_2} \\ \frac{\partial\psi_2}{\partial\theta_1} & \frac{\partial\psi_2}{\partial\theta_2} \end{bmatrix} = \begin{bmatrix} 1200\,\theta_1^2 - 400\,\theta_2 + 2 & -400\,\theta_1 \\ -400\,\theta_1 & 200 \end{bmatrix}$$

```
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] ) }
```

The value of $\widehat{\theta}$ as determined by Newton-Raphson is as follows:

```
NRresult <- NewtonRaphson(theta= c(0,0), psiFn = g, psiPrimeFn = psiPrime)

print(NRresult)
```

```
## $theta
## [1] 1 1
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 3
##
## $fnValue
## [1] 0 0
```

NR converged to the global min whereas GD converged to a local min.

Let's compare that to the value of $\widehat{\theta}$ we found with gradient descent:

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

print(GDresult)
```

```
## $theta
## [1] 0.7987955 0.6278095
##
```

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

- Let's check the sensitivity of these results to the starting value $\boldsymbol{\theta}_0$
  - Ideally an algorithm's performance shouldn't depend too much on the initial guess
  - But in practice, an accurate initial guess can make a huge difference
- Rather than starting at $\boldsymbol{\theta}_0 = (0, 0)$, let's start the algorithms at $\boldsymbol{\theta}_0 = (-2, -1)$

Newton-Raphson:

```
NRresult <- NewtonRaphson(theta= c(0-2,-1), psiFn = g, psiPrimeFn = psiPrime, maxIterations=10^4)

print(NRresult)
```

```
## $theta
## [1] 1.000133 1.000266
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 4195
##
## $fnValue
## [1]  2.660508e-04 -2.214051e-11
```

Gradient Descent:

```
GDresult = gradientDescent(theta= c(-2,-1), rhoFn = rho, gradientFn = g,
                lineSearchFn = gridLineSearch,
                testConvergenceFn = testConvergence, maxIterations=10^4)

print(GDresult)
```

```
## $theta
## [1] 0.7989451 0.6280480
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 426
##
## $fnValue
## [1] 0.05096056
```

Connection between Newton-Raphson and Gradient Descent

- Notice that when the objective function of interest is $\rho(\boldsymbol{\theta}; \mathcal{P})$ then $\boldsymbol{\psi}(\boldsymbol{\theta}; \mathcal{P}) = \nabla\rho(\boldsymbol{\theta}; \mathcal{P}) = \boldsymbol{g}$ and the system of equations

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

is equivalent to

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

- Also notice that the updating equation associated with Newton-Raphson is given by

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

which, in light of the previous point, can be rewritten as

$$\widehat{\boldsymbol{\theta}}_{i+1} = \widehat{\boldsymbol{\theta}}_i - H_i^{-1}\, \boldsymbol{g}_i$$

where

$$H_i = \begin{bmatrix} \frac{\partial^2\rho}{\partial\theta_1^2} & \frac{\partial^2\rho}{\partial\theta_1\theta_2} & \cdots & \frac{\partial^2\rho}{\partial\theta_1\theta_k} \\ \frac{\partial^2\rho}{\partial\theta_2\theta_1} & \frac{\partial^2\rho}{\partial\theta_2^2} & \cdots & \frac{\partial^2\rho}{\partial\theta_2\theta_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2\rho}{\partial\theta_k\theta_1} & \frac{\partial^2\rho}{\partial\theta_k\theta_2} & \cdots & \frac{\partial^2\rho}{\partial\theta_k^2} \end{bmatrix}$$

is the *Hessian* of ρ (i.e., the matrix of second partial derivatives of ρ with respect to $\theta_1, \theta_2, \ldots, \theta_k$)

- Recall that the updating equation associated with gradient descent is given by

$$\begin{aligned} \widehat{\boldsymbol{\theta}}_{i+1} &= \widehat{\boldsymbol{\theta}}_i - \widehat{\lambda}_i\, \boldsymbol{d}_i \\ &= \widehat{\boldsymbol{\theta}}_i - \frac{\widehat{\lambda}_i}{||\boldsymbol{g}_i||}\boldsymbol{g}_i \end{aligned}$$

- Thus Newton-Raphson applied to the gradient of the objective function is essentially equivalent to gradient descent applied directly to the objective function but with step sizes modulated by the Hessian of the objective function

2.3.4 Iteratively Reweighted Least Squares

When $\boldsymbol{\theta} = (\alpha, \beta)^\mathsf{T}$ is a vector valued attribute associated with a linear regression, **iteratively reweighted least squares (IRLS)** provides a conventient iterative method for finding $\widehat{\boldsymbol{\theta}} = (\widehat{\alpha}, \widehat{\beta})^\mathsf{T}$, the solution to

$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}\in\boldsymbol{\Theta}}{\operatorname{argmin}} \sum_{u\in\mathcal{P}} \rho(y_u - \alpha - \beta(x_u - c))$$

Differentiating this with respect to α and β and setting the result equal to zero yields

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

and $\widehat{\boldsymbol{\theta}} = (\widehat{\alpha}, \widehat{\beta})^{\mathsf{T}}$ is found by solving this system of equations.

If we let $r_u = y_u - \alpha - \beta(x_u - c)$ and $\boldsymbol{z}_u = (1,\ x_u - c)^{\mathsf{T}}$, then the system above becomes

$$\sum_{u \in \mathcal{P}} \rho'(r_u)\ \boldsymbol{z}_u = \mathbf{0}$$

- Note that in WLS $\rho(r_u) = w_u r_u^2$ and $\rho'(r) = 2 w_u r_u$ and sp the system reduces to

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

simplifies to this in WLS

- With the IRLS algorithm we will exploit the fact that the general objective function minimization problem can recast as a weighted least squares problem
- The general equation can be made to look like the WLS equation as follows:

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

where $w_u = \frac{\rho'(r_u)}{r_u}$ is the weight for unit u provided $r_u \neq 0$

  - Why did we do this?

Because WLS has a closed form solution.

- If we had some initial value for the residuals r_u (and hence the weights w_u) we could solve this equation in closed form yielding a value for $\boldsymbol{\theta}$, call it $\widehat{\boldsymbol{\theta}}_1 = (\widehat{\alpha}_1, \widehat{\beta}_1)^{\mathsf{T}}$
  - This requires initial values for α and β
  - Since $\boldsymbol{\theta}_0 = (\alpha_0, \beta_0)$ is likely far from $\widehat{\boldsymbol{\theta}} = (\widehat{\alpha}, \widehat{\beta})^{\mathsf{T}}$ we should iterate and update our values of the residuals (and hence weights) with $\widehat{\boldsymbol{\theta}}_1 = (\widehat{\alpha}_1, \widehat{\beta}_1)^{\mathsf{T}}$
  - This process can be repeated until the estimates of α and β converge
  - This process is called **iteratively reweighted least squares**
- We can generalize this algorithm by expressing the problem in terms of the vector-valued attribute $\boldsymbol{\theta} = (\alpha,\ \beta)^{\mathsf{T}}$: $r_u = y_u - \boldsymbol{z}_u{}^{\mathsf{T}} \boldsymbol{\theta}$. $= y_u - \alpha - \beta(x_u - c)$

The Algorithm

Given an initial value $\widehat{\boldsymbol{\theta}}_0$

1. **Initialize** ; $i \leftarrow 0$;
2. LOOP:
   a. **Construct residuals and weights** for all $u \in \mathcal{P}$

$$r_u = y_u - \boldsymbol{z}_u{}^{\mathsf{T}}\boldsymbol{\theta}_i \qquad \text{and} \qquad w_u = \frac{\rho'(r_u)}{r_u}$$

   b. **Solve** the weighted least squares problem, i.e., find $\widehat{\boldsymbol{\theta}}$, the value of $\boldsymbol{\theta}$ such that:

$$\sum_{u \in \mathcal{P}} w_u (y_u - \boldsymbol{z}_u{}^{\mathsf{T}}\boldsymbol{\theta})\ \boldsymbol{z}_u = \mathbf{0}$$

   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 works for any vector-valued attribute $\boldsymbol{\theta}$ that arises in the context of a **linear response model**:

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

R code for IRLS

```
irls <- function(y, x, theta, rhoPrimeFn,
                 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, rhoPrimeFn, 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)
}
```

Example 1 (Least Squares)

- Let us perform **least squares** regression to fit the following simple linear regression model

$$y_u = \alpha + \beta(x_u - \overline{x}_w) + r_u$$

  but by using IRLS as the method of estimation.

- In order to use the `irls` function we must first write the functions `getWeights(...)` and `getTheta(...)`, which for any problem are:

```
getWeights <- function(resids, rhoPrimeFn, delta = 1e-12) {
  ## for calculating weights, minimum |residual| will be delta
  smallResids <- abs(resids) <= delta
  ## take care to preserve sign (in case rhoPrime not symmetric)
  resids[smallResids] <- delta * sign(resids[smallResids])
  ## calculate and return weights
  rhoPrimeFn(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 must also define `getResids(...)` and a `rhoPrime` function, which for ordinary least squares regression are both based on

$$r_u = y_u - \alpha - \beta(x - \overline{x}_w), \qquad \rho(r_u) = r_u^2, \qquad \rho'(r_u) = 2r_u$$

```
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)
}

LSrhoPrime <- function(resid) {2*resid}
```

Let's now calculate $\widehat{\boldsymbol{\theta}} = (\widehat{\alpha}, \widehat{\beta})^{\mathsf{T}}$ for the `Where's Waldo` data using IRLS:

```
IRLSresult <- irls(waldo$Y, waldo$X, theta = c(0,0), rhoPrimeFn = LSrhoPrime)

print(IRLSresult)
```

```
## $theta
## [1] 3.8753064 0.1429041
##
## $converged
## [1] TRUE
##
## $iteration
## [1] 2
```

Let's also compare this result to that which is obtained by using `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 2 (Robust Regression)

- Recall the Huber Function was defined as

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

  and implemented in `R` with the function `huber.fn`

- Recall also that the Huber Function's derivative is given by

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

  and implemented in `R` with the function `huber.fn.prime`:

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

- Let's perform **robust regression** on the `Animals` data but by using IRLS as our method of estimation

```
result <- irls(log(Animals2$brain), log(Animals2$body),
               theta = c(1,1),
               rhoPrimeFn = huber.fn.prime, maxIterations = 100)
print(result)
```

```
## $theta
## [1] 3.1282452 0.6885898
##
## $converged
## [1] TRUE
##
```

```
## $iteration
## [1] 8
```

- Compare this result to that which is obtained by `rlm` function from the `MASS` package in `R`.

```
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
```