Response models

R.W. Oldford

Winter 2017

Contents

1 Response models **1**
1.1 Linear models . . . 2
1.2 A generative probability model . . . 19
1.3 Aside: Interactive plots with loon . . . 25
1.4 Weighted least squares . . . 29

2 Whither the weights? **30**
2.1 Differing variances . . . 30
2.2 Zero one weights . . . 31
2.3 Robust estimation . . . 38
2.4 Power transformations . . . 70
2.5 Selecting a subset . . . 76

---

1 Response models

Suppose we have a population $\mathcal{P}$ having individuals $i \in \{1, \ldots, N\}$ and variates x_i, y_i measured on individual i.
For many such populations, it is often of interest to distinguish these variates between those which are **response variates** and those which are **explanatory variates**. Traditional notation uses y to denote response variates and x to denote explanatory.

As the name "response" suggests, sometimes the "explanatory" variate attains its value before the response does and we might be interested in any effect the value of the explanatory might have on the value subsequently attained by the response variate. For example, in the case of treating a patient for hypertension we might be interested in how the dosage of a particular drug might affect the subsequent measurement of the patient's blood pressure.

However, such direct causal relations need not necessarily exist for us to adopt the same language. We might rather be interested how the values taken by the response might be associated with, or explained by, the values of any explanatory variates.

To emphasize this relationship between response and explanatory variates, we might imagine relating the response to the explanatory variate as follows:

$$y_i = \mu(x_i) + r_i \qquad i = 1, \ldots, N$$

where $\mu(\cdot)$ is some function of its argument (to be determined) and r_i is now introduced to simply be the difference between the value of the response and that of the function evaluated at the corresponding explanatory variate's value. We call $r_i = y_i - \mu(x_i)$ the i'th **residual**.

Choosing $\mu(\cdot)$ will be a particular challenge. We want it to be parsimonious, in the sense that it is no more complex than it need be. This will make it easier to understand. But also we want it to close enough to y that it seems to capture its behaviour adequately. This might mean introducing fairly complicated forms for $\mu(\cdot)$. Response modelling is about balancing these two desirable characteristics.

Oftentimes we will specify only the functional form of $\mu(\cdot)$ and then use the values of the response and explanatory variates to estimate the details.

1.1 Linear models

Some common choices for the functional form of $\mu(\cdot)$ include, for example, the following

$$
\begin{array}{lrcl}
(1) & \mu(x) & = & \alpha + \beta x \\
(2) & \mu(x) & = & \alpha + \beta x + \gamma z \\
(3) & \mu(x) & = & \alpha + \beta_1 x + \beta_2 x^2 + \ldots + \beta_p x^p \\
(4) & \mu(x) & = & \alpha + \beta_1 x + \beta_2 \sin(x) + \beta_3 \log(x)
\end{array}
$$

where the Greek letters represent **parameters** whose values are unknown.

These are all examples of **linear models**, so called because the function $\mu(\cdot)$ is *linear in its parameters*. This means we can represent $\mu(\cdot)$ in vector form as

$$\mu(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\beta}$$

where now $\mathbf{x}$ is a vector of, say, p entries all of which are known values and $\boldsymbol{\beta}$ is a parameter vector of p unknown values to be determined from the data. Rewriting the above examples we have

$$
\begin{array}{ll}
(1) & \mathbf{x}^T = (1, x) \;\text{ and }\; \boldsymbol{\beta}^T = (\alpha, \beta) \\
(2) & \mathbf{x}^T = (1, x, z) \;\text{ and }\; \boldsymbol{\beta}^T = (\alpha, \beta, \gamma) \\
(3) & \mathbf{x}^T = (1, x, x^2, \ldots, x^p) \;\text{ and }\; \boldsymbol{\beta}^T = (\alpha, \beta_1, \beta_2, \ldots \beta_p) \\
(4) & \mathbf{x}^T = (1, x, \sin(x), \log(x)) \;\text{ and }\; \boldsymbol{\beta}^T = (\alpha, \beta_1, \beta_2, \beta_3)
\end{array}
$$

When we have N vectors of values $\mathbf{x_1}, \ldots, \mathbf{x_N} \in \mathbb{R}^p$, say, then we have N equations in p unknowns ($\boldsymbol{\beta} \in \mathbb{R}^p$) which we can write all together as

$$\boldsymbol{\mu} = \mathbf{X}\boldsymbol{\beta}$$

where $\boldsymbol{\mu} = (\mu(\mathbf{x}_1), \ldots, \mu(\mathbf{x}_N))^T$ is an $N \times 1$ vector and $\mathbf{X} = [\mathbf{x_1}, \ldots, \mathbf{x_N}]^T$ is an $N \times p$ matrix of known values.

We now use the observed values of the explanatory variates involved in determining $\mathbf{X}$ and the response variate values $\mathbf{y} = (y_1, \ldots, y_N)^T$ to **estimate** the unknown parameters $\boldsymbol{\beta}$. Our model connecting the two is written in matrix and vector form as

$$\mathbf{y} = \boldsymbol{\mu} + \mathbf{r}$$

or

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \mathbf{r}$$

1.1.1 Least squares estimation

One of the earliest and most commonly used methods of estimation (dating back to Legendre and Gauss) is that of **least-squares**. Here we choose $\widehat{\beta}$ to be that value which minimizes the sum of squared residuals:

$$\sum_{i=1}^{N} r_i^2 = \sum_{i=1}^{N} (y_i - \mathbf{x}_i^{\mathsf{T}} \boldsymbol{\beta})^2$$

$$or$$

$$\mathbf{r}^{\mathsf{T}}\mathbf{r} = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^{\mathsf{T}} (\mathbf{y} - \mathbf{X}\boldsymbol{\beta}).$$

The value which minimizes this sum (assuming $\mathbf{X}$ is of full column rank p) is

$$\widehat{\beta} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{y}$$

and

$$\begin{aligned} \widehat{\boldsymbol{\mu}} &= \mathbf{X}\widehat{\boldsymbol{\beta}} \\ &= \mathbf{X}\left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{y} \\ &= \mathbf{P}\mathbf{y}, \quad \text{say.} \end{aligned}$$

The matrix $\mathbf{P}$ is a **projection matrix** since it is *idempotent* (i.e. $\mathbf{P} \cdot \mathbf{P} = \mathbf{P}$) and because it is also *symmetric* (i.e. $\mathbf{P}^{\mathsf{T}} = \mathbf{P}$) it is also an **orthogonal projection matrix**.

The N-dimensional geometry of least-squares estimation is shown in the following picture:

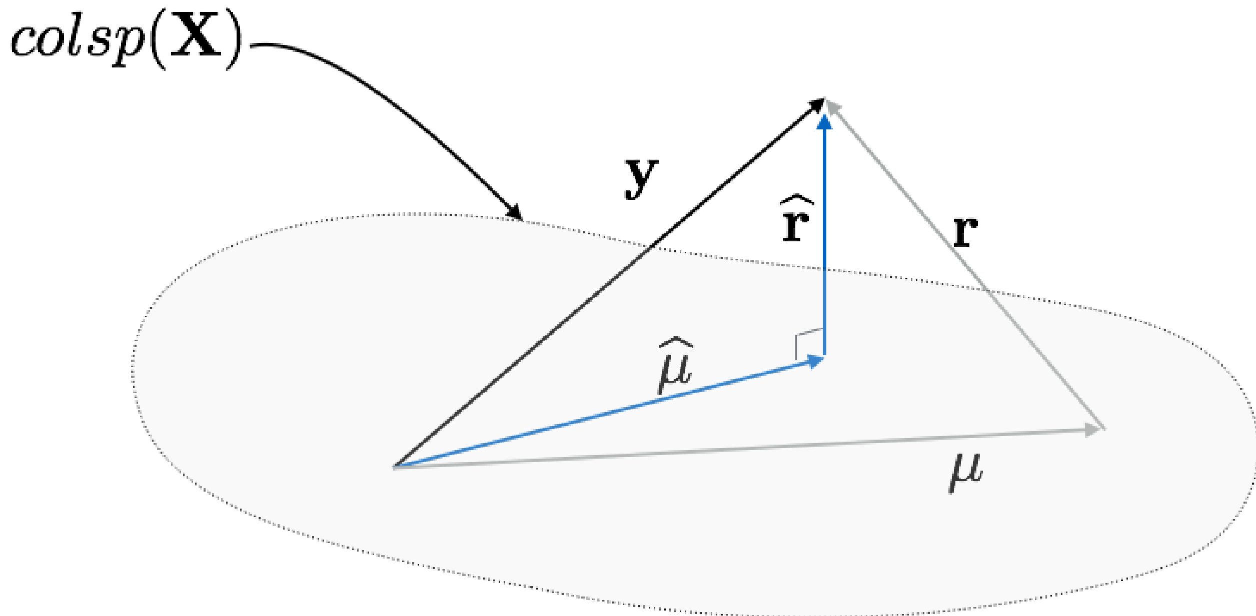


Figure 1: The N-dimensional geometry of least-squares estimation

The pieces are

- $\mathbf{y} = \boldsymbol{\mu} + \mathbf{r} \in \mathbb{R}^N$
- $\boldsymbol{\mu} = \mathbf{X}\boldsymbol{\beta}$ and hence $\boldsymbol{\mu} \in colsp(\mathbf{X})$, namely the "column-space" of $\mathbf{X}$ (i.e. the *span* of the column vectors of $\mathbf{X}$)
- $\widehat{\mathbf{r}} = (\mathbf{I}_N - \mathbf{P})\mathbf{y}$ is chosen to to be the vector having shortest Euclidean length (least-squares) which connects to the closest vector $\widehat{\mu} = \mathbf{P}\mathbf{y} \in colsp(\mathbf{X})$

- being the shortest possible vector implies $\widehat{\mathbf{r}}$ is perpendicular to $colsp(\mathbf{X})$ (which is easily checked)

Note that the **dimension** of the $colsp(\mathbf{X})$ is equal to the rank of the matrix $\mathbf{X}$, typically p (otherwise $(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}$ is not defined). Similarly, since $\widehat{\mathbf{r}}$ is perpendicular to the $colsp(\mathbf{X})$, it necessarily lies in the **orthogonal complementary subspace** of $colsp(\mathbf{X})$ in $\mathbb{R}^N$. This orthogonal complementary subspace is necessarily of dimension $N - p$.

Note that the dimensions of these two subspaces, $colsp(\mathbf{X})$ and its orthogonal complement, are exactly the **degrees of freedom** associated with the model and residual respectively.

When the $N \times p$ matrix $\mathbf{X}$ is of full column rank p, its column vectors form a **basis** for $colsp(\mathbf{X})$. The columns of the projection matrix $\mathbf{P}$ also form a set of **generators** for $colsp(\mathbf{X})$ but **not** a basis (since the columns of $\mathbf{P}$ are not linearly independent). Similarly, the columns of $\mathbf{I}_N - \mathbf{P}$ generate the residual space but do not form a basis (again, these are not linearly independent e.g. $(\mathbf{I}_N - \mathbf{P})\mathbf{X} = \mathbf{0}$).

1.1.2 Illustration on the Facebook data

We can now illustrate this modelling in `R` using the `facebook` data and the built in "linear model" function `lm(...)`.

```
dataDir <- "/Users/rwoldford/Documents/Admin/courses/Stat444/Data/UCIrvineMLRep/FacebookMetrics"
completePathname <- paste(dataDir, "facebook.csv", sep="/")
facebook <- read.csv(completePathname, header=TRUE)
```

Recall that several of the variates in `facebook` have missing values, recorded as `NA`. For illustration, let's remove all those cases which have missing data (**for any variate**).

```
fb <- na.omit(facebook)
```

This reduces our sample to only 495 cases instead of 500.

```
# Let's add a couple of variates to the facebook data
fb$x <- log(fb$Impressions)
fb$y <- log(fb$like + 1)
# We fit a line as
facebook.fit1 <- lm(y ~ x, data=fb)
# The estimates of the coefficients are
facebook.fit1$coefficients
```

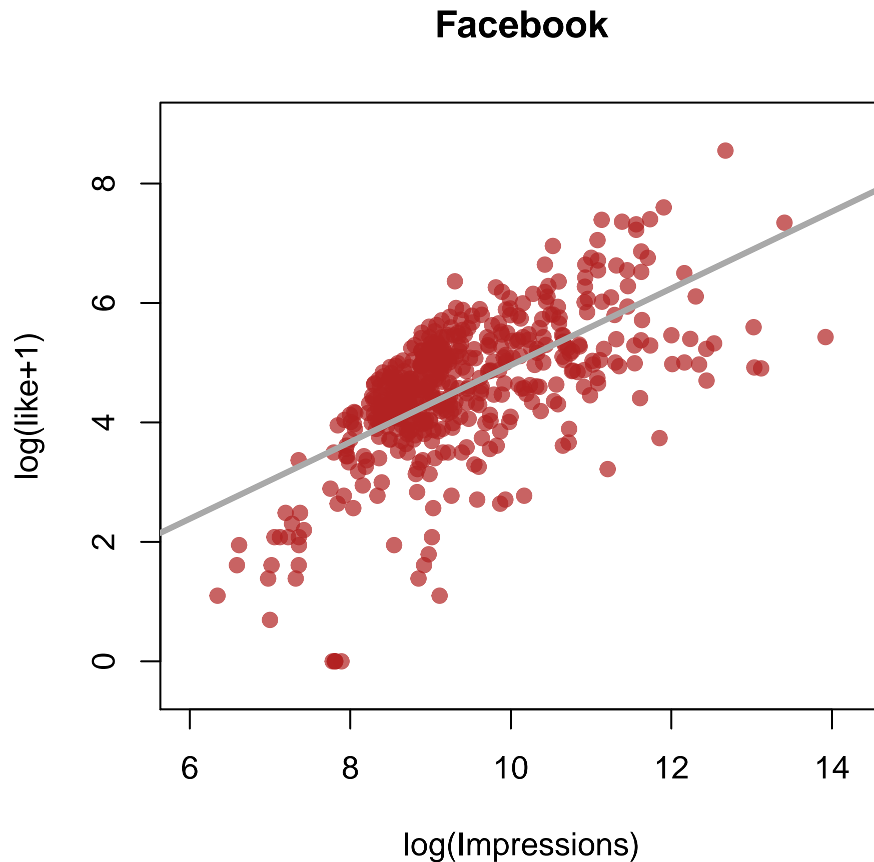
```
## (Intercept)           x
##  -1.4721861   0.6430359
```

```
# or equivalently
coef(facebook.fit1)
```

```
## (Intercept)           x
##  -1.4721861   0.6430359
```

```
# Which we can plot the fit on the data as follows.
# First get ranges; these will be handy when we want to ensure plots
# are shown on the same range as determinined by the facebook data.
xlim <- extendrange(fb$x)
ylim <- extendrange(fb$y)
plot(fb$x, fb$y,
     xlim = xlim, ylim = ylim,
     main = "Facebook",
     xlab = "log(Impressions)",
     ylab = "log(like+1)",
     pch=19,
```

```
     col=adjustcolor("firebrick", 0.7)
     )
# add the fitted line
abline(facebook.fit1, col= "darkgrey", lwd=3)
```



Note how the model was specified: `Y ~ X`. This means of command lime specification is called **Wilkinson-Rogers notation** which first appeared in the statistical computing system called `GENSTAT` and was popularized by the interactive statistical system called `glim` (for generalized linear models). It has been adopted by many statistical systems since and was incorporated into the `S` language as a `formula` specification. `R` is an open source implementation of the `S` language and follows this tradition for specifying the structure of a linear model. In `R`, many other semantics for model building have since been added to this notation.

In brief, the notation generally has the following semantics in characterising the structure of a linear model:

- all terms appearing in the formula must reference components in the data frame in the fitting call. For example, the data frame `facebook` has named variates `like`, `Impressions`, `share`, etc.. Any of these may appear in the formula.
  - the response appears to the left of the tilde `~`, the explanatory terms in the model to the right.
  - the intercept is specified as `1` but is assumed to be present by default. For example, the model

    $$y = \alpha + \beta x + r$$

    can be specified using the data frame `facebook` as either `Y ~ 1 + X` where the intecept term is explicitly shown, or as `Y ~ X` where the intecept is implicit. Because intercepts are nearly always present, they must be explicitly removed in the formula when not part of the model. For example, the model

    $$y = \beta x + r$$

will have formula `Y ~ X - 1` and not `Y ~ X`.

- factors (defined in `R` and having some finite number of levels) may contribute many parameters and terms to the model (typically one fewer than the number of levels).
- (almost) any function of the variates may appear in the formula. For example `Y ~ X + sin(X)` will work fine, as will `Y = X + sin(X * Z)`. The former specifies the model $y = \alpha + \beta x + \gamma \sin(x) + r$, the latter $y = \alpha + \beta x + \gamma \sin(xz) + r$. Where trouble occurs is in trying to use operators such as `+`, `-`, `*`, and `^`. These will first be interpreted as formual operators and not as arithmetic operators.

For example, `Y ~ X + (X * Z)` does not model $y = \alpha + \beta x + \gamma xz + r$ but rather $y = \alpha + \beta x + \gamma z + \delta xz + r$.

This is because `X * Z` has special model meaning – it means include both `X` and `Z` in the model as well as their "interaction" or product xz.

If the model $y = \alpha + \beta x + \gamma xz + r$ is what you want, you can do this in one of many ways.

These include: `Y ~ X + X:Z` where the last term indicates add only the interaction (':') and not the effect of `Z` on it own, by `Y ~ X * Z - Z` where the term `X * Z` means `X + Z + X:Z` from which the last term removes the stand alone `Z`, and perhaps most importantly one could use the "inhibit" function `I(...)` which prevent its arguments from being interpreted as formula objects.

To express the same model we could use `Y ~ X + I( X * Z)`; this makes the arithmetic calculation first and feeds the result as a new variate for the formula.

If you are ever wondering about whether you have the model you meant to have, you can always look at the matrix $\mathbf{X}$ in `R` as follows:

```
head(model.matrix(facebook.fit1))
```

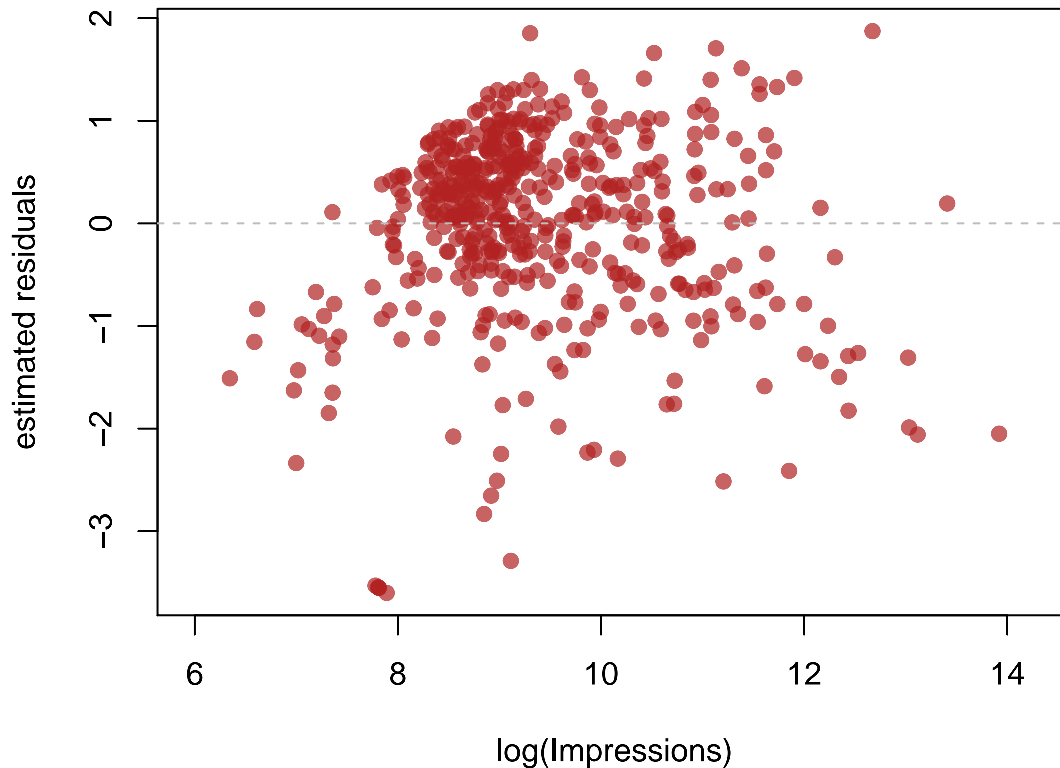
```
##   (Intercept)         x
## 1           1  8.535230
## 2           1  9.855190
## 3           1  8.383205
## 4           1 11.384990
## 5           1  9.517384
## 6           1  9.945061
```

OK, back to our producing a left to right strategy for finding facebook by a left to right single search. How well is our line doing? To assess this we might have a look at the **fitted residuals**:

$$\begin{aligned} \widehat{r}_i &= y_i - \widehat{\mu}(x_i) \\ &= y_i - \widehat{\alpha} - \widehat{\beta} x_i \end{aligned}$$

for $i = 1, 2, \ldots, N$. We could plot the pairs $(x_i, \widehat{r}_i)$ ad see what structure, if any, remains.

```
# Recall that there were some NA values in this data.

plot(fb$x, residuals(facebook.fit1),
     xlim = xlim,
     main = "Residual plot",
     xlab = "log(Impressions)",
     ylab = "estimated residuals",
     pch=19,
     col=adjustcolor("firebrick", 0.7)
     )
abline(h=0, col="grey", lty=2)
```

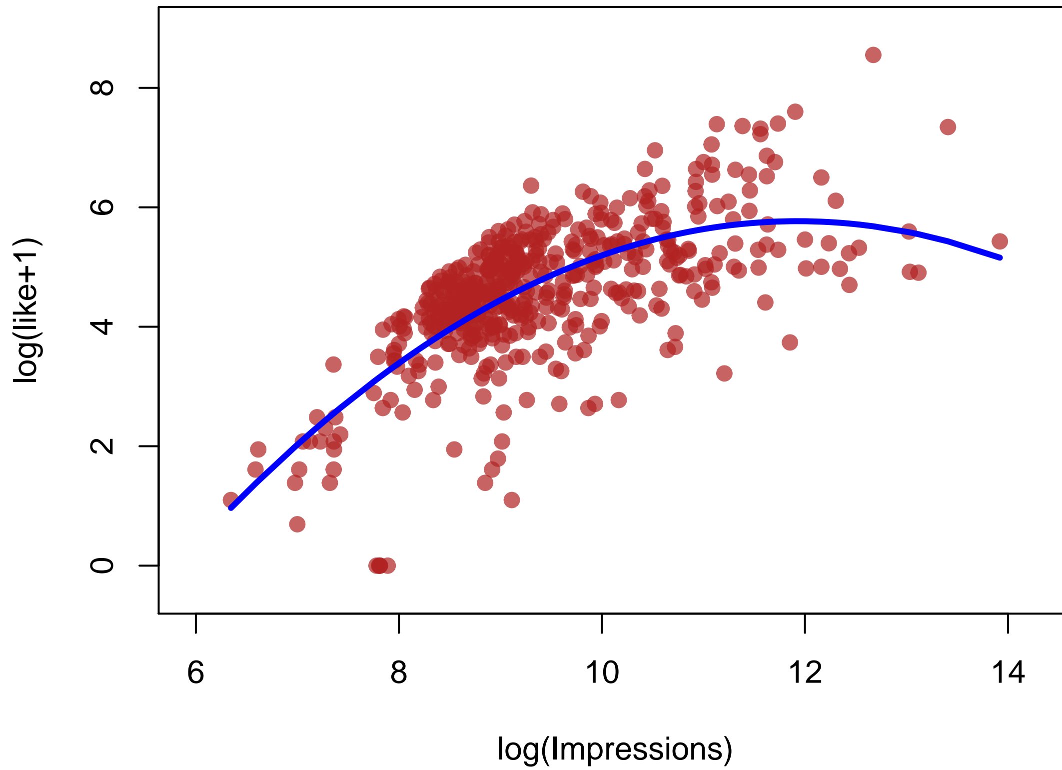


How about, instead of a line, we try a quadratic function?

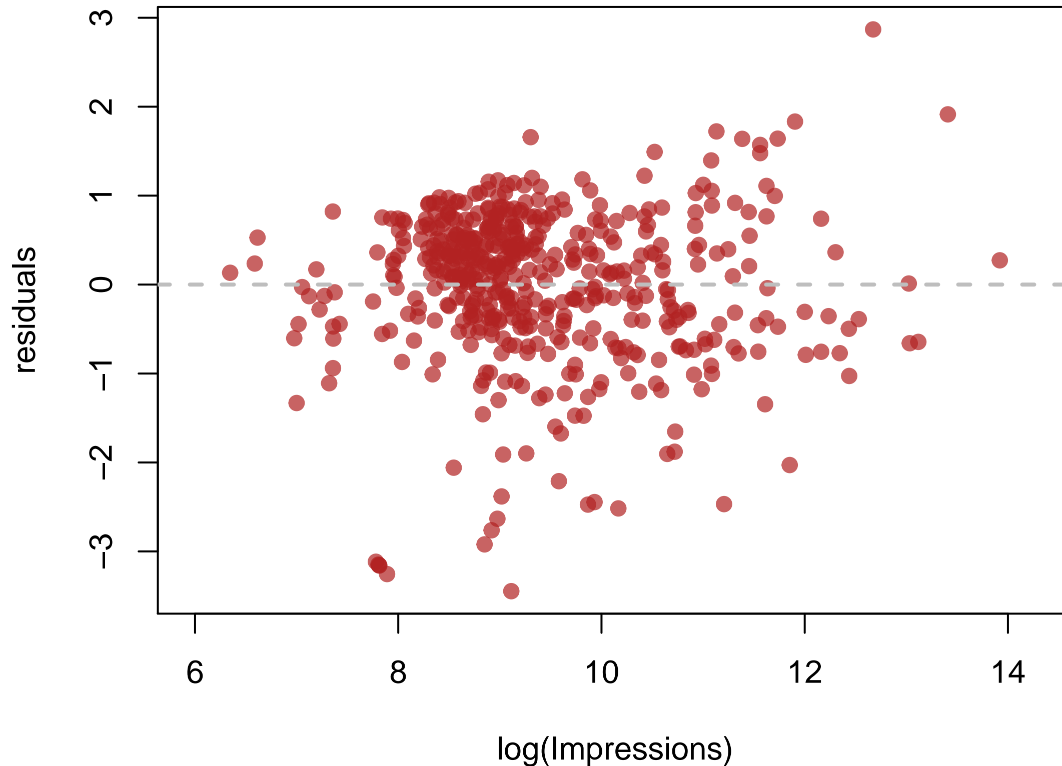
```
#
# How about a quadratic?
#
facebook.fit2 <- lm(y ~ x+ I(x^2), data=fb)
coef(facebook.fit2)
```

```
## (Intercept)           x      I(x^2)
## -16.1531581   3.6753933  -0.1540597
```

```
plot(fb$x, fb$y,
     xlim = xlim, ylim = ylim,
     main = "Facebook",
     xlab = "log(Impressions)",
     ylab = "log(like+1)",
     pch=19,
     col=adjustcolor("firebrick", 0.7)
     )
# or the fitted (predicted) values for y at each point
# (N.B in order of X!)
Xorder <- order(fb$x)
lines(fb$x[Xorder],
      predict(facebook.fit2)[Xorder],
      col="blue", lwd=3)
```




```
plot(fb$x, residuals(facebook.fit2),
     xlim = xlim,
     main = "Facebook: fit from a quadratic",
     xlab = "log(Impressions)",
     ylab = "residuals",
     pch=19,
     col=adjustcolor("firebrick", 0.7)
     )
abline(h=0, col="grey", lty=2, lwd=2)
```

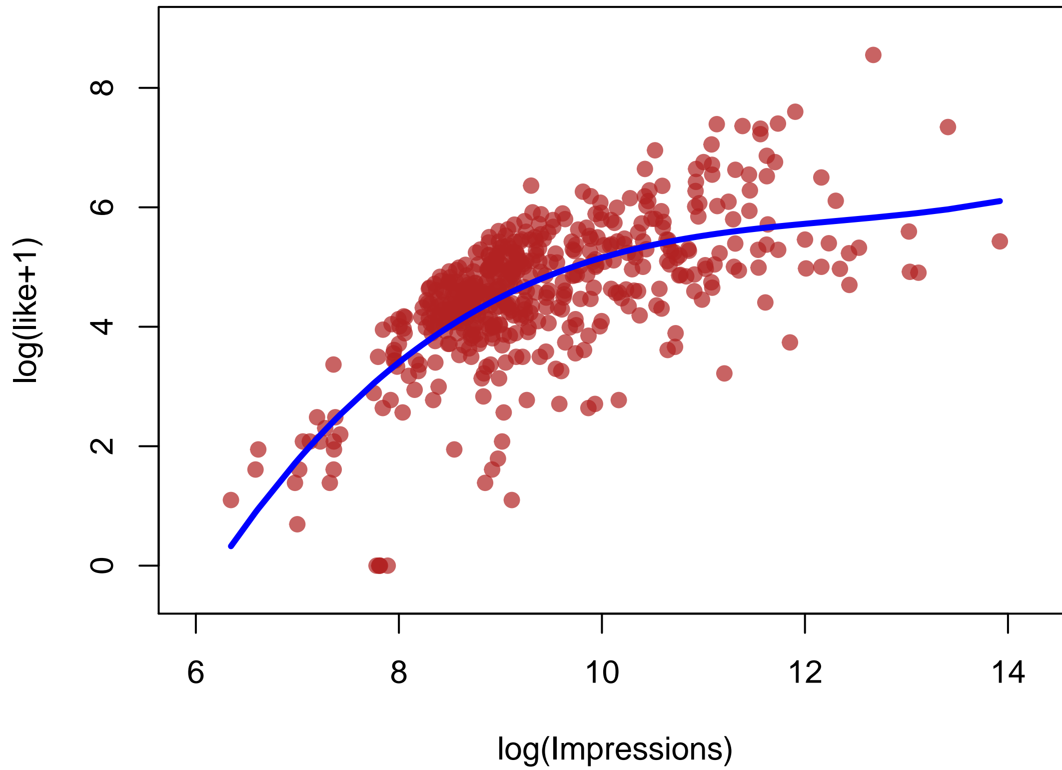


Or maybe a cubic?

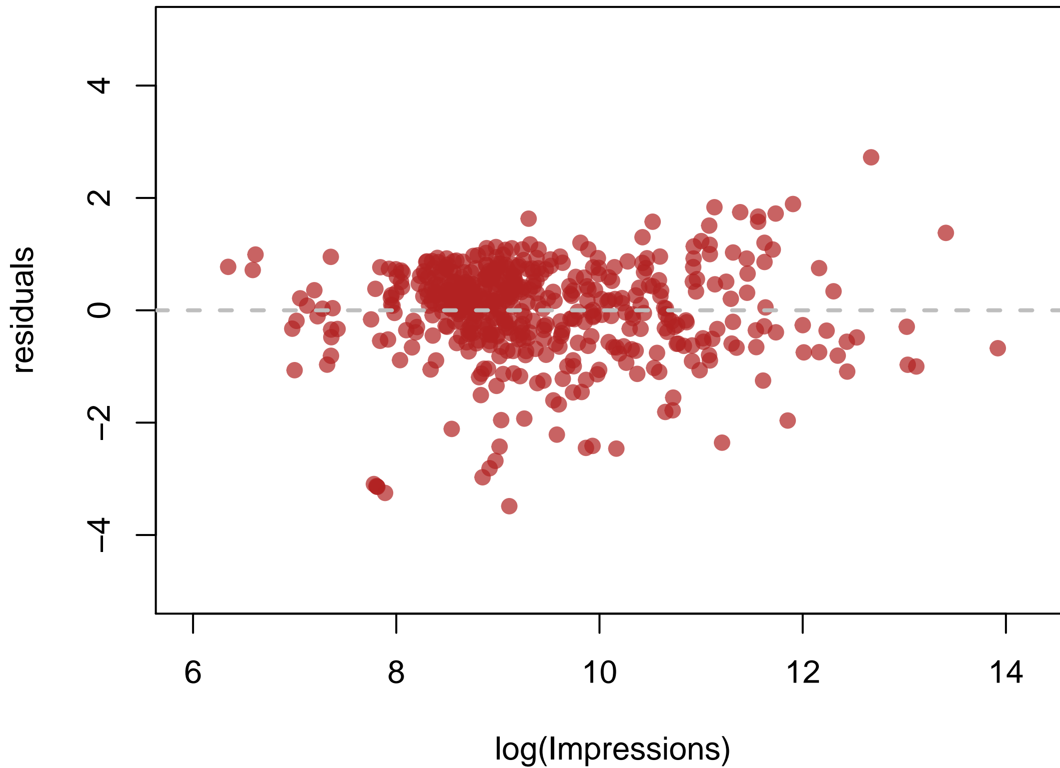
```
#
# How about a cubic?
#
facebook.fit3 <- lm(y ~ x+ I(x^2)+ I(x^3), data=fb)
coef(facebook.fit3)
```

```
##   (Intercept)             x        I(x^2)        I(x^3)
## -35.97427196   9.86644308  -0.78885821   0.02135218
```

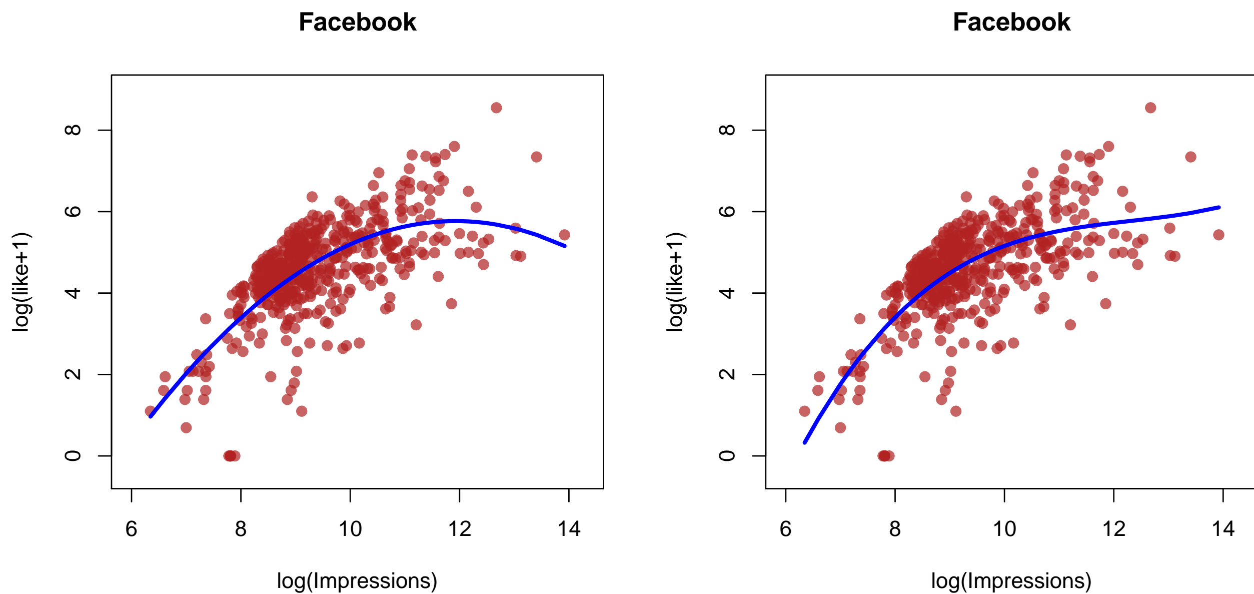
```
plot(fb$x, fb$y,
     xlim = xlim, ylim = ylim,
     main = "Facebook",
     xlab = "log(Impressions)",
     ylab = "log(like+1)",
     pch=19,
     col=adjustcolor("firebrick", 0.7)
     )
# or the fitted (predicted) values for y at each point (N.B in order of X!)

lines(fb$x[Xorder],
      predict(facebook.fit3)[Xorder],
      col="blue", lwd=3)
```




```r
# Look at residuals
plot(fb$x, residuals(facebook.fit3),
     xlim = xlim, ylim=c(-5,5),
     main = "Facebook: fit from a cubic",
     xlab = "log(Impressions)",
     ylab = "residuals",
     pch=19,
     col=adjustcolor("firebrick", 0.7)
     )
abline(h=0, col="grey", lty=2, lwd=2)
```

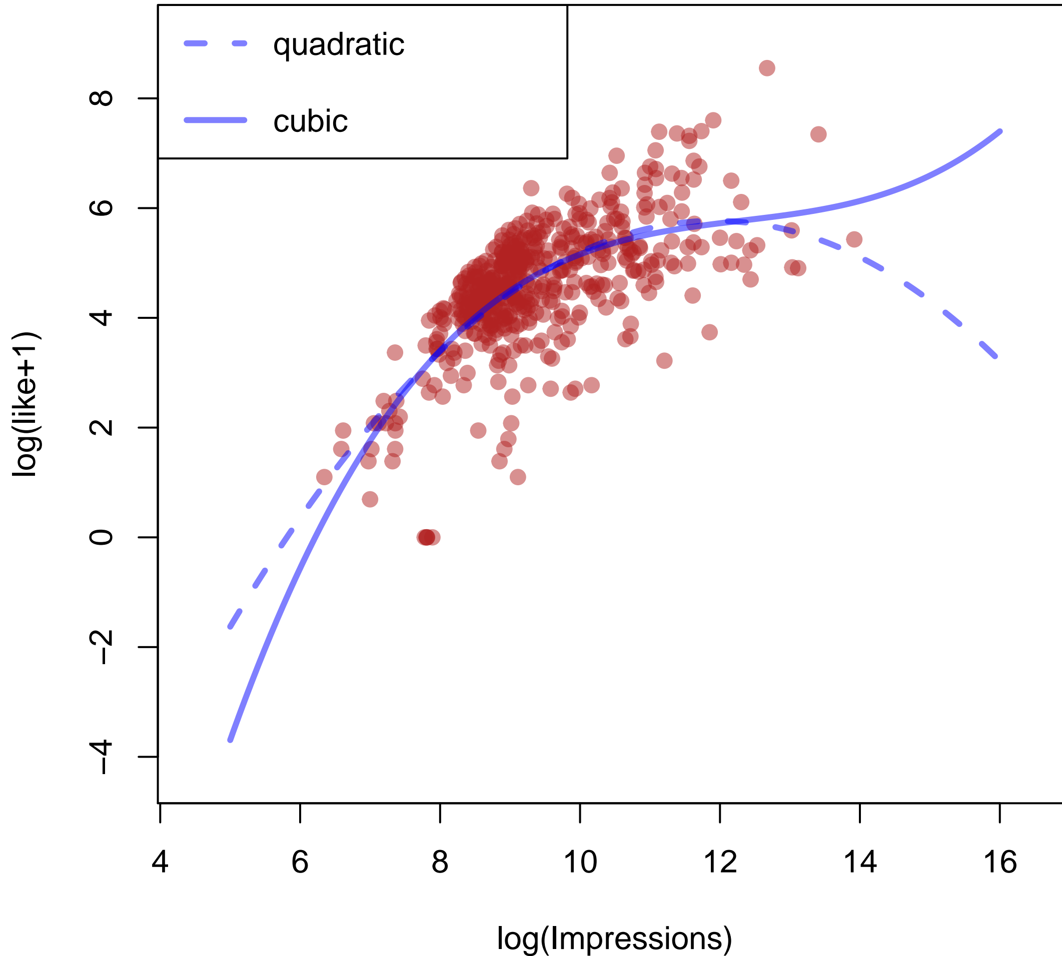


What difference do you observe between the two curves?



Note that both models are **global**, in the sense that the function $\widehat{\mu}(x)$ is defined once for all x. This means, for example, that either model could be used to predict beyond the range of the observed x values. This is problematic since our study population is restricted in the number of `Impressions` that have been recorded. If we plot the curves for a larger set of x values, we can see substantial differences between them:

```r
# Create the new data
newX <- data.frame(x=seq(5,16, length.out=1000))
newY2 <- predict(facebook.fit2, newdata = newX)
newY3 <- predict(facebook.fit3, newdata = newX)
newXlim <- extendrange(newX)
newYlim <- extendrange(c(newY2, newY3, fb$y))
plot(fb$x, fb$y,
     xlim = newXlim, ylim = newYlim,
     main = "Facebook",
     xlab = "log(Impressions)",
     ylab = "log(like+1)",
     pch=19,
     col=adjustcolor("firebrick", 0.5)
     )

lines(newX$x, newY2,
      col=adjustcolor("blue", 0.5),
      lty=2, lwd=3)

lines(newX$x, newY3,
      col=adjustcolor("blue", 0.5),
      lty=1, lwd=3)

# Add a legend
legend("topleft",
       legend = c("quadratic", NA, "cubic"),
       col = c(adjustcolor("blue", 0.5),NA,adjustcolor("blue", 0.5)),
       lty=c(2,NA,1), lwd=c(3,NA, 3),
       text.width = 4
       )
```



If we are predicting outside the range of the x the quadratic model does not agree with experience. Having the number of `likes` go to zero (here `log(like +1)` going to $-\infty$) as the number of `Impressions` (here `log(Impressions)`) increases seems quite wrong. The cubic at least continues to increase. Note, that it **is** wrong is an argument outside of the data. THe argument is **extra-statistical** in that it is based on our understanding of the target population - it just does not make sense for `likes` to go to zero as `Impressions` increases. On the left, however, both models behave more as expected, each having the `likes` heading towards zero as the number of `Impressions` do.

Remember the dictum attributed to George Box: "All models are wrong, but some are useful."

1.1.3 The model geometry, revisited

Another way of thinking about the linear model is that it asserts that $\mu(x)$ can be expressed as some (unknown) *linear combination of functions*.

For example, letting $g_1(x) = 1$, $g_2(x) = x$, $g_3(x) = \sin(x)$, and $g_4(x) = \log(x)$ as in model (4) above, we are specifying that the function $\mu(x)$ is some unknown linear combination of these four functions, say

$$\mu(x) = \theta_1 g_1(x) + \theta_2 g_2(x) + \theta_3 g_3(x) + \theta_4 g_4(x)$$

the coefficients $\theta_i \in \mathbb{R}$ being unknown constants.

As with real-valued vectors, we can think of all linear combinations of these four specific functions as forming a *subspace*, this time a subspace of all real-valued functions. The functions $g_1, \ldots, g_4$ are *generators* of that subspace and provided they are also *linearly independent* of one another, then these functions also form a set of *basis* functions for that subspace. The linear model asserts that $\mu(x)$ lies in that subspace, a subspace of *dimension* equal to the number of basis functions which define it.

This is, perhaps, an unfamiliar way of looking at the linear model. Recall for example the infinite series expansion of e^x (say for $x \in \mathbb{R}$):

$$e^x = 1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \frac{1}{4!}x^4 + \cdots$$

This essentially says that the exponential function is a known (infinite) linear combination of all the (infinitely many) simple polynomials in x. These polynomials form an **infinite-dimensional basis** for the space of real-valued functions of a real-valued variable.

Recall that Taylor's theorem says, that (for x near 0, anyway) that using only the first few terms in this series might still give a reasonable approximation of e^x. We can think of this as finding a function $\mu(x)$ which is a linear combination of, say, only the simple polynomials up to degree 4:

$$\mu(x) = \theta_0 x^0 + \theta_1 x + \cdots + \theta_4 x^4.$$

That is $\mu(x)$ lies in a 5 dimensional subspace of the space of real functions (as defined by these 5 linearly independent functions). Taylor's theorem tells us what the coefficients θ_i for $i = 0, \ldots, 4$ but suppose we instead try to estimate these by least-squares.

Here's how that might go:

```
x <- seq(-2, 2, length.out = 1000)
y <- exp(x)
# We fit our "model" as before
#
fit <- lm(y~x + I(x^2) + I(x^3) + I(x^4))
#
# We can now compare coefficients to the Taylor series values
# First the fit
round(fit$coefficients,3)
```

```
## (Intercept)           x      I(x^2)      I(x^3)      I(x^4)
##       1.002       0.964       0.489       0.207       0.050
```

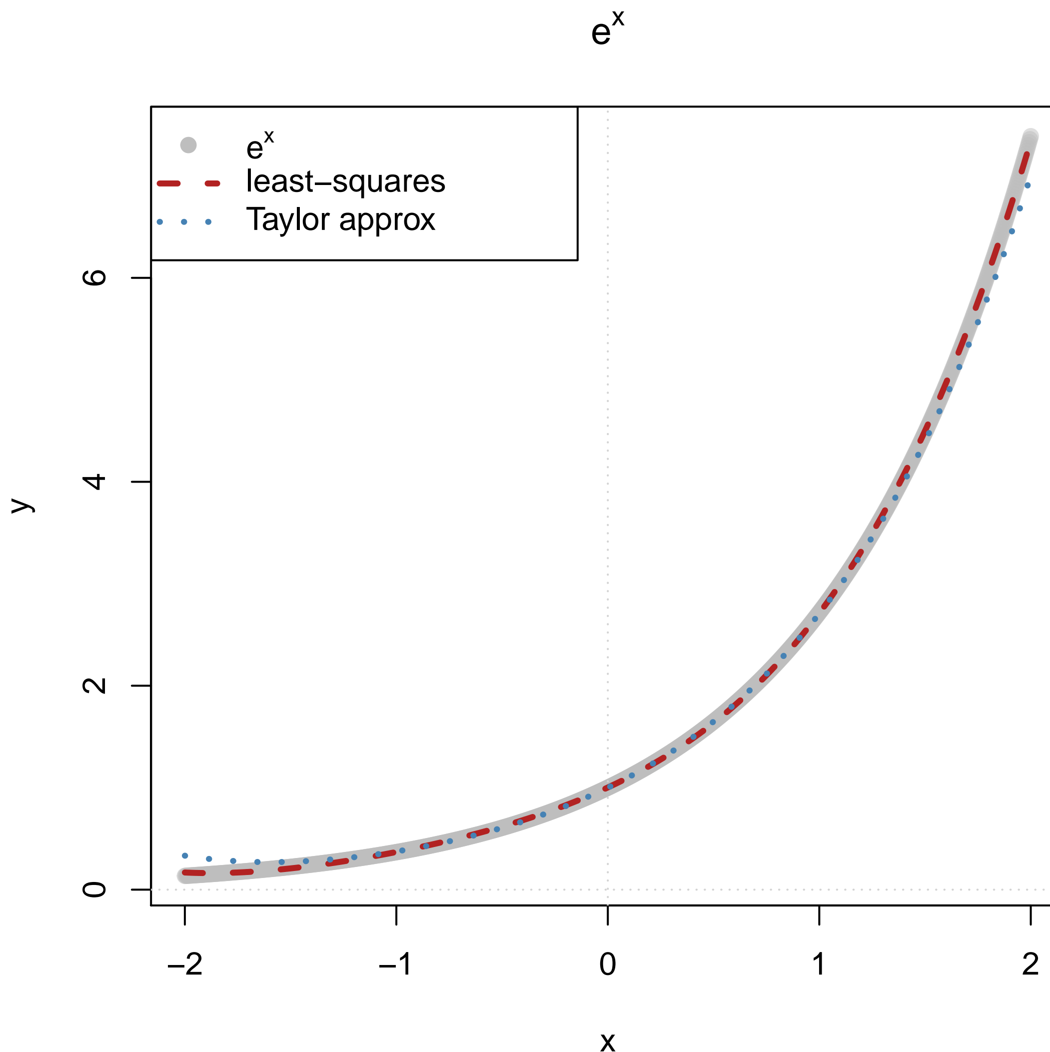
```
# and now the Taylor coefficients
round(sapply(0:4, function(x) 1/factorial(x)), 3)
```

```
## [1] 1.000 1.000 0.500 0.167 0.042
```

They are not exactly the same, but they are close. We might plot the fit and residuals as below.

```
plot(x, y, col=adjustcolor("grey", 0.5), pch=19,
     main = expression(e^x), xlab="x", ylab="y")
abline(h=0, col="lightgrey", lty=3)
abline(v=0, col="lightgrey", lty=3)

# Note that the x's are in order 1:N so order(x) was unnecessary here
lines(x, fit$fitted.values, col="firebrick", lty=2, lwd=3)

# The Taylor approx
ytaylor = 1 + x + x^2/2  + x^3/6 + x^4 /24
lines(x, ytaylor, col="steelblue", lty=3, lwd=3)
```

```
# Add a legend
legend("topleft",
       legend = c(expression(e^x), "least-squares", "Taylor approx"),
       col = c("grey", "firebrick", "steelblue"),
       lty=c(NA, 2,3 ), pch = c(19, NA, NA), lwd=c(NA, 3, 3),
       text.width = 1.5
       )
```

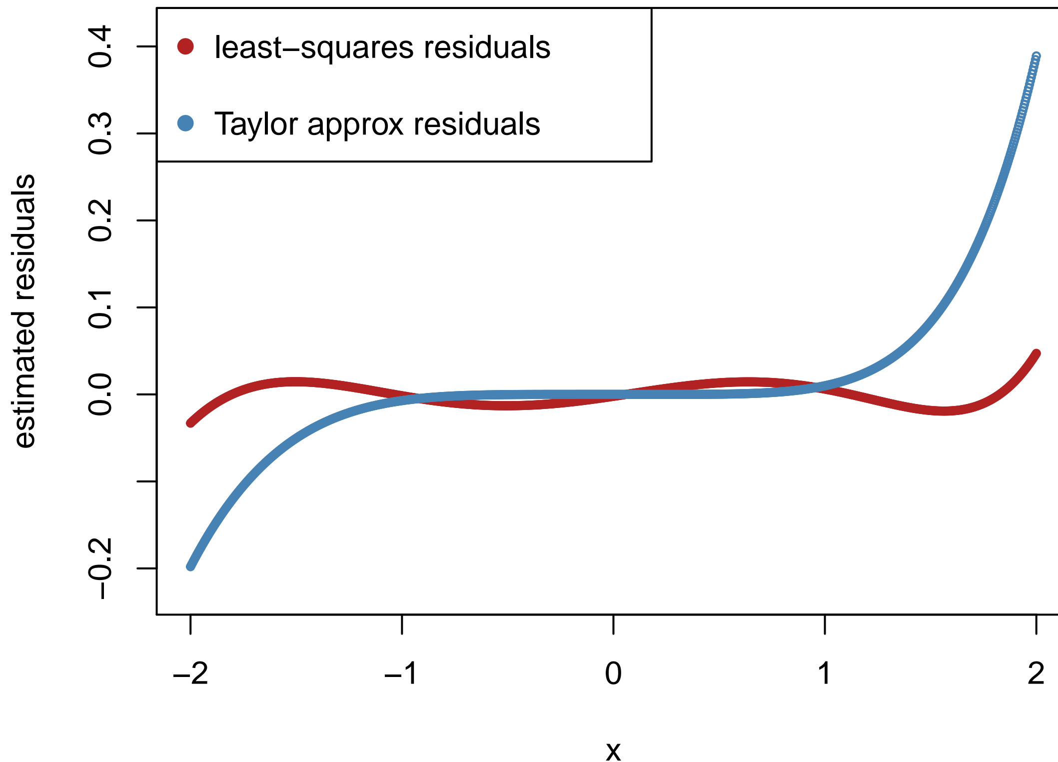


Note that the least-squares fitted function performs better than does the Taylor series approximation!

We can also have a look at what is "left over" from our fitted function. This "residual function"

```
taylor_resids <- y - ytaylor
residlim <- extendrange(c(taylor_resids, fit$residuals))

# The residuals from ls
plot(x, fit$residuals, cex=0.5, col="firebrick",  ylim=residlim,
     main = "The residual `function' for each",
     xlab="x", ylab="estimated residuals")
# Taylor residuals
points(x, taylor_resids, cex=0.5, col="steelblue")
```

```r
# Add a legend
legend("topleft",
       legend = c("least-squares residuals", "", "Taylor approx residuals"),
       col = c("firebrick", NA, "steelblue"),
       pch = c(19, NA, 19),
       text.width = 2
       )
```



Again, the least-squares estimation appears to have produced a better fit overall. To be fair, however, we know that the Taylor approximation holds for x near zero. In the range $(-1, 1)$ the Taylor approximation outperforms least-squares (based on $x \in [-2, 2]$).

Note that had we instead used `lm(y ~ poly(x,4))` to fit the data, the simple polynomials x^0, x, x^2, x^3, and x^4 would not have been used but rather some **orthogonal polynomials** which generated the same subspace. That is an **orthogonal basis** would have been used.

```r
fit_orthog <- lm(y ~ poly(x,4))
# Same mu-hat
round(sum(fit$fitted.values - fit_orthog$fitted.values), 14)
```

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

```r
# different coefficients
fit$coefficients
```

```
## (Intercept)           x      I(x^2)      I(x^3)      I(x^4)
##  1.00217165  0.96402413  0.48876261  0.20733773  0.04986943
```

```r
fit_orthog$coefficients
```

```
## (Intercept) poly(x, 4)1 poly(x, 4)2 poly(x, 4)3 poly(x, 4)4
##    1.815381   53.461139   24.943173    7.953894    1.930130
```

Note that the fitted $\widehat{\mu}(x)$ is the same but the coefficients are different because **two different sets of basis functions** were used.

1.1.4 Facebook example geometry

We can similarly break down the fitted model `facebook.fit3` which had a $\mu(x)$ which was cubic in x.

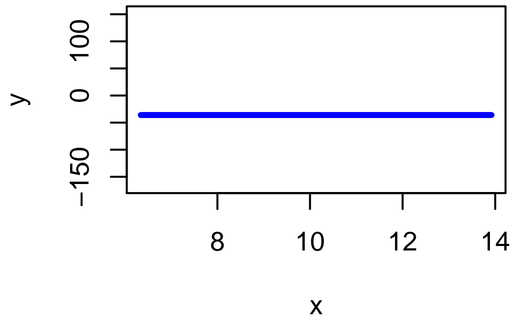
The coefficients of the model were

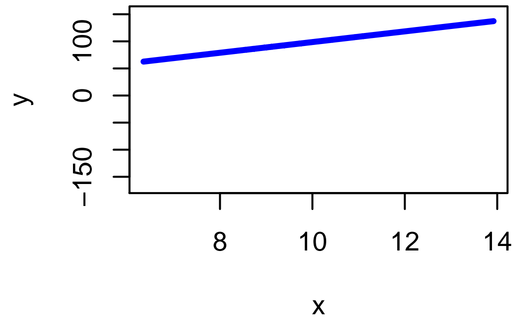
```
facebook.fit3$coefficients
```

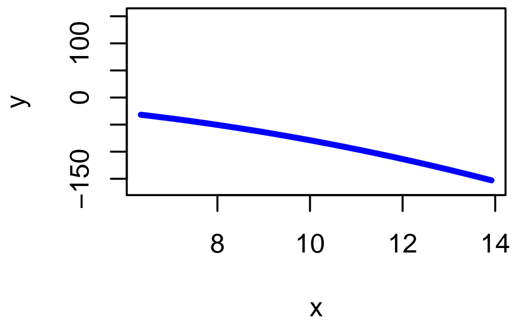
```
##  (Intercept)            x       I(x^2)       I(x^3)
## -35.97427196   9.86644308  -0.78885821   0.02135218
```

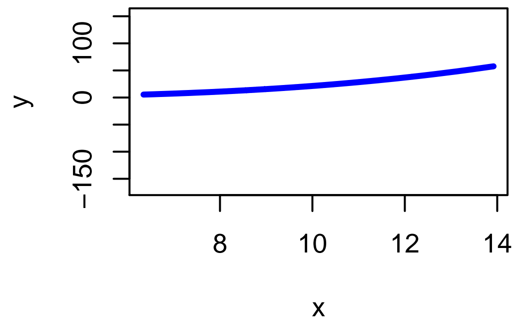
So the cubic fit can be decomposed as the sum of these four pieces

```
Xorder <- order(fb$x)
n <- length(Xorder)
g0 <- rep(facebook.fit3$coefficients[1], n)    # constant term  x^0
g1 <- fb$x * facebook.fit3$coefficients[2]     # linear term    x
g2 <- fb$x^2 * facebook.fit3$coefficients[3]   # squared term   x^2
g3 <- fb$x^3 * facebook.fit3$coefficients[4]   # cubed term     x^3
savePar <- par(mfrow=c(2,2))
glim <- extendrange(c(g0, g1, g2, g3))
plot(fb$x[Xorder], g0, col="blue", type="l",
     ylim=glim, lwd=3,
     main=paste0("g0 =",
                 round(facebook.fit3$coefficients[1],2)),
     xlab="x", ylab="y")
plot(fb$x[Xorder], g1[Xorder], col="blue", type="l",
     ylim=glim, lwd=3,
     main=paste0("g1 =",
                 round(facebook.fit3$coefficients[2],2),
                 "x"),
     xlab="x", ylab="y")
plot(fb$x[Xorder], g2[Xorder], col="blue", type="l",
     ylim=glim, lwd=3,
     main=paste0("g2 =",
                 round(facebook.fit3$coefficients[3],2),
                 "x^2"),
     xlab="x", ylab="y")
plot(fb$x[Xorder], g3[Xorder], col="blue", type="l",
     ylim=glim, lwd=3,
     main=paste0("g3 =",
                 round(facebook.fit3$coefficients[4],2),
                 "x^3"),
     xlab="x", ylab="y")
```





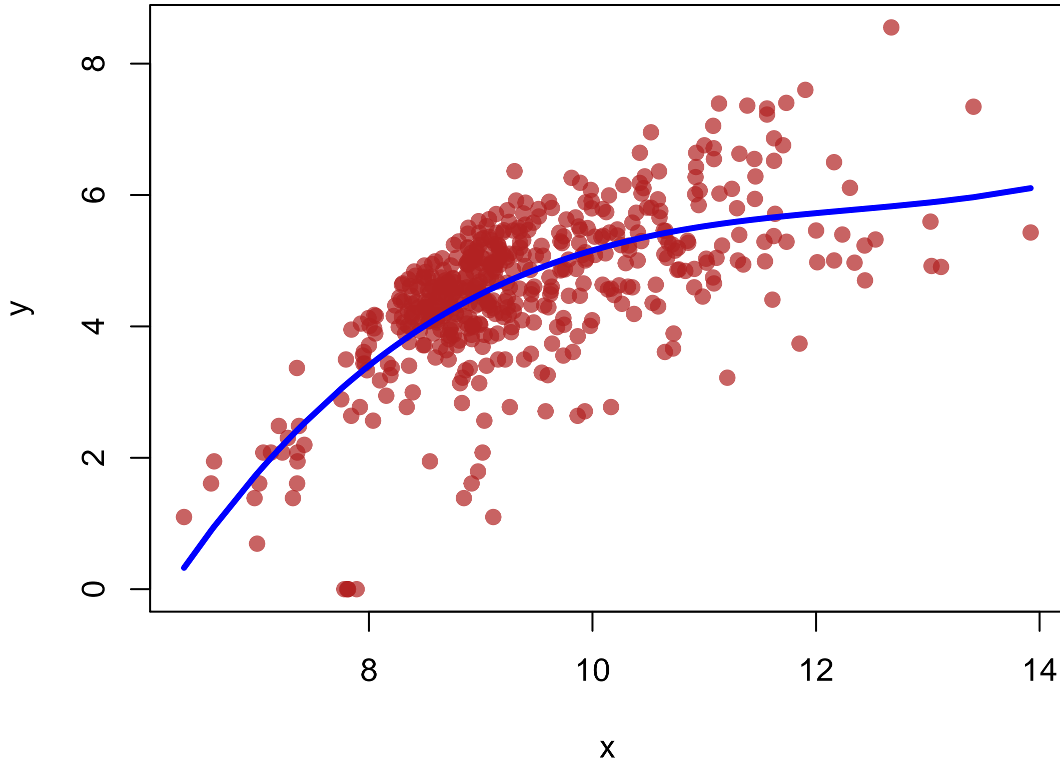





```
par(savePar)
```

And their sum is

```
plot(fb$x, fb$y,
     main = "mu = g0 + g1 + g2 + g3",
     xlab = "x",
     ylab = "y",
     pch=19,
     col=adjustcolor("firebrick", 0.7)
     )
lines(fb$x[Xorder], g0[Xorder] + g1[Xorder] + g2[Xorder] + g3[Xorder],
     col="blue", lwd=3)
```



1.2 A generative probability model

Sometimes, we might like to consider a possible generative model, one that holds the values of the explanatory variates fixed but which generates values of the response based on our model.

For example, suppose that we have p explanatory variates (including the intercept if it appears in the model), then our model on the data values we have in hand can be expressed as

$$\begin{aligned} y_i &= \mu(\mathbf{x}_i) + r_i \\ &= \mathbf{x}_i^T \boldsymbol{\beta} + r_i \end{aligned} \tag{1}$$

A model for the data which explains how the response might have been *generated* from the values of the explanatory variates (fixed here) might be:

$$\begin{aligned} Y_i &= \mu(\mathbf{x}_i) + R_i \\ &= \mathbf{x}_i^T \boldsymbol{\beta} + R_i \end{aligned} \tag{2}$$

for $i = 1, \ldots, n$.

Here, the upper case notation indicates that Y_i is a random variate representing the set of possible values which the i'th response might take. The lower case y_i is a number, a realized value from the random distribution of $Y - i$. So too for R_i and r_i.

Note that $n, \mathbf{x_i}$ and $\boldsymbol{\beta}$ are being modelled as fixed (known and unknown) quantities and not as random variates in this model.

Further model assumptions often include:

- $E(R_i) = 0$ for all $i = 1, \ldots, n$ – *common and zero residual mean*
- $SD(R_i) = \sigma$ for all $i = 1, \ldots, n$ – *common constant residual standard deviation*
- $R_1, R_2, \ldots, R_n$ are *independently* distributed
- $R_1, R_2, \ldots, R_n$ are *identically* distributed
- R_i is *normally* distributed (or, equivalently, distributed as Gaussian)

Putting this all together and the **normal or Gaussian linear model** can be written as

$$\begin{aligned} Y_i &= \mathbf{x}_i^T \boldsymbol{\beta} + R_i \qquad (3) \\ \text{and} \qquad & R_i \sim N(0, \sigma^2) \;\; \text{independently} \end{aligned}$$

for $i = 1, \ldots, n$.

Alternatively, an equivalent expression of this generative model, one which simply emphasizes the mean structure, is

$$\begin{aligned} Y_i &= \mu_i + R_i \qquad (4) \\ \mu_i &= \mathbf{x}_i^T \boldsymbol{\beta} \\ \text{and} \qquad & R_i \sim N(0, \sigma^2) \;\; \text{independently} \end{aligned}$$

for $i = 1, \ldots, n$.

One thing this model might be used for is to get some idea of how reliable the our fitted curve might be. We can imagine, for example, generating new responses for our explanatory variates and fitting the model again. Each new sample generated will have its own estimate of $\mu(\mathbf{x})$ for any $\mathbf{x}$.

1.2.1 Confidence intervals and prediction intervals

Given the above assumptions, if $\widehat{\mu}(\mathbf{x})$ is an estimate the distribution of the estimator $\widetilde{\mu}(\mathbf{x})$) is easily shown to be

$$\widetilde{\mu}(\mathbf{x}) \sim N\left(\mu(\mathbf{x}), \;\; \sigma^2 \mathbf{x}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}\right)$$

for any vector valued $\mathbf{x}$. This, together with the fact that

$$(N-p)\frac{\widetilde{\sigma}^2}{\sigma^2} \sim \chi^2_{(N-p)}$$

independently of $\widetilde{\mu}(\mathbf{x})$, means that we have a basis for all kinds of inferential tools such as **confidence intervals**, **hypothesis tests**, and **prediction intervals**.

For example, from the above two distibutional results (and independence of the estimators) we have

$$\frac{\widetilde{\mu}(\mathbf{x}) - \mu(\mathbf{x})}{\sigma\sqrt{\left(\mathbf{x}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}\right)}} \sim N(0,1),$$

and dividing this by $\widetilde{\sigma}/\sigma$ gives

$$\frac{\widetilde{\mu}(\mathbf{x}) - \mu(\mathbf{x})}{\sigma\sqrt{\left(\mathbf{x}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}\right)}} \times \frac{\sigma}{\widetilde{\sigma}} = \frac{\widetilde{\mu}(\mathbf{x}) - \mu(\mathbf{x})}{\widetilde{\sigma}\sqrt{\left(\mathbf{x}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}\right)}} \sim \frac{N(0,1)}{\sqrt{\chi^2_{N-p}/(N-p)}} = t_{N-p}.$$

We can now use the Student t distribution to determine a positive constant a, for any probability α say to be such

$$\begin{aligned} \alpha &= Pr(-a \;\leq\; t_{N-p} \;\leq\; a) \\ &= Pr\left(-a \;\leq\; \frac{\widetilde{\mu}(\mathbf{x}) - \mu(\mathbf{x})}{\widetilde{\sigma}\sqrt{\left(\mathbf{x}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}\right)}} \leq \; a\right) \end{aligned}$$

To find a for α we have

$$Pr(t_{N-p} \ \le \ a) = \alpha + \frac{1-\alpha}{2} = \frac{1}{2}(1+\alpha)$$

or

$$a = Q_{t_{N-p}}\left(\frac{1+\alpha}{2}\right)$$

where $Q_{t_{N-p}}(\cdot)$ is the **quantile function** for the t distribution on $N-p$ degrees of freedom.

The value of a is had in `R` using this relationship. For example, suppose $N = 68$ and $p = 4$ (as it would for our fitted cubic model on the facebook data), and that $\alpha = 0.95$, then in `R` we calculate `a <- qt(0.975, 64)` to yield a value of `a` as approximately 2.

With the value of a in hand we turn the above probability statement about a random variate (the function of the estimators) into a probability statement about a random interval as follows:

$$\begin{aligned} \alpha &= Pr\left(-a \ \le \ \frac{\widetilde{\mu}(\mathbf{x}) - \mu(\mathbf{x})}{\widetilde{\sigma}\sqrt{\mathbf{x}^T(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{x}}} \le \ a\right) \\ &= Pr\left(\ \mathbf{I}(\mathbf{x}) \ni \mu(\mathbf{x}) \ \right) \end{aligned}$$

where $\mathbf{I}(\mathbf{x})$ is the **random interval**

$$\left[\ \widetilde{\mu}(\mathbf{x}) - a \ \times \ \widetilde{\sigma}\sqrt{\mathbf{x}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}} \ \ , \ \ \widetilde{\mu}(\mathbf{x}) + a \ \times \ \widetilde{\sigma}\sqrt{\mathbf{x}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}} \ \right].$$

That is, this is a statement about the probability that a **random interval contains the true unknown constant** $\mu(\mathbf{x})$ for that $\mathbf{x}$. The good news is that we have **one realization of this random interval** namely:

$$\left[\ \widehat{\mu}(\mathbf{x}) - a \ \times \ \widehat{\sigma}\sqrt{\mathbf{x}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}} \ \ , \ \ \widehat{\mu}(\mathbf{x}) + a \ \times \ \widehat{\sigma}\sqrt{\mathbf{x}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}} \ \right]$$

where every value above will be known – that is this will be some numerical interval.

We call this particular realized interval a **confidence interval** and associate a **level of confidence** with it of $100\alpha\%$, α being the probability associated with the mechanism creating the intervals.

To consider prediction of an individual Facebook `like`, we can proceed in the same fashion. The only difference is that we recognize that an individual new observation would be a realization of Y_{new} say and that

$$Y_{new} = \mu(\mathbf{x}_{new}) + R_{new} \sim N(\mu(\mathbf{x}_{new}) \ , \ \sigma^2).$$

This is distributed independently of $\widetilde{\mu}(\mathbf{x}_{new})$ which is based on the independent variates $R_1, \ldots, R_N$. We can therefore write down the distribution of the difference $Y_{new} - \widetilde{\mu}(\mathbf{x}_{new})$ as

$$Y_{new} - \widetilde{\mu}(\mathbf{x}_{new}) \sim N\left(0 \ , \ \sigma^2\left(1 + \mathbf{x}_{new}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}_{new}\right)\right).$$

This gives us something to work with. As we did for $\mu(\mathbf{x})$, for a given probability α we find a from a Student t_{N-p} distibution and derive

$$\begin{aligned} \alpha &= Pr\left(-a \ \le \ \frac{Y_{new} - \widetilde{\mu}(\mathbf{x}_{new})}{\widetilde{\sigma}\sqrt{1 \ + \ \mathbf{x}_{new}^T(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{x}_{new}}} \le \ a\right) \\ &= Pr\left(\ \mathbf{I}_{new}(\mathbf{x}_{new}) \ni Y_{new} \ \right) \end{aligned}$$

where $\mathbf{I}_{new}(\mathbf{x}_{new})$ is the **random interval**

$$\left[\ \widetilde{\mu}(\mathbf{x}_{new}) - a \ \times \ \widetilde{\sigma}\sqrt{1 \ + \ \mathbf{x}_{new}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}_{new}} \ \ , \ \ \widetilde{\mu}(\mathbf{x}_{new}) + a \ \times \ \widetilde{\sigma}\sqrt{1 \ + \ \mathbf{x}_{new}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}_{new}} \ \right].$$

This probability statement is quite different than that we had when building confidence intervals. This statement is about the **probability that a random interval contains a random variate**. It should come as no surprise that such an interval for the random Y_{new} would be wider than that for the fixed value $\mu(\mathbf{x}_{new})$.

The one such interval we have is again had by replacing estimat**ors** by the corresponding estimat**es**

$$\left[\widehat{\mu}(\mathbf{x}_{new}) - a \times \widehat{\sigma}\sqrt{1 + \mathbf{x}_{new}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}_{new}}\ ,\ \widehat{\mu}(\mathbf{x}_{new}) + a \times \widehat{\sigma}\sqrt{1 + \mathbf{x}_{new}^T \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{x}_{new}}\ \right]$$

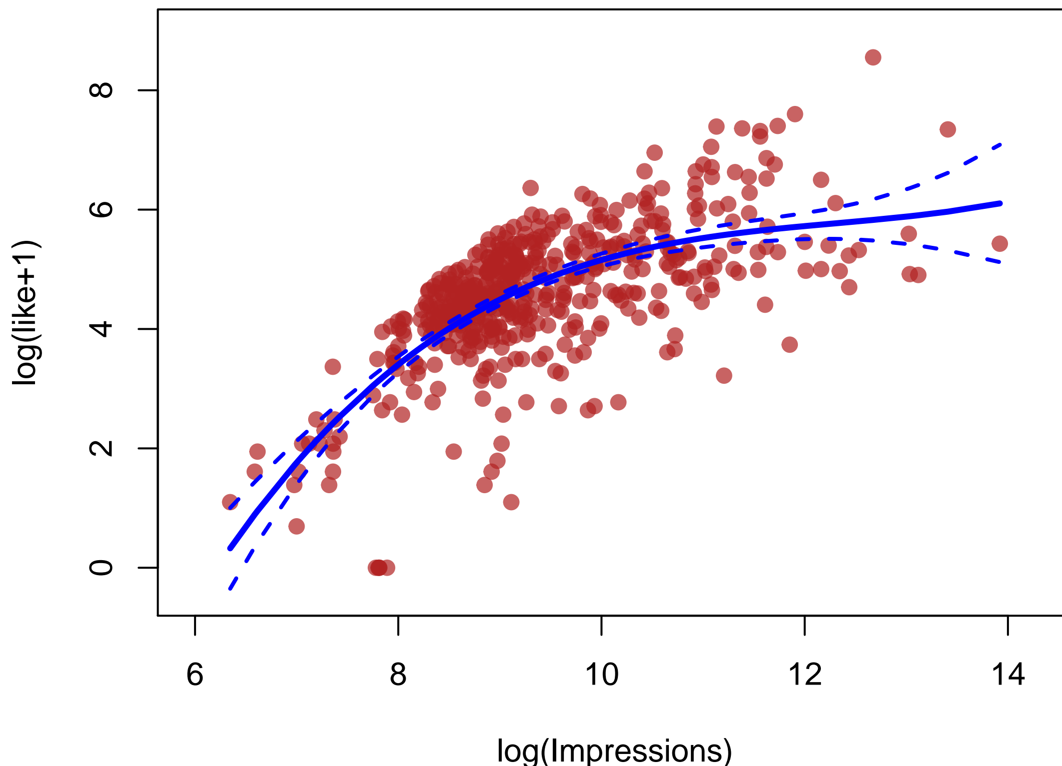
which, to distinguish it from a confidence interval, we will call a $100\alpha\%$ **prediction** interval for Y_{new}.

These intervals (confidence and prediction) are made easily available in `R` using the `predict(...)` function. Here we will begin with getting 95% confidence intervals for $\mu(\mathbf{x})$ at the observed values of $\mathbf{x}$

```
conf95.fit3 <- predict(facebook.fit3, interval="confidence", level=0.95)
head(conf95.fit3)
```

```
##        fit      lwr      upr
## 1 4.046280 3.943749 4.148811
## 2 5.081672 4.980982 5.182361
## 3 3.878429 3.770007 3.986852
## 4 5.614266 5.440067 5.788465
## 5 4.880715 4.788021 4.973408
## 6 5.128876 5.024989 5.232762
```

As you can see, in addition to the fitted values $\widehat{\mu}(\mathbf{x}_i)$, the upper and lower values of the confidence interval are also there and these can be plotted.



These are confidence intervals for $\mu(\mathbf{x})$ or, if our generative model is to be believed, for the expected

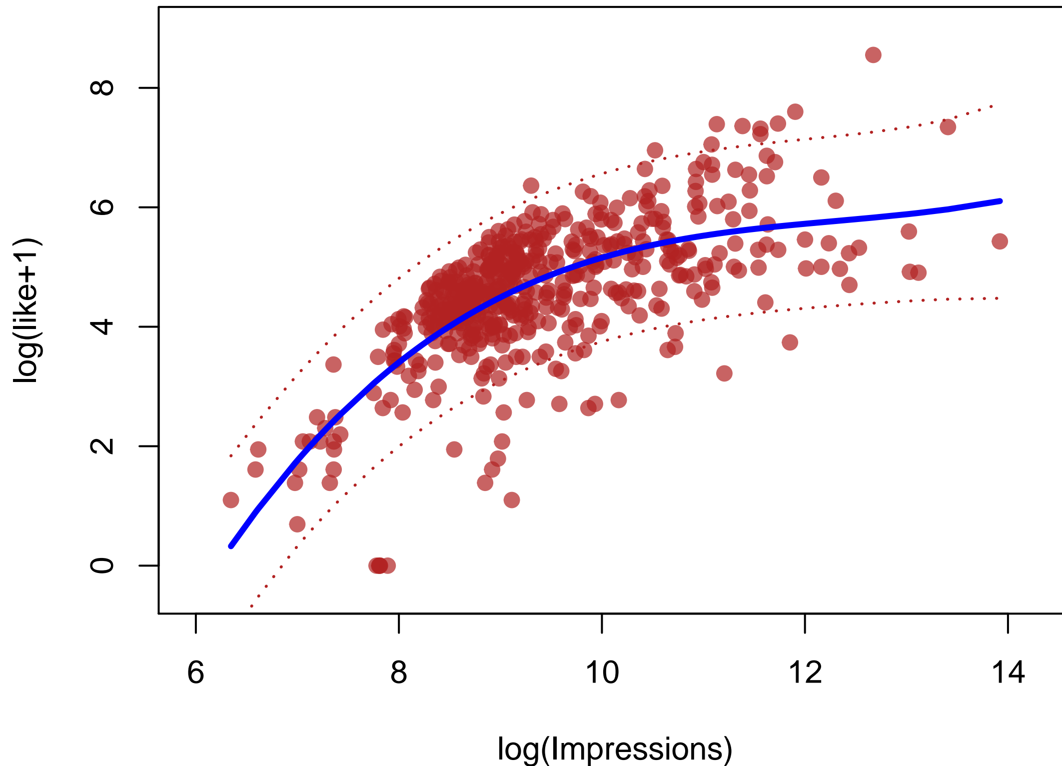
`log(like+1)` given the `log(Impressions)`. These connected confidence intervals represent a guess (with $100\alpha\% = 95\%$ "confidence") and where we think the generative mean function lies.

But here we are really interested in finding a rule for predicting the number of facebook likes given the number of impressions of a post. That suggests that we should really be looking for a prediction interval for the number of facebook likes (or `log(like+1)` as opposed to its average). The higher the level of the **coverage probability** α the more certain we are that our intervals will contain the actual number of facebook likes (or `log(like+1)`).
Suppose you would like to be about 90% certain about your prediction, say?

```
plot(fb$x, fb$y,
     xlim = xlim, ylim = ylim,
     main = "Facebook: 90% prediction intervals",
     xlab = "log(Impressions)",
     ylab = "log(like+1)",
     pch=19,
     col=adjustcolor("firebrick", 0.7)
     )

# or the fitted (predicted) values for y
# at each point (N.B in order of X!)

lines(fb$x[Xorder],
      predict(facebook.fit3)[Xorder],
      col="blue", lwd=3)

#  If you want to be really certain you find
#  facebook, say 90% of the time
#
pred90.fit3 <- predict(facebook.fit3,
                       interval="prediction",
                       level=0.90)
```

```
## Warning in predict.lm(facebook.fit3, interval = "prediction", level = 0.9): predictions on current data r
```

```
lines(fb$x[Xorder],
      pred90.fit3[,'lwr'][Xorder],
      col="firebrick", lwd=1.5,
      lty=3)

lines(fb$x[Xorder],
      pred90.fit3[,'upr'][Xorder],
      col="firebrick", lwd=1.5,
      lty=3)
```



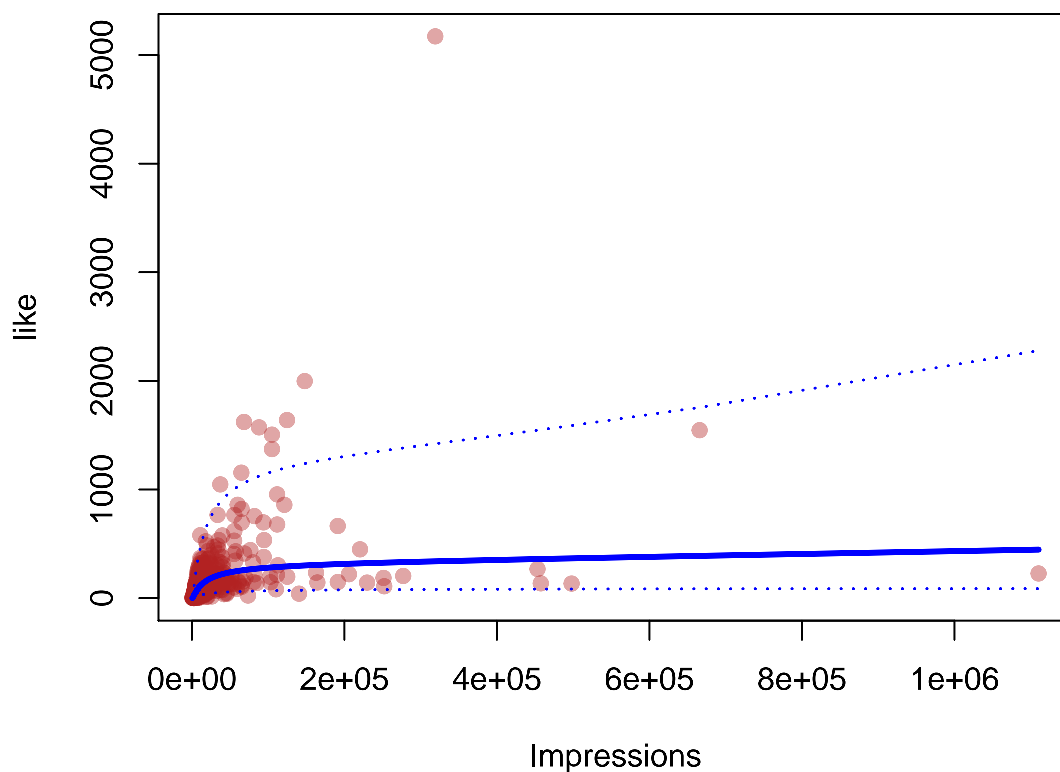
Note that, these confidence and prediction intervals are designed for the mean function $\mu(x) = E(Y)$ and for Y_{new} and in our example Y is `log(like+1)`. The latter is a one to one monotonically increasing function of `like` so we can easily transform these back to the original scale. For any $a = log_e(like + 1)$, the inverse transformation is $like = e^a - 1$.

The prediction interval can be transformed in this way with the same prediction proabilities. So too can the confidence intervals for $\mu(x)$ **except** that the new intervals will be for $e^{\mu(x)} - 1$ and **not** for the expected value of `like`. Transforming the x axis is no problem for either.

Applying the transforms we can get back to the original scales; Here for the prediction intervals.

```
plot(fb$Impressions, fb$like,
     main = "Facebook: 90% prediction intervals",
     xlab = "Impressions",
     ylab = "like",
     pch=19,
     col=adjustcolor("firebrick", 0.4)
     )

# or the fitted (predicted) values for y
# at each point (N.B in order of X!)

centre <- exp(predict(facebook.fit3)) -1
lines(fb$Impressions[Xorder],
      centre[Xorder],
      col="blue", lwd=3)

#  If you want to be really certain you find
```

```
#  facebook, say 90% of the time
#
predTransformed <- exp(pred90.fit3 ) - 1

lines(fb$Impressions[Xorder],
      predTransformed[,'lwr'][Xorder],
      col="blue", lwd=1.5,
      lty=3)
lines(fb$Impressions[Xorder],
      predTransformed[,'upr'][Xorder],
      col="blue", lwd=1.5,
      lty=3)
```



1.3 Aside: Interactive plots with loon

`loon` is an `R` package that provides highly interactive and **extendable** graphics that we will use from time to time. It is available on the Math (not CS) machines. You can also download it from the website

`http://waddella.github.io/loon/`

Follow the instructions there under "installation"; to learn about its functionality look under `learn`, especially `displays`.

Here is a quick (though more complicated than we need here) example. It allows us to quickly shift focus back and forth between confidence intervals for μ and prediction intervals for Y_{new}.

```
library(loon)

p <- l_plot(fb$x, fb$y,
            linkingGroup="Facebook",
            xlabel="log(Impressions)",
            ylabel="log(like + 1)",
            color = "firebrick",
            glyph="ocircle",
            showScales=TRUE,
            showGuides=TRUE,
            itemlabel=row.names(fb),
            title="Facebook")

# Choose lots of new X values (in order) for the plot
# First let's just get the xlim and ylim again so that
# the interactive loon code more or less stands on its own
# (once the data has been read in)
#
xlim <- extendrange(fb$x)
ylim <- extendrange(fb$y)
newX <- seq(min(xlim), max(xlim), length.out=200 )

# Get intervals for these new X values
#
#  confidence intervals for the mean
#
conf95 <- predict(facebook.fit3, data.frame(x=newX),
                  interval="confidence",
                  level=0.95)
# Now some with the same probability level as our
# prediction intervals
pred95 <- predict(facebook.fit3, data.frame(x=newX),
                  interval="prediction",
                  level=0.95)
#
#  We want to add these as polygons to the scatterplot
#  (polygons just to better mark the regions)
#
#  We first build a layer group for the intervals
#  The grouping is not necessary but will be convenient
#  in interacting with the plot layers
#
#  Confidence interval group
intervals <- l_layer_group(p,
                       label="Intervals",
                       index="end")

#  The above is presently empty, so we will add layers to it.
#
#  We now layer each set of intervals
#  as a layer within the intervals group.
#
```

```r
l_layer_polygon(p,
                x=c(newX,rev(newX)),
                y=c(conf95[,'lwr'],rev(conf95[,'upr'])),
                color="lightblue2",
                parent=intervals,
                label="95% confidence",
                index="end")
#
# We now do the same for the prediction intervals
#

l_layer_polygon(p,
                x=c(newX,rev(newX)),
                y=c(pred95[,'lwr'], rev(pred95[,'upr'])),
                color="lightgrey",
                parent=intervals,
                label="95% prediction",
                index="end")
```

Note that we can link this plot with others, like a histogram

```r
library(loon)
l_hist(log(fb$comment +1), linkingGroup="Facebook")
l_plot(fb$Paid, fb$Impressions,linkingGroup="Facebook")
l_plot(fb$Category, fb$Type,linkingGroup="Facebook")
```

1.3.1 Interactive plot states

An important characteristic of interactive plots is that their state is changed by the interaction. The analyst interacts with the data using the mouse (or other gestures) and changes the display characteristics of the data immediately. It is important, then, to be able to access the values of any state of interest in a plot – perhaps just to record a result of an analysis or to use the state's values in further analysis.

In `loon`, many of a plot's states are immediately accessible from the command line (or programmatically) in `R`.

```r
# We have our scatterplot
p

# which is simply a string in R, BUT has "class" "loon"
#
# We can query the value of its states.
# For example, the colour of the points
# (each is a hexadecimal string)

head(p['color'])

# which points are currently selected
head(p['selected'])

# which are active
head(p['active'])

# what linking group is the plot currently participating in
p['linkingGroup']
```

```r
# and so on.
# To see all of the states available:
names(l_info_states(p))

#
# See l_help() for more info
#
```

This kind of interactive exploration can raise a number of questions about the quality of our fitted $\widehat{\mu}(x)$. For example, you might give the points different colours (or shapes) depending on the value of another variate (e.g. `Paid` or `Type` or `Category`). We might then ask whether the same curve should be used for all colours/shapes?

Exercise Suppose, for example, that we thought we should have a different curve fitted for posts which were either paid or not. How might the $\mathbf{X}$ matrix and coefficient vector $\boldsymbol{\beta}$ be changed to adapt to two separate curves (one for `Paid = 0`, one for `Paid = 1`)?

Alternatively, we could **change the states programmatically** or from the console:

```r
# make all points selected in the display.
#
sel <- p['selected']
p['selected'] <- TRUE
#
# and turn them back again
p['selected'] <- sel

# randomly change the size of the first 10 points
sizes <- p['size']
p['size'][1:10] <- sample(seq(from=50, to=100, by=2), 10)
# and return them to the original size
p['size'][1:10] <- sizes[1:10]

# same for colour, etc.
cols <- p['color']
p['color'] <- 'black'

# the glyph
glyphs <- p['glyph']
p['glyph'][fb$Category == "Product"] <- "circle"
p['color'][fb$Category == "Product"] <- "red"
p['glyph'][fb$Category  == "Action"] <- "square"
p['color'][fb$Category  == "Action"] <- "green"
p['glyph'][fb$Category == "Inspiration"] <- "triangle"
p['color'][fb$Category == "Inspiration"] <- "blue"

## Seems like the "Action" catefory post produces fewer likes
# And returning them to the original colours
p['color'] <- cols
p['glyph'] <- glyphs
```

1.4 Weighted least squares

Consider the same linear model as before

$$\begin{aligned} y_i &= \mu(\mathbf{x}_i) + r_i \\ &= \mathbf{x}_i^T \boldsymbol{\beta} + r_i \end{aligned}$$

for $i = 1, \ldots, N$ except now we introduce non-negative weights w_i and choose $\widehat{\boldsymbol{\beta}}$ to be that value which minimizes $\Delta(\boldsymbol{\beta})$ where

$$\begin{aligned} \Delta(\boldsymbol{\beta}) &= \textstyle\sum_{i=1}^N w_i r_i^2 \\ &= \textstyle\sum_{i=1}^N w_i \left(y_i - \mathbf{x}_i^T \boldsymbol{\beta}\right)^2 \end{aligned}$$

We could minimize this as before to find the solution, but it is immediate from the solution for ordinary least-squares. To see this, we let

$$\begin{aligned} r_i^* &= \sqrt{w_i} \times r_i \\ y_i^* &= \sqrt{w_i} \times y_i \end{aligned}$$

and

$$\mathbf{x}_i^* = \sqrt{w_i} \times \mathbf{x}_i.$$

Then $\Delta(\boldsymbol{\beta}) = \sum_{i=1}^N (r_i^*)^2$ and we are back to ordinary least squares!

Letting $\mathbf{W} = diag(w_1, w_2, \ldots, w_N)$ be an $N \times N$ diagonal matrix of weights, the corresponding matrices and vectors can be written as

$$\begin{aligned} \mathbf{r}^* &= \mathbf{W}^{1/2} \times \mathbf{r} \\ \mathbf{y}^* &= \mathbf{W}^{1/2} \times \mathbf{r} \end{aligned}$$

and

$$\mathbf{X}^* = \mathbf{W}^{1/2} \times \mathbf{X}.$$

and the ordinary least-squares solution here will be

$$\begin{aligned} \widehat{\boldsymbol{\beta}} &= \left((\mathbf{X}^*)^T(\mathbf{X}^*)\right)^{-1} (\mathbf{X}^*)^T(\mathbf{y}^*) \\ &= \left((\mathbf{W}^{1/2}\mathbf{X})^T(\mathbf{W}^{1/2}\mathbf{X})\right)^{-1} (\mathbf{W}^{1/2}\mathbf{X})^T(\mathbf{W}^{1/2}\mathbf{y}) \\ &= \left(\mathbf{X}^T\mathbf{W}^{1/2}\mathbf{W}^{1/2}\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{W}^{1/2}\mathbf{W}^{1/2}\mathbf{y} \\ &= \left(\mathbf{X}^T\mathbf{W}\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{W}\mathbf{y} \end{aligned}$$

which of course is now the **weighted least-squares solution** for the original problem.

From this we have

$$\begin{aligned}\widehat{\boldsymbol{\mu}} &= \mathbf{X}\widehat{\boldsymbol{\beta}} \\ &= \mathbf{X}\left(\mathbf{X}^T\mathbf{W}\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{W}\mathbf{y},\end{aligned}$$

$$\widehat{\mathbf{r}} = \left(\mathbf{I}_N - \mathbf{X}\left(\mathbf{X}^T\mathbf{W}\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{W}\right)\mathbf{y},$$

and

$$\widehat{\sigma}^2 = \widehat{\mathbf{r}}^T\widehat{\mathbf{r}}/(N-p).$$

Confidence intervals and prediction intervals are determined as before.

In 'R' we can calculate the weighted least-squares solutions by simply handing a weight vector to the fitting procedure `lm(...)`. See `help("lm")`.

2 Whither the weights?

So where do they weights come from? The answer, as usual, is "it depends". It depends on what problem is being addressed and what the purpose of the weighting might be.

The weights might, for example, be a consequence of the generative model, or they might simply be because we hope to give more emphasis to some observations in determining the estimates (i.e. attributes of interest) than to others. It could even be a simple way to select some points for participation in the calculation and not others.

2.1 Differing variances

This might be the most familiar to you.

Consider again our generative response model:

$$Y_i = \mu(\mathbf{x}_i) + R_i$$

for $i = 1, ..., n$. Usually, the stochastic part of the model has the random variates R_i be independently distributed with $E(R_i) = 0$ and $SD(R_i) = \sigma$.

For some problems (e.g. each response y_i in the model is actually the arithmetic average of n_i observations taken at the same value $\mathbf{x}_i$), it makes sense to suppose that the standard deviations of the residuals are not identical but are each proportional (by differing but known amounts) to an unknown constant σ. That is, we have

$$SD(R_i) = k_i\sigma$$

where σ is unknown but $k_i > 0$ is known for every value of $i = 1, \ldots, N$. If we specified that the distribution of each R_i also be normal, then we would also have $R_i \sim N(0, k_i^2\sigma^2)$.

The response model, in vector form, would as before be

$$\mathbf{Y} = \boldsymbol{\mu} + \mathbf{R}$$

or

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \mathbf{R}$$

but now with $Var(\mathbf{R}) = \sigma^2\mathbf{D}_k^2$ where $\mathbf{D}_k = diag(k_1, \ldots, k_N)$ is a diaginal matrix having the k_is as its non-zero (and positive) diagonal elements.

If we left-multiply both sides of the equations by D_k^{-1} we have

$$\mathbf{Y}^* = \mathbf{X}^*\boldsymbol{\beta} + \mathbf{R}^*$$

say, where $\mathbf{Y}^* = D_k^{-1}\mathbf{Y}$, $\mathbf{X}^* = D_k^{-1}\mathbf{X}$, and $\mathbf{R}^* = D_k^{-1}\mathbf{R}$.

Now $Var(\mathbf{R}^*) = \sigma^2 I_N$ and we are back to a situation where ordinary least squares might reasonably be applied.

Moreover, minimizing $\sum(r_i^*)^2$ is equivalent to minimizing

$$\sum_{i=1}^{N}\left(\frac{1}{k_i}\right)^2 \; r_i^2$$

which is a weighted least squares problem with weights $w_i = 1/k_i^2$.

The solution is again

$$\widehat{\boldsymbol{\beta}} = \left(\mathbf{X}^T\mathbf{W}\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{W}\mathbf{y}$$

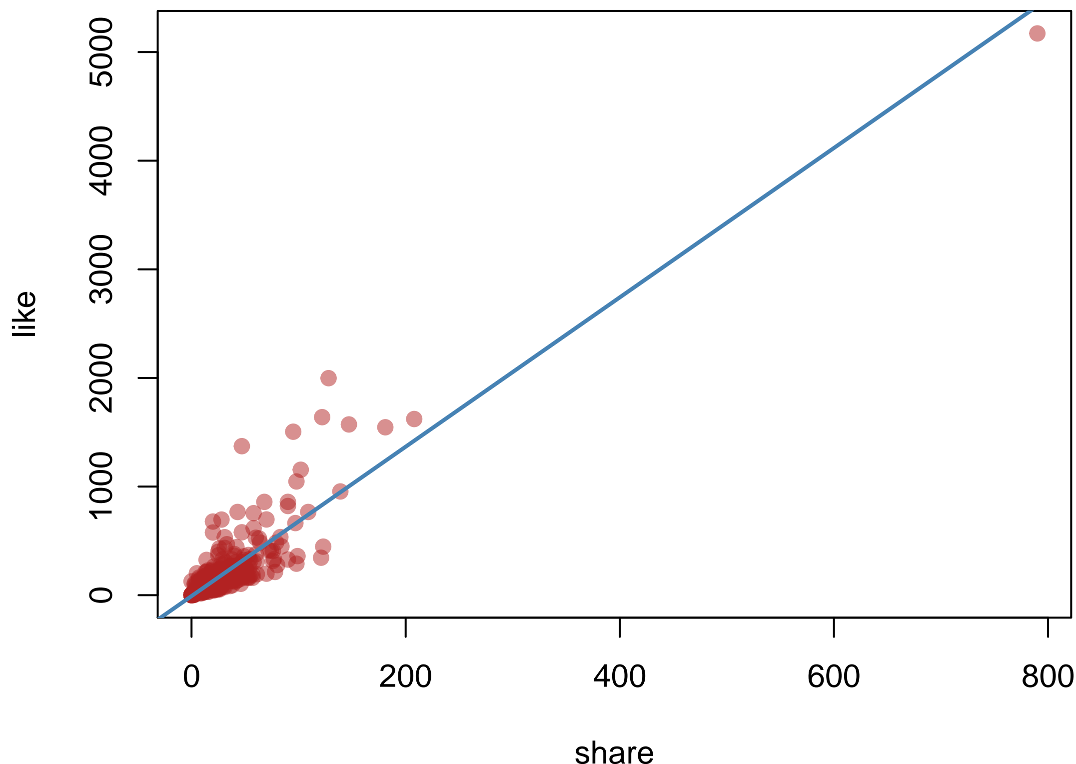
where $\mathbf{W} = D_k^{-2}$.

2.2 Zero one weights

If the weights are restricted to be either exactly zero or exactly one, then we are simply declaring which observations we choose to involve in the fit and which we do not.

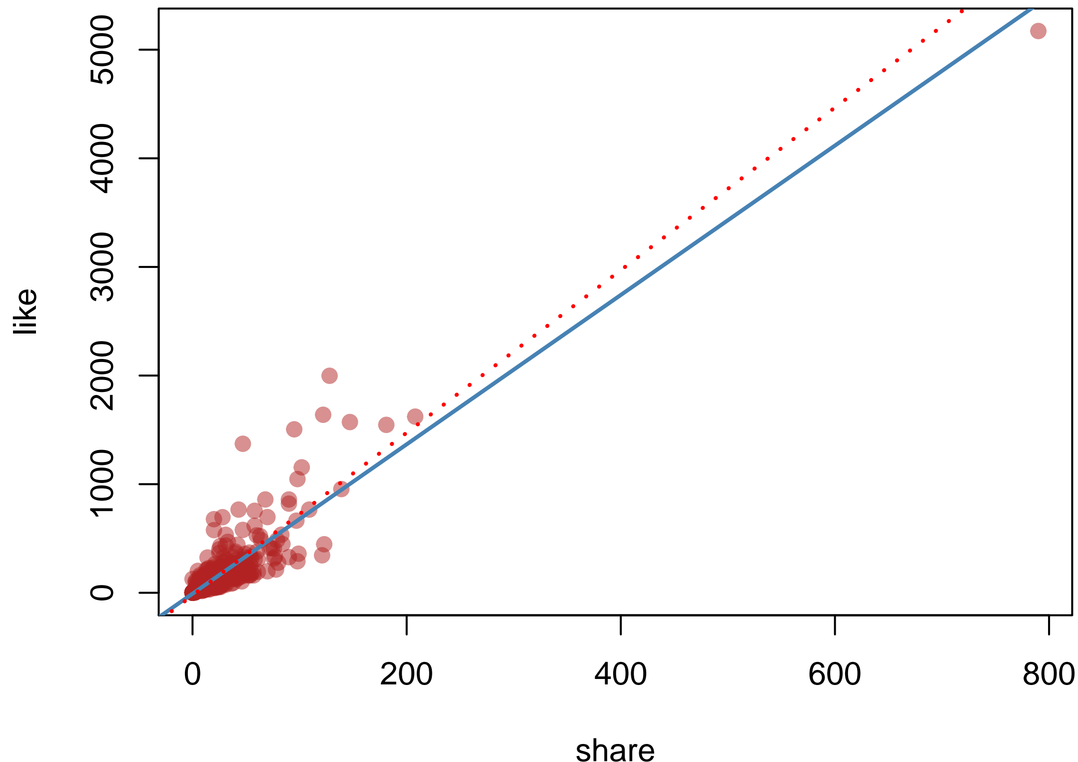
2.2.1 Removing a point

As we saw in the Facebook data, there can be points in our plot which might have a significant effect on our estimated function $\widehat{\mu}(x)$. For example, in the figure below we might wonder how much influence some of the outlying points might have on the coefficient estimates of the fitted line.



Would the fit be different, for example, if outlying points were not allowed to have any influence on the fit? For example, if we were to fit the line to all points **except** that in the top right corner, how would the function change?

We can easily illustrate the difference for this example. The fitted $\widehat{\mu}(x)$ based on all but this one point is shown below in red, together with the fit in blue based on all points.



Above we simply excluded the point from the data set and fitted our model $\mu(x)$ to the remaining data. The removal of the point could also have been effected by weighted least squares, simply by giving weight 0 to that point.

The identity of the point in `fb` is had as

```
n <- nrow(fb)
outlier <- (1:n)[!restVals] # Note "not" the rest of the values.
```

It is point numbered 241.

We can now achieve the same fit by using weighted least squares on all the points with just zero-one weights.

```
## First a summary of the fit without the outlier
summary(like_share_out.fit)
```

```
##
## Call:
## lm(formula = like ~ share, data = fb[restVals, ])
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -537.18  -50.74   -7.55   26.86 1063.42
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept) -23.4745     8.8290  -2.659   0.0081 **
## share         7.4848     0.2448  30.579   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 137.6 on 492 degrees of freedom
## Multiple R-squared:  0.6552, Adjusted R-squared:  0.6545
## F-statistic: 935.1 on 1 and 492 DF,  p-value: < 2.2e-16
```

```
## And now using all the data but with the outlier's weight set to 0.
wts <- rep(1,n) # All weights
wts[outlier] <- 0
like_share_out_wls.fit <-  lm(like ~ share,  data = fb, weights=wts)
summary(like_share_out_wls.fit)
```

```
##
## Call:
## lm(formula = like ~ share, data = fb, weights = wts)
##
## Weighted Residuals:
##     Min      1Q  Median      3Q     Max
## -537.18  -50.72   -7.34   26.73 1063.42
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -23.4745     8.8290  -2.659   0.0081 **
## share         7.4848     0.2448  30.579   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 137.6 on 492 degrees of freedom
## Multiple R-squared:  0.6552, Adjusted R-squared:  0.6545
## F-statistic: 935.1 on 1 and 492 DF,  p-value: < 2.2e-16
```

Check how these summaries compare.
Setting the weight to zero is as if the observation was never there!

2.2.2 Demonstration: Effect of individual points on the fit.

In loon, we could be more ambitious and define new interactions. For example, it is possible to "bind" functionality to changes in a plot's states. If we can have state changes cause other changes, say in the calculation of some attributes, it affords us the possibility of asking many more "what if?" questions – both of the data and of its summary attributes. Plots from `loon` can be easily extended so that they dynamically adapt to the data. This allows us to explore how individual points affect the fit.

For example, noting that we seem to have some outlying points in our scatterplot of points, we might be interested in removing some of these points, if not from the plots, than at least from our calculations.

We can effect that with a little programming. The plot will track the following interactions/events with it:

- removing points: points are removed by selecting them and deactivating them (all points can be reactivated from the `loon inspector`.
- relocating points: points can also be moved around by holding down `<ctrl>-<button-1>` and moving the mouse. (groups of selected points can be moved with `<shift>-<ctrl>-<button-1>`).

Here is the code to get a more dynamic plot.

```
library(loon)
#
# Here we will build a plot with a fitted function on top
#
# The fitted function will update itself if the points change
# in some way (as specified at the end of this block of code).
#
# It will be simplest to work with a new data frame
#
like_share.df <- data.frame(x = fb$share, y = fb$like)

like_share.plot <- l_plot(like_share.df$x,
                          like_share.df$y,
                          linkingGroup = "Facebook",
                          xlabel = "shares",
                          ylabel = "likes",
                          color = "firebrick",
                          glyph ="circle",
                          showScales = TRUE,
                          showGuides = TRUE,
                          showItemlabels = TRUE,
                          itemlabel = row.names(fb),
                          title = "Facebook")

like_share.fit <- lm(y ~ x, data=like_share.df)

like_share.xrange <- extendrange(like_share.df$x)

like_share.newX <- seq(from=min(like_share.xrange),
                       to=max(like_share.xrange),
                       length.out=200)

# We will layer two lines on, one that stays true
# to the original points (in blue)
# and another (in red) which will adapt to the
# points dynamically
#
# Here is the original fit.  It won't change.
like_share.mu_hat_original   <- l_layer_line(like_share.plot,
                                    x=like_share.newX,
                                    y=predict(like_share.fit,
                                              newdata=data.frame(x=like_share.newX)),
                                    color="blue",
                                    linewidth=3,
                                    label="Original mu_hat(x)",
                                    index="end")

# This one will change as we interact with the points
like_share.mu_hat   <- l_layer_line(like_share.plot,
                                    x=like_share.newX,
                                    y=predict(like_share.fit,
                                              newdata=data.frame(x=like_share.newX)),
                                    color="red",
```

```
                                    linewidth=3,
                                    label="Changing mu_hat(x)",
                                    index="end")

# Now the fun begins.
# We need to write a function that we can attach to the plot.
# It will be executed whenever the specified characterisics (`events') are changed
#
updateRegression <- function(p, fit) {
  ## Identify the active points
  points_to_be_fit <- p['active']
  ## The coordinates of only these points will be used in the fit
  ##
  ## We'll also only use the temporary locations whenever
  ## someone has moved the points, otherwise the original coordiates.
  ##
  ## For x
  xnew <- p['xTemp']
  if (length(xnew) == 0) { # back to original coordinate then
    xnew <- p['x']
  }
  ## For y
  ynew <- p['yTemp']
  if (length(ynew) == 0) { # back to original coordinate then
    ynew <- p['y']
  }
  ## Redo our fit
  ##
  formula <- formula(fit)  # Note, use whatever the formula was
                           # (MUST be in terms of x and y for next to work)
  fit.temp <- lm(formula,
                 data=subset(data.frame(x=xnew, y=ynew),
                             points_to_be_fit) )

  xrng <- extendrange(xnew)

  ## the x values
  ##
  xvals.temp <- seq(from=min(xrng),to=max(xrng),
                    length.out=200)

  muhat.temp <- predict(fit.temp, newdata=data.frame(x=xvals.temp))

  ## And configure the curve to match the fit
  l_configure(like_share.mu_hat,
              x=xvals.temp,
              y=muhat.temp
  )
  ## Update the tcl language's event handler
  tcl('update', 'idletasks')
}
```

```r
# Here is we "bind" actions to state changes.
# Whenever the active state, or the temporary locations
# of the points change on the plot p,
# an anonymous function with no arguments will be called.
# This in turn will call the function "updateRegression"
# on the plot p, using the fit like_share.fit
#
l_bind_state(like_share.plot,
             c("active","xTemp","yTemp"),
             function() {updateRegression(like_share.plot, like_share.fit)}
)
```

If you have `loon` installed, play around with the above to see the effects. Try one point first, then several. Remember that reactivating will return all points to the display (and to the estimation) and that you may return selected points (highlighted via the `Select` section of the inspector) to their original positions with the circular arrow button in the `move` subsection of the `Modify` section of the inspector.

We can also see the effect on our original cubic model, as well as the effect on confidence and prediction intervals. The following adds new dynamic layers to our earlier interactive plot where `log(like +1)` is fitted to a cubic function of ‘log(Impressions).

```r
#
library(loon)
#
# This implementation is a straightforward extension of the previous one.
# In addition to the changing fit, we will also show the 95% prediction
# interval changing as well.
#
dynamic <- l_layer_group(p,
                         label="dynamic intervals",
                         index="end")

# then the individual polygons
#
dynamic95 <- l_layer_polygon(p,
                             x=c(newX,rev(newX)),
                             y=c(pred95[,'lwr'],rev(pred95[,'upr'])),
                             color="lightpink",
                             parent=dynamic,
                             label="dynamic 95% prediction",
                             index="end")

# And add the estimated function itself
#
mu_hat   <- l_layer_line(p,
                         x=newX,
                         y=predict(facebook.fit3,
                                   newdata=data.frame(x=newX)),
                         color="blue",
                         linewidth=3,
                         label="mu_hat(x)",
                         index="end")
```

```r
updatePredictions <- function(p, fit) {
  ## use only the active points for regression
  points_to_be_fit <- p['active']
  ## these will be the coordinates to use for regression
  ## We'll use the temporary locations if
  ## someone has moved the points
  ## otherwise the original coordiates.
  ##
  ## For x
  xnew <- p['xTemp']
  if (length(xnew) == 0) {
    xnew <- p['x']
  }
  ## For y
  ynew <- p['yTemp']
  if (length(ynew) == 0) {
    ynew <- p['y']
  }
  ## Redo our fit
  ##
  formula <- formula(fit)
  fit.temp <- lm(formula,
                 data=subset(data.frame(x=xnew, y=ynew),
                             points_to_be_fit)
                 )

  xrng <- extendrange(xnew)

  ## the intervals
  ##
  xpvals.temp <- seq(from=min(xrng),to=max(xrng),
                     length.out=200)

  pred95.temp <- predict(fit.temp, data.frame(x=xpvals.temp),
                         interval="prediction", level=0.95)

  ## update the prediction intervals
  ##
  l_configure(dynamic95,
              x=c(xpvals.temp,rev(xpvals.temp)),
              y=c(pred95.temp[,'lwr'],rev(pred95.temp[,'upr'])))

  ## And the fit
  l_configure(mu_hat,
              x=xpvals.temp,
              y=pred95.temp[,'fit']
              )
  ## Update the tcl language's event handler
  tcl('update', 'idletasks')
}

# We now "bind" actions to state changes on the existing plot `p`
#
```

```r
# Th function "updateRegression"
# on the plot p, using the fit facebook.fit3
#
l_bind_state(p,
             c("active","xTemp","yTemp"),
             function() {updatePredictions(p, facebook.fit3)}
             )
```

Once we have the above in place, we can deactivate points which will effectively delete them from the data set and recalculate all of the prediction intervals.

Again, it is worth taking some time to explore this effect.

2.3 Robust estimation

The untransformed values of `like` versus `share` exhibited a number of outlying points, and moving them about in the dynamic plots demonstrated the influence that each had on the fit. The more outlying in y the greater the influence on the fit. Note also that if the point is outlying in x as well, it has greater **leverage** in that it amplifies the effect of outlying y.

The impact, or **influence** of (x_1, y_1) on the fitted values (and our predictions) is eliminated by removing it from the data *before* fitting $\widehat{\mu}$ (or equivalently $\widehat{\boldsymbol{\beta}}$). This can also be done by using weighted least squares where a weight of $w_i = 0$ is assigned to any point to be removed, and a weight of $w_i = 1$ to all the rest.

This seems rather drastic and requires examining the points to see where these outlying y values might be. It might be better if we had small (not necessarily zero) weights for outlying points and larger weights for those that are not outlying. Clearly, it might be desirable to have the weights get smaller the farther away the outliers are as well.

One way to achieve that is to consider our estimation criterion more generally. We might more generally choose $\widehat{\boldsymbol{\beta}}$ to be the value that minimized:

$$\Delta(\boldsymbol{\beta}) = \sum_{i=1}^{N} \rho(y_i - \mathbf{x}_i^T \boldsymbol{\beta})$$

for some specific function $\rho(\cdot)$. Differentiating $\Delta(\boldsymbol{\beta})$ with respect to $\boldsymbol{\beta}$ we have

$$\begin{array}{rcl}
\frac{\partial}{\partial \boldsymbol{\beta}} \Delta(\boldsymbol{\beta}) & = & \sum_{i=1}^{N} \frac{\partial}{\partial \boldsymbol{\beta}} \rho(y_i - \mathbf{x}_i^T \boldsymbol{\beta}) \\
 & = & -\sum_{i=1}^{N} \rho'(y_i - \mathbf{x}_i^T \boldsymbol{\beta}) \ \mathbf{x}_i^T \\
 & = & -\sum_{i=1}^{N} \rho'(r_i) \ \mathbf{x}_i^T
\end{array}$$

Setting this to zero, we could solve for a value $\widehat{\boldsymbol{\beta}}$. The set of equations to be solved can be written as

$$\sum_{i=1}^{N} \psi(r_i) \ \mathbf{x}_i = \mathbf{0}$$

where we $\psi(r) = \rho'(r)$.

These sorts of equations should be familiar. For example if $\rho(r) = r^2$, the first equation is just the least

squares criterion and with $\psi(r) = 2r$ the second equation becomes the "normal equations":

$$\begin{aligned}
\mathbf{0} &= \textstyle\sum_{i=1}^N \psi(r_i)\ \mathbf{x}_i \\
&= 2\textstyle\sum_{i=1}^N r_i\ \mathbf{x}_i \\
&= 2\textstyle\sum_{i=1}^N (y_i - \mathbf{x}_i^T\widehat{\boldsymbol{\beta}})\ \mathbf{x}_i \\
&= 2\textstyle\sum_{i=1}^N (\mathbf{x}_i y_i - \mathbf{x}_i\mathbf{x}_i^T\widehat{\boldsymbol{\beta}}) \\
&= 2(\textstyle\sum_{i=1}^N (\mathbf{x}_i y_i) - \sum_{i=1}^N (\mathbf{x}_i\mathbf{x}_i^T\widehat{\boldsymbol{\beta}})) \\
&= 2(\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{X}\widehat{\boldsymbol{\beta}}).
\end{aligned}$$

Another familiar example would be if the **log likelihood** were expressable as

$$\begin{aligned}
l(\boldsymbol{\beta}) &= \textstyle\sum_{i=1}^N l_i(\boldsymbol{\beta}) \\
&= \textstyle\sum_{i=1}^N l(r_i).
\end{aligned}$$

Then maximizing the log-likelihood $l(\boldsymbol{\beta})$ would be equivalent to minimizing $\Delta(\boldsymbol{\beta})$ with $\rho(r) = -l(r)$. Similarly, solving the equation involving $\psi(\cdot)$ for $\widehat{\boldsymbol{\beta}}$ becomes the same as setting the **score function** to zero and solving for $\widehat{\boldsymbol{\beta}}$.

This similarity to **maximum likelihood estimation** gives the name to estimators $\widetilde{\boldsymbol{\beta}}$ that come from minimizing $\sum_{i=1}^N \rho(r_i)$, namely **M-estimators** for "maximum-likelihood type estimators".

Interestingly, we can also write the estimating equations as equations to solve a weighted least-squares problem:

$$\begin{aligned}
\mathbf{0} &= \textstyle\sum_{i=1}^N \psi(r_i)\ \mathbf{x}_i \\
&= \textstyle\sum_{i=1}^N \frac{\psi(r_i)}{r_i} r_i\ \mathbf{x}_i \\
&= \textstyle\sum_{i=1}^N w(r_i) \times (y_i - \mathbf{x}_i^T\widehat{\boldsymbol{\beta}})\ \mathbf{x}_i \\
&= \textstyle\sum_{i=1}^N w_i(y_i - \mathbf{x}_i^T\widehat{\boldsymbol{\beta}})\ \mathbf{x}_i
\end{aligned}$$

where $w_i = w(r_i) = \psi(r_i)/r_i$. The last equation suggests that the solution is really weighted least squares, namely $\widehat{\boldsymbol{\beta}} = \left(\mathbf{X}^T\mathbf{W}\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{W}\mathbf{y}$ with $\mathbf{W} = diag(w_1, \ldots, w_n)$.

Clearly the estimate $\widehat{\boldsymbol{\beta}}$ depends on the weights w_i. Unfortunately, the weights $w_i = \psi(r_i)/r_i$ and $r_i = y_i - \mathbf{x}_i^T\boldsymbol{\beta}$ is not known because $\boldsymbol{\beta}$ being unknown. With an estimate of β however, we could estimate r_i and hence w_i.

This suggests a possible algorithm for finding an estimate for β:

0. $j \leftarrow 0$.
1. Given an estimate $\widehat{\boldsymbol{\beta}}^{(j)}$ compute residuals

$$\widehat{r}_i^{(j)} = y_i - \mathbf{x}_i^T\widehat{\boldsymbol{\beta}}^{(j)}$$

   for all $i = 1, \ldots, N$.

2. Compute new weights

$$w_i^{(j)} = \frac{\psi\left(\widehat{r}_i^{(j)}\right)}{\widehat{r}_i^{(j)}}$$

for all $i = 1, \ldots, N$ and form

$$\mathbf{W}^{(j)} = diag(w_1^{(j)}, \ldots, w_N^{(j)}).$$

3. Get the new weighted least squares estimate

$$\widehat{\boldsymbol{\beta}}^{(j+1)} = \left(\mathbf{X}^T\mathbf{W}^{(j)}\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{W}^{(j)}\mathbf{y}.$$

4. $j \leftarrow j+1$
5. Repeat steps 1-4 until "convergence".

This algorithm is an example of an **iteratively reweighted least squares** algorithm.

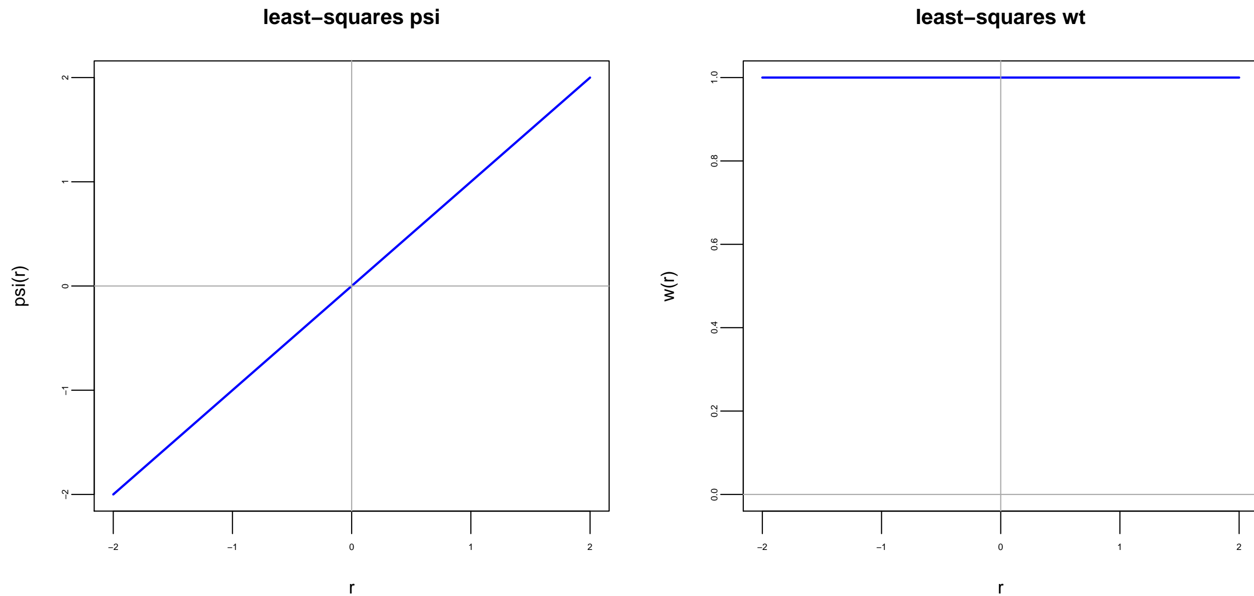
Clearly, there are some details that need to be filled in. For example, where does the zero-th estimate come from? When has the algorithm converged? What function $\psi(\cdot)$ should we use?

This last question becomes interesting, especially since we can now think of $\psi(r_i)/r_i$ as the weight given to the ith observation in a weighted least squares solution. For large values of r_i we would like this to be small. And for small values of r_i we would like it to be nearer to 1.

It might be helpful to consider the graph of the ψ and the corresponding weight function $w(r)$.

In ordinary least squares, $\psi(r) = r$ and the weights become uniformly unity. The plots look like:

```r
r <- seq(-2,2,length.out=200)

psi <- function (r) {r}

weight <- function(r) {
  zeros <- r == 0
  wt <- r
  wt[zeros] <- 1
  wt[!zeros] <- psi(r[!zeros])/r[!zeros]
  wt
  }

parOptions <- par(mfrow=c(1,2))

plot(r, psi(r),
     type="l", col="blue",
     xlab="r", ylab="psi(r)",
     lwd=2,
     cex.axis=0.5, tck=-0.05,
     main="least-squares psi")
abline(h=0, col="darkgrey")
abline(v=0, col="darkgrey")

plot(r, weight(r),
     type="l", col="blue",
     xlab="r", ylab="w(r)",
     lwd=2, ylim=c(0,1),
     cex.axis=0.5, tck=-0.05,
     main="least-squares wt")
abline(h=0, col="darkgrey")
abline(v=0, col="darkgrey")
```

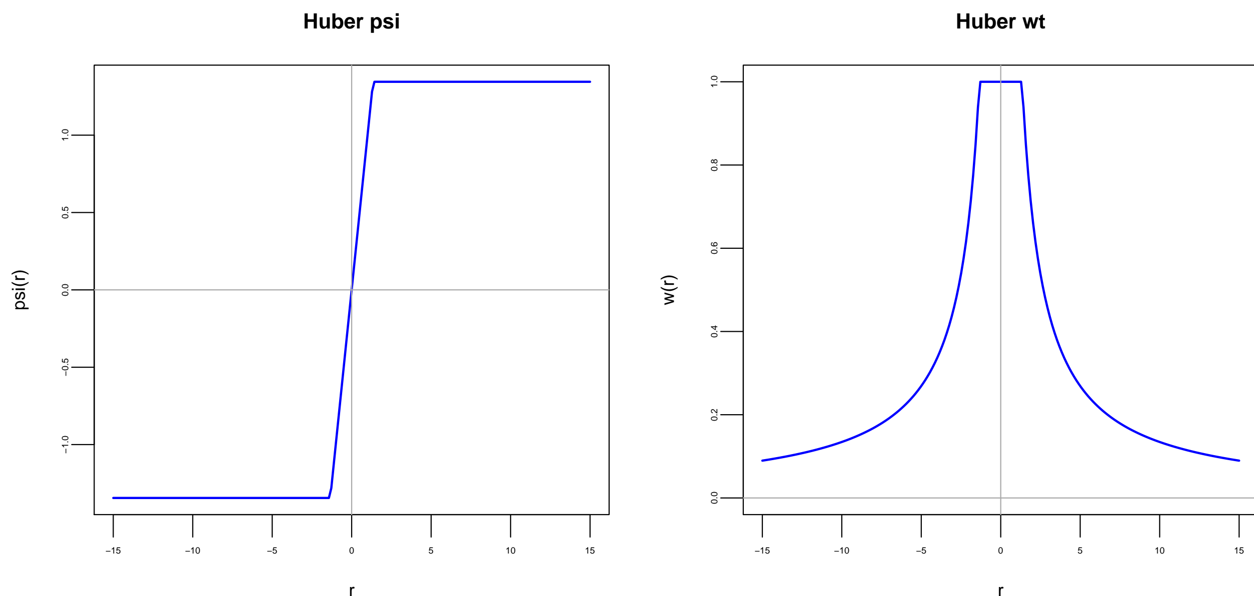



```
par(parOptions)
```

This suggests that the “psi-function”” be chosen so that it does not continue unbounded. Once such suggestion is the so-called “Huber psi” function after its author, Peter Huber. It is largely like least-squares but is turned to a horizontal line at some point. Here is Huber’s proposal:

```
r <- seq(-15,15, length.out=200)

Huberpsi <- function (r, b=1.345) {
  psivals <- r
  flatvals <- abs(r) > b
  psivals[flatvals] <- b * sign(r[flatvals])
  psivals
}


Huberweight <- function(r, b=1.345){
  zeros <- r == 0
  wt <- r
  wt[zeros] <- 1
  wt[!zeros] <- Huberpsi(r[!zeros],b)/r[!zeros]
  wt
  }

parOptions <- par(mfrow=c(1,2))

plot(r, Huberpsi(r),
     type="l", col="blue",
     xlab="r", ylab="psi(r)",
     lwd=2,
     cex.axis=0.5, tck=-0.05,
     main="Huber psi")
abline(h=0, col="darkgrey")
abline(v=0, col="darkgrey")
```

```r
plot(r, Huberweight(r),
     type="l", col="blue",
     xlab="r", ylab="w(r)",
     lwd=2, ylim=c(0,1),
     cex.axis=0.5, tck=-0.05,
     main="Huber wt")
abline(h=0, col="darkgrey")
abline(v=0, col="darkgrey")
```

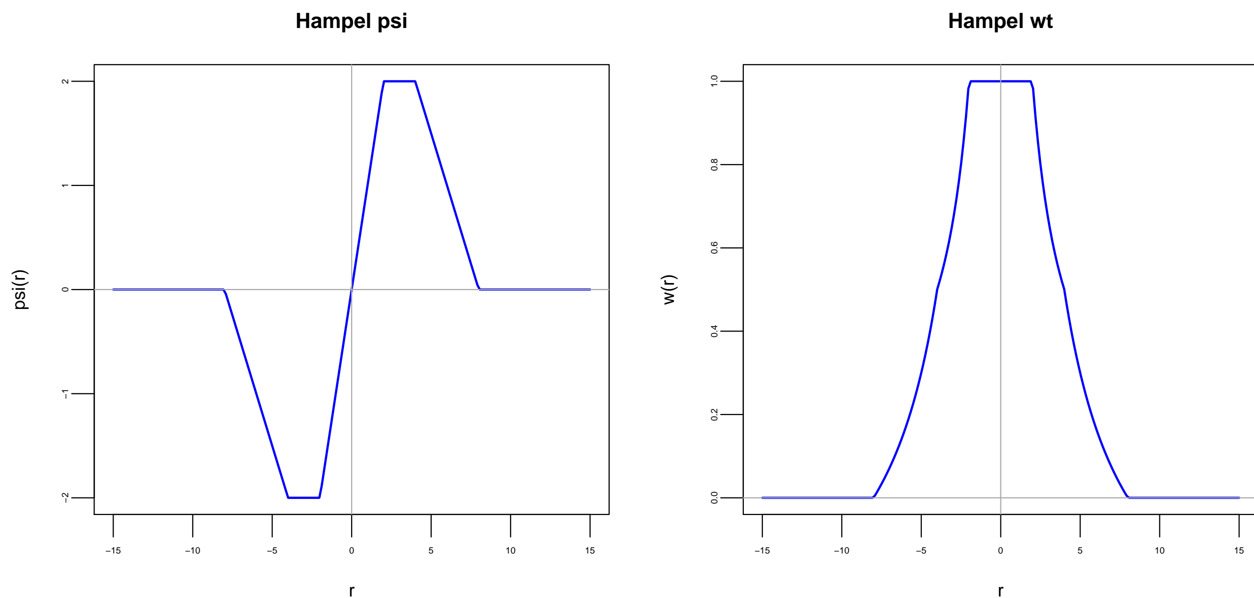



```r
par(parOptions)
```

Another idea then might be to have the psi function "redescend", that is actually come back to the horizontal axis. There are many such suggestions. Two that are popular are "Hampel's psi", named after Frank Hampel, and "Tukey's bisquare weight" psi or "Tukey's biweight" psi named after John Tukey.

```r
r <- seq(-15, 15, length.out=200)

Hampelpsi <- function (r, a=2, b=4, c=8) {
  psivals <- rep(0, length(r))
  # middle
  middlevals  <-{abs(r) < a}
  psivals[middlevals] <- r[middlevals]
  # flat
  flatvals <- {a <= abs(r)} & {abs(r) < b}
  psivals[flatvals] <- a * sign(r[flatvals])
  # down
  downvals <- {b <= abs(r)} & {abs(r) < c}
  psivals[downvals] <- ( (c - abs(r[downvals]) ) / (c-b) ) * a * sign(r[downvals])
  # otherwise zero
  psivals
}


Hampelweight <- function(r, a=2,b=4,c=8){
```

```
  zeros <- r == 0
  wt <- r
  wt[zeros] <- 1
  wt[!zeros] <- Hampelpsi(r[!zeros],a,b,c)/r[!zeros]
  wt
  }

parOptions <- par(mfrow=c(1,2))

plot(r, Hampelpsi(r),
     type="l", col="blue",
     xlab="r", ylab="psi(r)",
     lwd=2,
     cex.axis=0.5, tck=-0.05,
     main="Hampel psi")
abline(h=0, col="darkgrey")
abline(v=0, col="darkgrey")

plot(r, Hampelweight(r),
     type="l", col="blue",
     xlab="r", ylab="w(r)",
     lwd=2, ylim=c(0,1),
     cex.axis=0.5, tck=-0.05,
     main="Hampel wt")
abline(h=0, col="darkgrey")
abline(v=0, col="darkgrey")
```

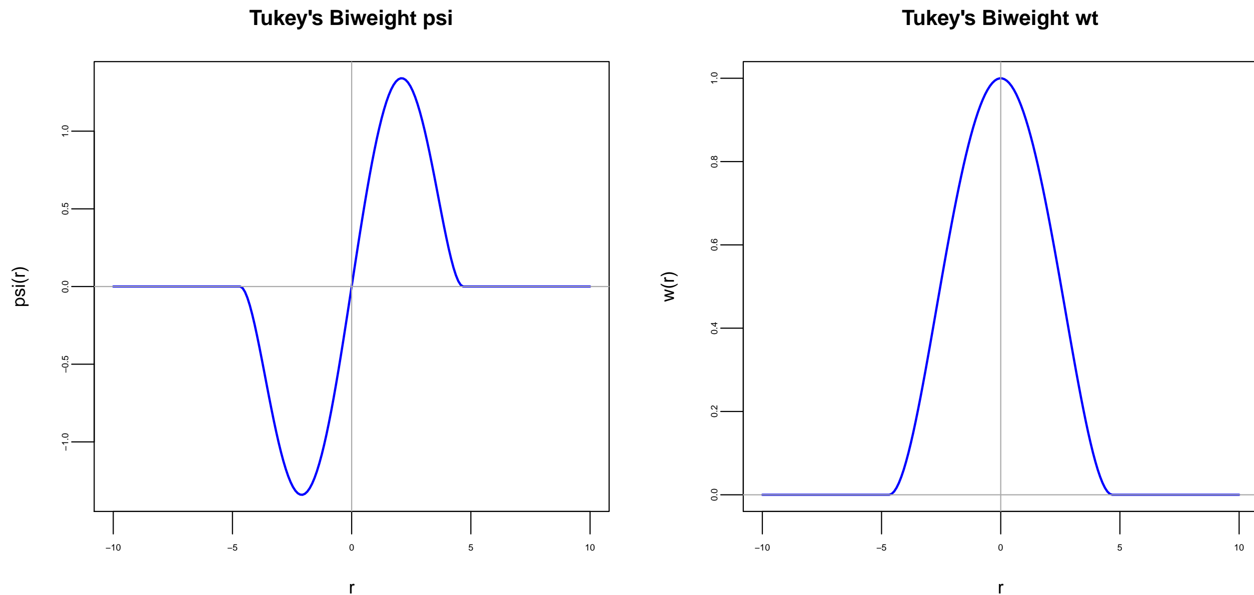



```
par(parOptions)
```

Tukey's biweight is much like Hampel's but smoother. Namely,

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

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

TukeyBiweightpsi <- function (r, c=4.685) {
  psivals <- rep(0, length(r))
  # middle
  middlevals  <-{abs(r) <= c}
  psivals[middlevals] <- r[middlevals] * (1 - (r[middlevals]/c)^2)^2
  # otherwise zero
  psivals
}


TukeyBiweight <- function(r, c=4.685){
  zeros <- r == 0
  wt <- r
  wt[zeros] <- 1
  wt[!zeros] <- TukeyBiweightpsi(r[!zeros],c)/r[!zeros]
  wt
  }

parOptions <- par(mfrow=c(1,2))

plot(r, TukeyBiweightpsi(r),
     type="l", col="blue",
     xlab="r", ylab="psi(r)",
     lwd=2,
     cex.axis=0.5, tck=-0.05,
     main="Tukey's Biweight psi")
abline(h=0, col="darkgrey")
abline(v=0, col="darkgrey")

plot(r, TukeyBiweight(r),
     type="l", col="blue",
     xlab="r", ylab="w(r)",
     lwd=2, ylim=c(0,1),
     cex.axis=0.5, tck=-0.05,
     main="Tukey's Biweight wt")
abline(h=0, col="darkgrey")
abline(v=0, col="darkgrey")
```




```r
par(parOptions)
```

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

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

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

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

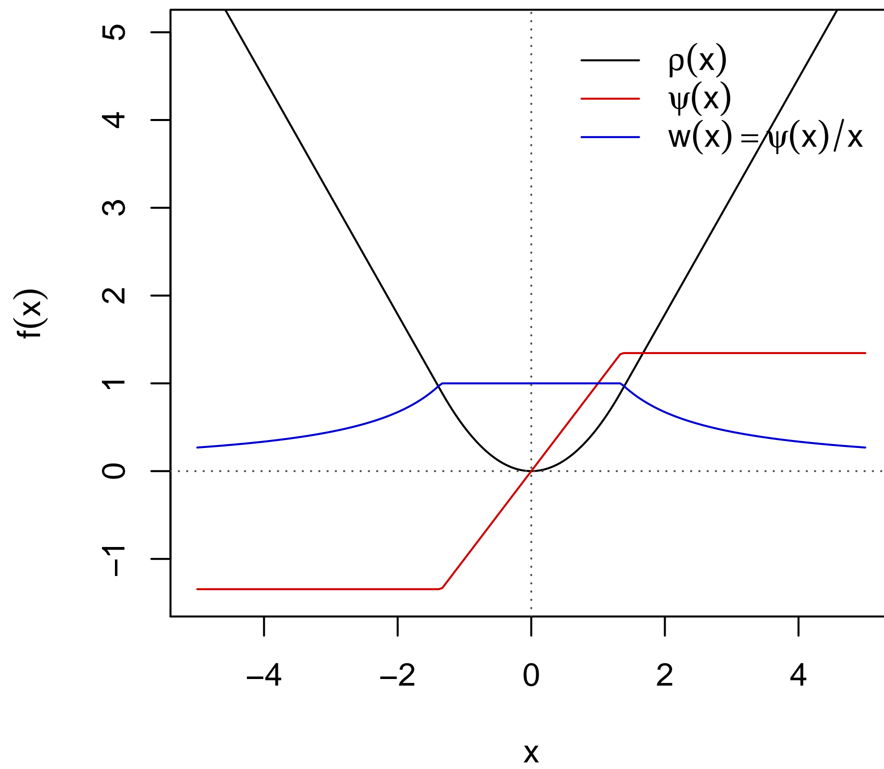
So we can actually choose ψ based on how we think it will behave and infer ρ from our choice.

In the `robustbase` package in `R` you can easily plot these functions and more.

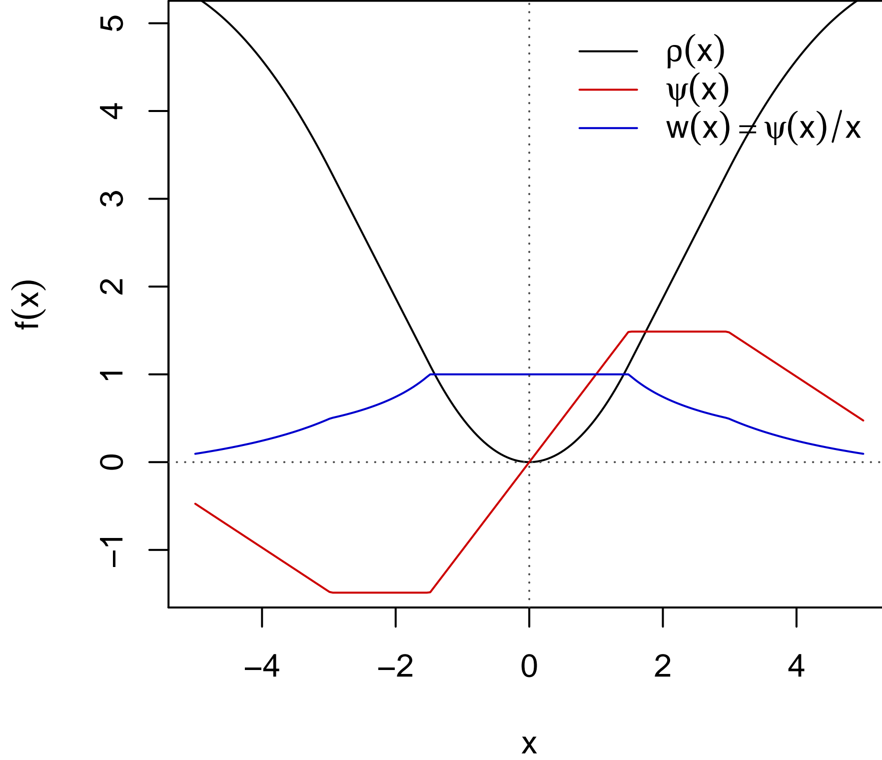
```r
library("robustbase")
```

```
## Warning: package 'robustbase' was built under R version 3.3.2
```

```r
# Huber
r <- seq(-5,5, length.out=200)
plot(huberPsi, r, ylim=c(-1.4, 5),
     which=c("rho", "psi", "wgt"),
     leg.loc="topright", main=FALSE)
```




```
# Hampel

r <- seq(-5,5, length.out=200)
plot(hampelPsi, r, ylim=c(-1.4, 5),
     which=c("rho", "psi", "wgt"),
     leg.loc="topright", main=FALSE)
```



2.3.1 Location-scale equivariance

For simplicity, let us suppose that our response model is simply

$$y_i = \mu + r_i \quad \text{for} \;\; i = 1, \ldots, N.$$

That is, we are just interested in a common mean model; there are no $\mathbf{x}_i$ s or alternatively there is a single scalar $x_i = 1$ for all i.

There is no substantive difference between the above model and

$$y_i^* = \mu^* + r_i^* \quad \text{for} \;\; i = 1, \ldots, N$$

where $y_i^* = b \times y_i + a$ for known constant scalars a and b. This would be just as if we decided to change y_i from a measurement of temperature in degrees Celsius to the same measurement in degrees Fahrenheit. The only change is in the location and scale of the measuring units. Of course, the equation also forces μ and r_i to follow suit with $\mu^* = b\mu + a$ and $r_i^* = br_i$.

If then we had a statistic $T(y_1, \ldots, y_N)$ that was meant to estimate μ in the first expression of the model, we would very much like to estimate μ^* by $aT(y_1, \ldots, y_N) + b$. That is, we would like the statistic $T(y_1, \ldots, y_N)$ to be **location and scale equivariant** meaning that

$$T(by_1 + a, \ldots, by_N + a) = bT(y_1, \ldots, y_N) + a.$$

As constructed, our M estimators are always thought of as a function of the residual, in this case $r = y - \mu$, so any estimator will be **location equivariant** but not necessarily **scale-equivariant**. Some M-estimates like the arithmetic average (corresponding to the least-square ψ) will also be **scale equivariant**. but others, like the Tukey biweight ψ, will not necessarily be so.

To ensure that we have both location and scale equivariance, the M-estimators are calculated not on r but on $r/(kS_n)$ where S_n is some estimate of the scale of the ys (or, equivalently, of the rs) and k is a constant

of our choice (chosen perhaps to give the estimator some good performance characteristic whenever the data actually do come from a normal distribution).

For the bisquare weight function, then, we would be using

$$\psi\left(\frac{r}{kS_n}\right) = \begin{cases} \frac{r}{kS_n} \left(1 - \left(\frac{r}{ckS_n}\right)^2\right)^2 & \left|\frac{r}{kS_n}\right| \le c \\ 0 & \left|\frac{r}{kS_n}\right| > c. \end{cases}$$

This sort of scaling is used for the other M-estimators as well.

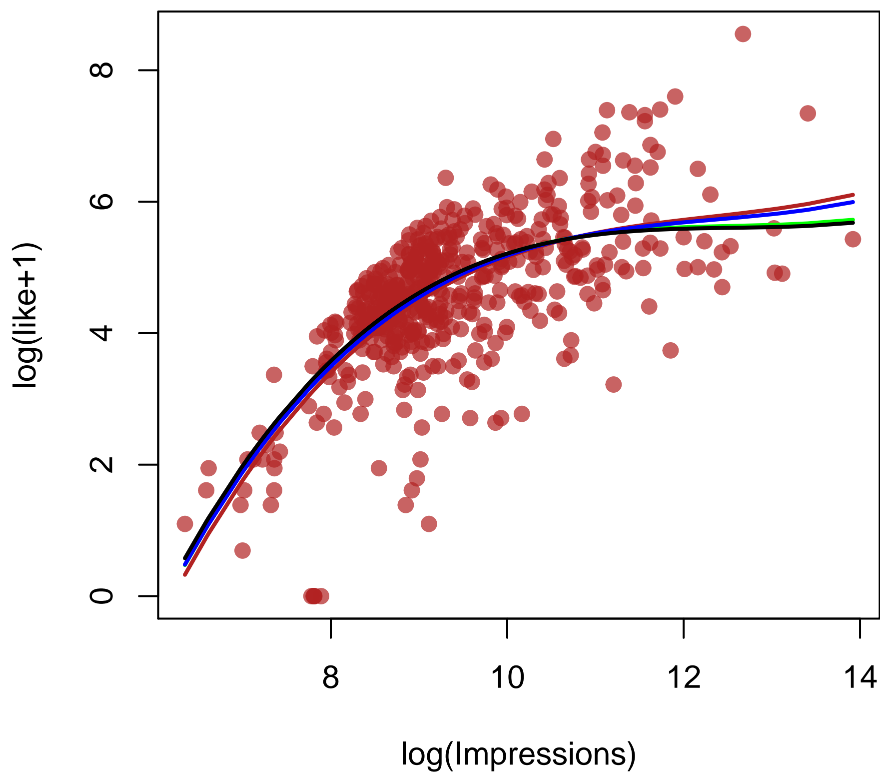
Note that the scale estimate S_n also needs to be determined from the data. It is sometimes determined simultaneously with the location estimates μ_i and also with attention to it being robust to outliers as well.

2.3.2 Robust linear models in R

To fit these functions to data, we use a "robust" regression, where robust here means essentially resistant to outliers (in the y direction).

```
library(MASS)

# It will be simplest to work with a new data frame
#

facebook.fit3 <- lm(y ~ x + I(x^2) + I(x^3),
                 data=fb)

#
# Using robust linear model function "rlm"
#
facebook.fit.Huber <- rlm(formula(facebook.fit3),
                       data=fb,
                       psi="psi.huber")

facebook.fit.Hampel <- rlm(formula(facebook.fit3),
                        data=fb,
                        psi="psi.hampel")

facebook.fit.Tukey <- rlm(formula(facebook.fit3),
                       data=fb,
                       psi="psi.bisquare")

plot(fb$x, fb$y,
     #xlim = xlim, ylim = ylim,
     main = "Huber",
     xlab = "log(Impressions)",
     ylab = "log(like+1)",
     pch=19,
     col=adjustcolor("firebrick", 0.7)
     )

Xorder <- order(fb$x)

lines(fb$x[Xorder],
```

```
      predict(facebook.fit3)[Xorder],
      col="firebrick", lwd=2)

lines(fb$x[Xorder],
      predict(facebook.fit.Huber)[Xorder],
      col="green", lwd=2)

lines(fb$x[Xorder],
      predict(facebook.fit.Hampel)[Xorder],
      col="blue", lwd=2)

lines(fb$x[Xorder],
      predict(facebook.fit.Tukey)[Xorder],
      col="black", lwd=2)
```



We might see how sensitive these are to outliers. Execute the following code in R and explore the effect of moving a single point around (say the outlier in the top left of the scatterplot). Also try selecting several points and moving them around (`<ctrl><shift><button1>`).

```
library(loon)

p <- l_plot(fb$x,
            fb$y,
            linkingGroup = "Facebook",
            xlabel = "log(Impressions)",
            ylabel = "log(like +1)",
            color = "firebrick",
```

```
            glyph ="circle",
            showScales = TRUE,
            showGuides = TRUE,
            showItemlabels = TRUE,
            itemlabel = row.names(fb),
            title="Comparing psi functions")

xlim <- extendrange(fb$x)
ylim <- extendrange(fb$y)
newX <- seq(min(xlim), max(xlim), length.out=200 )


fits <- l_layer_group(p,
                      label="Various fitting methods",
                      index="end")

# then the individual fits
#
curve.ls <- l_layer_line(p,
                         x=newX,
                         y=predict(facebook.fit3,
                                   newdata=data.frame(x=newX)
                                   ),
                         color="firebrick",
                         linewidth=4,
                         label="Least squares fit",
                         parent=fits,
                         index="end")
#  Plot for the names
#
p_names <- l_plot(title="Methods")
l_layer_text(p_names,
             x=0, y=8,
             text="Least-squares",
             size=20,
             color="firebrick",
             label="Least-squares",
             index="end")
l_scaleto_world(p_names)

curve.huber <- l_layer_line(p,
                            x=newX,
                            y=predict(facebook.fit.Huber,
                                      newdata=data.frame(x=newX)
                                      ),
                            color="green",
                            linewidth=4,
                            label="Huber psi",
                             parent=fits,
                            index="end")
l_layer_text(p_names,
             x=0, y=6,
             text="Huber psi",
```

```
             size=20,
             color="green",
             label="Huber",
             index="end")
l_scaleto_world(p_names)


curve.hampel <- l_layer_line(p,
                             x=newX,
                             y=predict(facebook.fit.Hampel,
                                       newdata=data.frame(x=newX)
                                       ),
                             color="blue",
                             linewidth=4,
                             label="Hampel psi",
                             parent=fits,
                             index="end")

l_layer_text(p_names,
             x=0, y=4,
             text="Hampel psi",
             size=20,
             color="blue",
             label="Hampel",
             index="end")
l_scaleto_world(p_names)

curve.tukey <- l_layer_line(p,
                            x=newX,
                            y=predict(facebook.fit.Tukey,
                                      newdata=data.frame(x=newX)
                                      ),
                            color="black",
                            linewidth=4,
                            label="Tukey bisquare psi",
                            parent=fits,
                            index="end")

l_layer_text(p_names,
             x=0, y=2,
             text="Tukey biweight",
             size=20,
             color="black",
             label="Tukey",
             index="end")
l_scaleto_world(p_names)


updateFits <- function(p, fit) {
  ## use only the active points for regression
  sel <- p['active']
  ## which coordinates to use for regression
  ## We'll use the temporary locations if
```

```
## someone has moved the points
## otherwise the original coordiates.
##
## For x
xnew <- p['xTemp']
if (length(xnew) == 0) {
  xnew <- p['x']
}
## For y
ynew <- p['yTemp']
if (length(ynew) == 0) {
  ynew <- p['y']
}
##
## Get data
##
formula <- formula(fit)
newdata <- subset(data.frame(x=xnew, y=ynew), sel)
xrng <- extendrange(xnew)

xvals_curve <- seq(min(xrng), max(xrng), length.out=200 )
## Redo each fit
fit.ls <- lm(formula, data=newdata)
fit.Huber <- rlm(formula, data=newdata, psi="psi.huber")
fit.Hampel <- rlm(formula, data=newdata, psi="psi.hampel")
fit.Tukey <- rlm(formula, data=newdata, psi="psi.bisquare")
##
## update the fitted curves
##
l_configure(curve.ls,
            x=xvals_curve,
            y=predict(fit.ls,
                      newdata=data.frame(x=xvals_curve)
                      )
            )

l_configure(curve.huber,
            x=xvals_curve,
            y=predict(fit.Huber,
                      newdata=data.frame(x=xvals_curve)
                      )
            )

l_configure(curve.hampel,
            x=xvals_curve,
            y=predict(fit.Hampel,
                      newdata=data.frame(x=xvals_curve)
                      )
            )

l_configure(curve.tukey,
            x=xvals_curve,
            y=predict(fit.Tukey,
```

```
                         newdata=data.frame(x=xvals_curve)
                         )
              )


  ## Update the tcl language's event handler
  tcl('update', 'idletasks')
}

# Here is we "bind" actions to state changes.
# Whenever the active state, or the temporary locations
# of the points change on the plot p,
# an anonymous function with no arguments will be called.
# This in turn will call the function "updateRegression"
# on the plot p, using the fit facebook.fit3
#
l_bind_state(p,
             c("active","xTemp","yTemp"),
             function() {updateFits(p, facebook.fit3)}
             )
```

2.3.3 Sensitivity curves

You should have noticed that each of these curves vary in their sensitivity to the vertical position of the outlier. The least-squares curve chases the outlier and, being forced by the model to be cubic, can produce a very poor fit on the remainder of the points. The three M-estimators seem to be less sensitive and in order from the Huber-ψ, to the redescending Hampel-ψ, to the Tukey biweight ψ, with the biweight being the least sensitive and the Huber-ψ always being somewhat affected by the outlier no matter how far away it is (the weight never quite gets to zero).

These various fitted curves can be thought of as descriptors of a population, that as population attributes. In choosing which population attribute we are interested in we might like to know how sensitive it is to a few points. We might, for example, prefer to have attributes of the population that describe the "bulk" of the population and so are not that sensitive to the values of one or two individuals in the population.

To get a handle on this we might consider a statistic $T_N(y_1, \ldots, y_N)$ as a potentially interesting population attribute. It might not be of interest if, for example, it can be extremely affected by a single individual. We imagine then its value, $T_N(y_1, \ldots, y_{N-1}, y)$ say, for our population of size N but where the value of one of the variates, say y_N, has been replaced by a *value of our choice*, namely y. How sensitive is the statistic to the value of y?

To answer this, we consider the **sensitivity curve**

$$\begin{aligned} SC(y) &= \frac{T_N(y_1,\ldots,y_{N-1},y) \; - \; T_{N-1}(y_1,\ldots,y_{N-1})}{\frac{1}{N}} \\ &= N \; \left(\; T_N(y_1,\ldots,y_{N-1},y) \; - \; T_{N-1}(y_1,\ldots,y_{N-1}) \; \right) \end{aligned}$$

which is simply the difference in the values $T_N(\cdot) - T_{N-1}(\cdot)$ compared to the size of the "contamination" $\frac{1}{N}$.

Considering this as a function of y, we can probably name a few desirable characteristics. We might not want it to be unbounded, for example, as that would mean it would be possible for a single individual in the population to completely determine the value of the attribute.

A couple of familiar scalar valued population attributes might help illustrate the interpretative value of this function.

2.3.3.1 Example: Arithmetic average

Suppose

$$T_N(y_1, \ldots, y_N) = \frac{1}{N}\sum_{i=1}^{N} y_i = \overline{y}$$

is the arithmetic average of the variate values (or the population/sample mean, depending on circumstances). Then

$$T_{N-1}(y_1, \ldots, y_{N-1}) = \frac{1}{N-1}\sum_{i=1}^{N-1} y_i$$

and

$$\begin{aligned} T_N(y_1, \ldots, y_{N-1}, y) &= \tfrac{1}{N}\left(\textstyle\sum_{i=1}^{N-1} y_i + y\right) \\ &= \tfrac{y}{N} + \tfrac{N-1}{N} T_{N-1}(y_1, \ldots, y_{N-1}) \end{aligned}$$

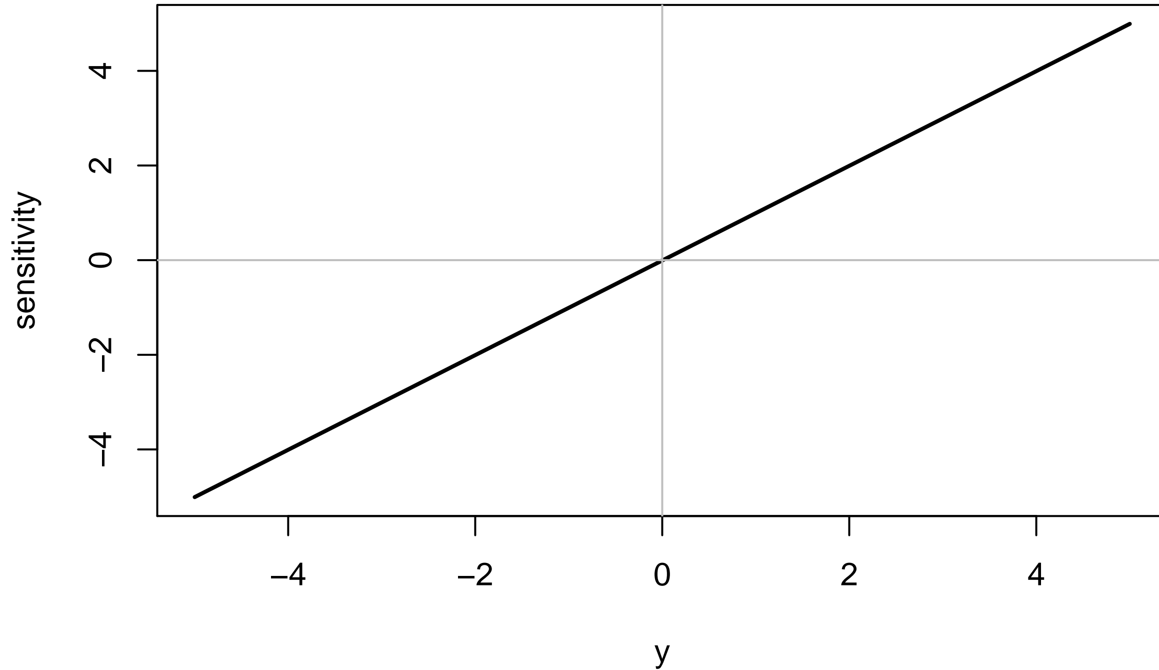
and we have

$$\begin{aligned} SC(y) &= y + (N-1)T_{N-1}(y_1, \ldots, y_{N-1}) - NT_{N-1}(y_1, \ldots, y_{N-1}) \\ &= y - \overline{y}_{N-1}. \end{aligned}$$

As a function of y, this is easily seen to be a straight line with slope 1 and intercept $-\overline{y}_{N-1}$.

We could of course draw this for any particular set of values $y_1, \ldots, y_{N-1}$ as follows:

```r
N <- 1000
# Let's just use a sample of size 999 = 1000 -1 from a N(0,1)
ys <- rnorm(999)
#  y can range from -5 to 5
y <- seq(-5,5, length.out=200)
#
# Sensitivity curve
sc <- y -  mean(ys)

plot(y, sc, type="l", lwd = 2,
     main="Sensitivity curve for the average",
     ylab="sensitivity")
abline(h=0, col="grey")
abline(v=0, col="grey")
```



Look familiar? It looks like the ψ function for least-squares.

Note that the sensitivity curve here gets higher (or lower) without bound as $y \to \infty$ (or as $y \to -\infty$). A single observation can change the average by a huge (even infinite) amount.

Averages may not be the best choice for a population attribute representing the location of a population.

2.3.3.2 Example: Median

Suppose $T_N(y_1, \ldots, y_N)$ is the median. For simplicity, let's also suppose that N is odd, that is $N = 2m + 1$. Let's also suppose that the ordered values are

$$y_{(1)} \leq y_{(2)} \leq \cdots \leq y_{(N)}.$$

Then

$$T_{N-1}(y_1, \ldots, y_{N-1}) = \frac{1}{2}\left(y_{(m)} + y_{(m+1)}\right).$$

Now the value of $T_N(y_1, \ldots, y_{N-1}, y)$ depends on the value of y.

If $y < y_{(m)}$, then $y_{(m)}$ is the median and so $T_N(y_1, \ldots, y_{N-1}, y) = y_{(m)}$.

Similarly, if $y > y_{(m+1)}$, then $y_{(m+1)}$ is the median and so $T_N(y_1, \ldots, y_{N-1}, y) = y_{(m+1)}$.

When y is between these two, that is $y_{(m)} \leq y \leq y_{(m+1)}$, then y itself is the median and $T_N(y_1, \ldots, y_{N-1}, y) = y$.

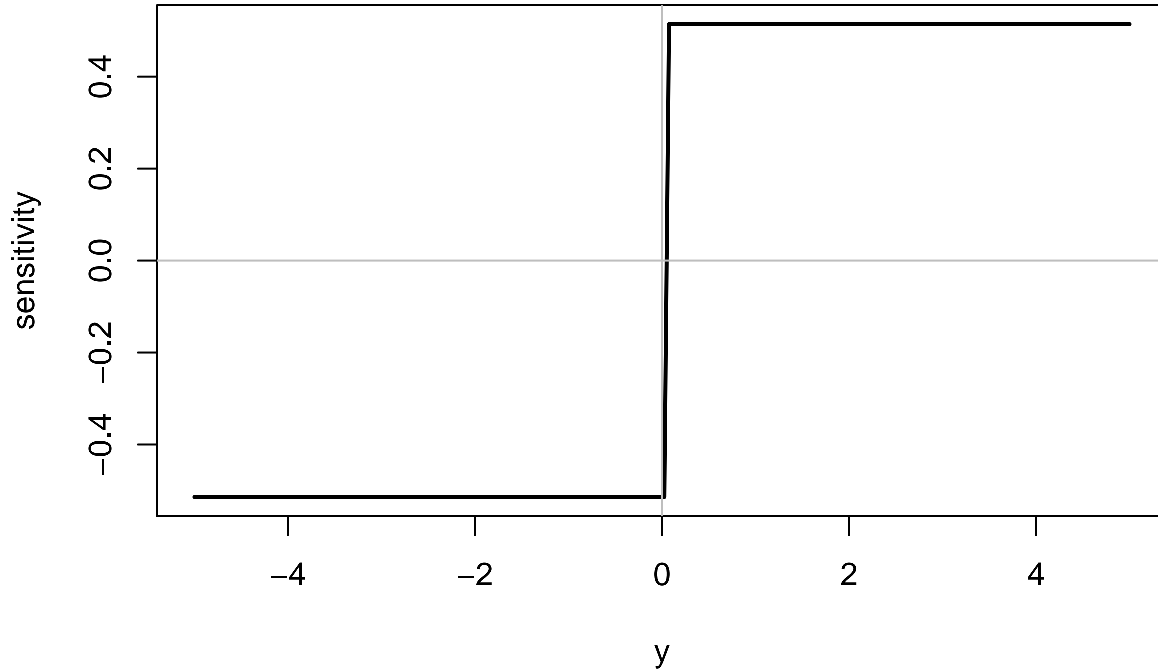
The sensitivity curve is easily seen to be

$$SC(y) = \begin{cases} -\frac{N}{2}\left(y_{(m+1)} - y_{(m)}\right) & \text{if} \quad y < y_{(m)} \\ \\ \frac{N}{2}\left(2y - y_{(m+1)} - y_{(m)}\right) & \text{if} \quad y_{(m)} \leq y \leq y_{(m+1)} \\ \\ \frac{N}{2}\left(y_{(m+1)} - y_{(m)}\right) & \text{if} \quad y > y_{(m+1)} \end{cases}$$

which looks like a negative constant when $y < y_{(m)}$, is a positive constant at $y_{(m+1)} < y$, and is a simple straight line with positive slope when y is between $y_{(m)}$ and $y_{(m+1)}$.

As with the arithmetic average we can draw the sensitivity curve now for the median for any particular sample.

```r
m <- 500
N <- 2 * m + 1
# Let's just use a sample of size N-1 from a N(0,1)
ys <- rnorm(N-1)
y_ordered <- sort(ys)

y_m <- y_ordered[m]
y_mPlusOne <- y_ordered[m+1]

#  y can range from -5 to 5
y <- seq(-5,5, length.out=200)
#
# Sensitivity curve
#
# Construct the three cases out there

case1 <- y < y_m
case2 <- (y_m <= y) & (y <= y_mPlusOne)
case3 <- y_mPlusOne < y

# Get a numeric vector of the right size
#
sc <- vector("numeric", 200)
#
# Now construct sc for the 3 cases
#
sc[case1] <- -N * (y_mPlusOne - y_m) / 2
sc[case2] <- N * ( 2 * y[case2] - y_mPlusOne - y_m) / 2
sc[case3] <- N * (y_mPlusOne - y_m) / 2

plot(y, sc, type="l", lwd = 2,
     main="Sensitivity curve for the median",
     ylab="sensitivity")
abline(h=0, col="grey")
abline(v=0, col="grey")
```



This again should look familiar. It resembles an extreme Huber-like ψ-function.

Unlike the arithmetic average, the sensitivity curve for the median is at least bounded. A single observation cannot change the median by very much. This makes the median a very interesting population attribute for the location of a variate.

2.3.4 Influence functions

The sensitivity curve is defined for a statistic on a given sample. We might like to address the same behaviour for a statistic on any sample from some distribution.

One way to mathematically quantify this behaviour is to imagine that our statistic $T(\cdot)$ can be expressed as a function of the **distribution function** F which generated the variate values. We will write $T(F)$ to emphasise this and so think of $T(F)$ as a population attribute, where the population is possibly infinite and its variate values are distributed as F.

Now we imagine the effect of changing the distibution ever so slightly so that instead of F we have the variate values produced by the mixture distribution $(1-\epsilon)\ F + \epsilon\ \delta_y$ where δ_y is a curious distribution that places all of the probability on a single location y.

One way to think of it is that most, the proportion $(1-\epsilon)$, of the data will come from F but that a small proportion, ϵ, will come from a single value y. In some sense F has been "contaminated" with a proportion ϵ of values at y.

The **influence function** for the statistic T at the distribution F is now defined as

$$IF(y; T, F) = \lim_{\epsilon \to 0} \frac{T(\ (1-\epsilon)F + \epsilon\delta_y) - T(F)}{\epsilon}$$

for those possible points y where the limit exists. For real y we typically think of the influence function as a function of y for a given statistic T. It looks like a measure of the error in T that would be caused by introducing a few outliers at y.

The influence function also looks something like the definition of a derivative which it is, of sorts. Formally, it is the *Gateaux derivative* of T at F in the direction of δ_y. Similarly, it provides a linear approximation to the statistic T at the "contaminated" distribution:

$$T(\ (1-\epsilon)F + \epsilon\delta_y) \approx T(F) + \epsilon\ IF(y; T, F)$$

which suggests that for finite populations/samples we might replace one observation by a single outlier at y and so have

$$T(y_1, \ldots, y_{N-1}, y) \approx T(F_n) + \frac{1}{n}\ IF(y; T, F).$$

Or if we replaced a small fraction of observations $\epsilon = m/N$ with y that

$$T(y_1, \ldots, y_{N-m}, y, \ldots, y) \approx T(F_n) + \frac{m}{n}\ IF(y; T, F).$$

So the influence function gives a sense of the sensitivity of a statistic to contamination from an individual point and even, provided the linear approximation holds, from a small proportion of the sample being contaminated.

But what does it look like? If F is symmetric, the ψ function is odd, and for auxiliarly scale $S = S(F)$, and tuning constant c, it may be shown (for proof see Huber, 1981) that the influence function for that M-estimator is given by

$$\begin{aligned} IF(y; T, F) &= \frac{cS \times \psi\left(\frac{y - T(F)}{cS} \right)}{\int_{-\infty}^{\infty} \psi'\left(\frac{y - T(F)}{cS} \right) f(y) dy} \\ &= (constant) \times (\psi - function). \end{aligned}$$

So the ψ function shows the shape of the influence function for M-estimators.

2.3.4.1 Some distribution theory

For M-estimators, there are some asymptotic distributional results available as well. For example, for M-estimators of location we have that as $n \to \infty$ the statistics are asymptotically normally distributed with

$$\sqrt{N}(T(F_n) - T(F)) \sim N(0, A(T, F))$$

where the asymptotic variance is related to the influence function as

$$A(T, F) = \int (IF(y; T, F))^2 f(y) dy$$

where F_n is the empirical distribution of a random sample of size n from F and f is the density function corresponding to F. As noted above the $IF(y; T, F)$ is expressible in terms of the ψ function.

Similar results hold for the asymptotic multivariate normality of M-estimators in regrssion. Such results can be used to provide test statistics for coefficients in `R`.

```
# For the Tukey biweight
summary(facebook.fit.Tukey)
```

```
##
## Call: rlm(formula = formula(facebook.fit3), data = fb, psi = "psi.bisquare")
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.60043 -0.52668  0.03028  0.46000  2.94592
##
## Coefficients:
##             Value    Std. Error t value
## (Intercept) -35.1393   6.9336    -5.0680
```

```
## x              9.6916   2.1360     4.5373
## I(x^2)        -0.7699   0.2173    -3.5436
## I(x^3)         0.0204   0.0073     2.8026
##
## Residual standard error: 0.7332 on 491 degrees of freedom
```

```
# from which you can get coefficients
coef(facebook.fit.Tukey)
```

```
##  (Intercept)            x       I(x^2)       I(x^3)
## -35.13932076   9.69161190  -0.76994755   0.02042904
```

```
# and a variance-covariance estimate
vcov(facebook.fit.Tukey)
```

```
##              (Intercept)            x       I(x^2)        I(x^3)
## (Intercept)  48.07485863 -14.76715604  1.489710551 -4.932492e-02
## x           -14.76715604   4.56251203 -0.462841996  1.540645e-02
## I(x^2)        1.48971055  -0.46284200  0.047208654 -1.579699e-03
## I(x^3)       -0.04932492   0.01540645 -0.001579699  5.313498e-05
```

Together with asymptotic normality (and asymptotic χ^2 for quadratic forms) this means that tests and approximate confidence regions may be found for any set of linear function of the coefficients.

2.3.5 Breakdown point

Another measure of robustness has been introduced called the **breakdown point** to give an assessment of just how large a proportion of the data might be contaminated before the statistic breaks down. The breakdown point of a statistic is the largest possible fraction of the observations Informally, the breakdown point also provides an upper bound on the proportion contamination for which the above linear approximation could hold.

A more formal definition suitable for response models (due to Donoho and Huber, 1983) can be had as follows.

Our data are N vectors,

$$\mathbf{z}_i = \begin{pmatrix} y_i \\ \mathbf{x}_i \end{pmatrix}$$

for $i = 1, \ldots, N$. For simplicity of notation we can gather these up into a set $Z = \{\mathbf{z}_1, \ldots, \mathbf{z_N}\}$ and imagine a second such set Z_m^* that has replaced m of the $\mathbf{z}_i$ in Z with arbitrary choices $\mathbf{z}_1^*, \ldots, \mathbf{z}_m^*$.

Our statistic T might now be thought of as a function of the a set like Z. For example, for least squares regression we might have $T(Z) = \widehat{\boldsymbol{\beta}}$.

Now consider the worst error associated with swapping out m $\mathbf{z}_i$ for any that are possible:

$$e(m; T, Z) = \sup_{Z_m^*} ||T(Z_m^*) - T(Z)||.$$

If this error is infinite, then we have breakdown. Therefore, if k is the smallest value of m for which the error is infinite, then the (finite sample) breakdown point is k/N. More formally, this is

$$min\left\{\frac{m}{N} : e(m; T, Z) = \infty\right\}.$$

For the average (and for least-squares regression) the breakdown point is $1/N$ or zero asymptotically. For the median, it is $1/2$, in that fully half of the data have to go to infinity before the median breaks down.

Clearly we would like to have fairly high breakdown for our M-estimators. Unfortunately we do not.

When you moved points around in the above example and observed the changes in the various fitted curves, you might also have noticed that all four estimators were quite sensitive to moving a single point away in the horizontal direction.

This is because the M-estimators were constructed to downweight large residuals without any attention paid to outlying $\mathbf{x}$ values. The least squares component will continue to be affected by outlying $\mathbf{x}$ values that have high “leverage”. Recall that this corresponds to high values of h_{ii} where

$$\mathbf{H} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X} = [h_{ij}]_{N\times N}.$$

The influence functions for our estimators do not reflect changes in the values of $\mathbf{x}$.

Some have tried to generalize the M-estimators to so-called “GM-estimators”” whereby in the estimating equation another weight is introduced that is a function of the corresponding $\mathbf{x}_i$ value. In this case at least the breakdown point need not be zero. However, it is a decreasing function of p, the dimensionality of the coefficient vector $\boldsymbol{\beta}$, and drops to zero as $p \rightarrow \infty$.

The M-estimators used in `R` tend to have non-zero breakdown largely because of using a very good estimates of the scaling S_N and good starting values for $\widehat{\boldsymbol{\beta}}^{(0)}$ in the iteration algorithm.

2.3.6 High breakdown methods

Recall that least-squares estimation is minimizing a **sum** of squares. Had we divided by the number of elements in the sum, the least squares is essentially minimizing the **average of the squares**. That is the least-squares estimate, $\widehat{\boldsymbol{\beta}}_{LS}$ is

$$\underset{\boldsymbol{\beta}}{\text{arg min}} \quad \underset{\text{all } i}{\text{average}} \quad (y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2.$$

Perhaps, as Rouseeuw (1984) suggested, the problem is not with the squares but with the average! It might be better to choose $\widehat{\boldsymbol{\beta}}_{LMS}$ to be

$$\underset{\boldsymbol{\beta}}{\text{arg min}} \quad \underset{\text{all } i}{\text{median}} \quad (y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2.$$

This is called **least-median-squares** regression and surprisingly achieves the breakdown point of

$$\frac{\lfloor N/2 \rfloor - p + 2}{N}$$

where $\lfloor x \rfloor$ is the largest integer $\leq x$ and p is the dimension of $\boldsymbol{\beta}$. If we imagine $N \rightarrow \infty$ and p fixed (that is increasing sample size), then the breakdown will be 50%! And for the case of a simple straight line model, we have $p = 2$ and the breakdown is 50% whatever the value of N.

Unlike M-estimators, this method will reject poorly fitting values when they have outlying $\mathbf{x}_i$ values. The drawback is that it is very inefficient to compute and converges at a rate of $N^{-1/3}$.

Rouseeuw also later suggested a more efficient method, again based on thinking of replacing the average sum of squares but this time by the **trimmed average** sum of squares. Here $\widehat{\boldsymbol{\beta}}_{LTS}$ is

$$\underset{\boldsymbol{\beta}}{\text{arg min}} \quad \sum_{i=1}^{k} (r)_{(i)}^2$$

where $(r)_{(i)}^2$ is the ith smallest squared residual.

This is called **least-trimmed-squares** regression and $\widehat{\boldsymbol{\beta}}_{LTS}$ the least trimmed squares estimate. It is much more efficiently calculated (convergence rate of $N^{-1/2}$) than is the least median squares estimate and is nearly as resistant to groups of outlying points in any direction. In fact, if we choose

$$k = \left\lfloor \frac{N}{2} \right\rfloor + \left\lfloor \frac{p+1}{2} \right\rfloor$$

the breakdown point of the least-trimmed-squares estimate is

$$\left(\left\lfloor \frac{N-p}{2} \right\rfloor + 1 \right) /N$$

which is even slightly better than least median squares!

We can again interactively explore these methods again using `loon` in `R` as below.
Execute the following code in `R` and explore the effect of moving a single point around (say the outlier in the top left of the scatterplot). Also try selecting several points and moving them around (`<ctrl><shift><button1>`).

```
library(loon)
library(MASS)

fit_line  <- lm(y~x, data=fb)
fit_cubic <- lm(y~x + I(x^2) + I(x^3), data=fb)
fit.lms1  <- lmsreg(formula(fit_line),  data=fb)
fit.lms3  <- lmsreg(formula(fit_cubic), data=fb)
fit.lts1  <- ltsreg(formula(fit_line), data=fb)
fit.lts3  <- ltsreg(formula(fit_cubic), data=fb)

p <- l_plot(fb$x, fb$y,
            linkingGroup="Facebook",
            xlabel="log(Impressions)",
            ylabel="log(like+1)",
            color = "firebrick",
            showScales=TRUE,
            showGuides=TRUE,
            title="Comparing estimators")


xlim <- extendrange(fb$x)
ylim <- extendrange(fb$y)

newX <- seq(min(xlim), max(xlim), length.out=200)


fits <- l_layer_group(p,
                      label="Various fitting methods",
                      index="end")

# then the individual fits
#
ls <- l_layer_group(p,
                    label="Least Squares",
                    parent=fits,
                    index="end")

curve.ls1 <- l_layer_line(p,
                          x=newX,
                          y=predict(fit_line,
                                    newdata=data.frame(x=newX)
                          ),
                          color="red",
                          linewidth=3,
```

```
                          dash=c(10,5),
                          label="LS line",
                          parent=ls,
                          index="end")

curve.ls3 <- l_layer_line(p,
                          x=newX,
                          y=predict(fit_cubic,
                                    newdata=data.frame(x=newX)
                          ),
                          color="red",
                          linewidth=3,
                          label="LS cubic",
                          parent=ls,
                          index="end")



lms <- l_layer_group(p,
                     label="Least Median",
                     parent=fits,
                     index="end")

curve.lms1 <- l_layer_line(p,
                           x=newX,
                           y=predict(fit.lms1,
                                     newdata=data.frame(x=newX)
                           ),
                           color="blue",
                           linewidth=3,
                           dash=c(10,5),
                           label="LMS line",
                           parent=lms,
                           index="end")

curve.lms3 <- l_layer_line(p,
                           x=newX,
                           y=predict(fit.lms3,
                                     newdata=data.frame(x=newX)
                           ),
                           color="blue",
                           linewidth=3,
                           label="LMS cubic",
                           parent=lms,
                           index="end")


lts <- l_layer_group(p,
                     label="Least Trimmed",
                     parent=fits,
                     index="end")

curve.lts1 <- l_layer_line(p,
```

```r
                           x=newX,
                           y=predict(fit.lts1,
                                     newdata=data.frame(x=newX)
                           ),
                           color="purple",
                           linewidth=3,
                           dash=c(10,5),
                           label="LTS line",
                           parent=lts,
                           index="end")

curve.lts3 <- l_layer_line(p,
                           x=newX,
                           y=predict(fit.lts3,
                                     newdata=data.frame(x=newX)
                           ),
                           color="purple",
                           linewidth=3,
                           label="LTS cubic",
                           parent=lts,
                           index="end")



#  Plot for the names
#
p_names <- l_plot(title="Methods")
l_layer_text(p_names,
             x=0, y=8,
             text="Least-squares",
             size=20,
             color="red",
             label="Least-squares",
             index="end")
l_layer_text(p_names,
             x=0, y=6,
             text="Least-median-squares",
             size=20,
             color="blue",
             label="Least-squares",
             index="end")
l_layer_text(p_names,
             x=0, y=4,
             text="Least-trimmed-squares",
             size=20,
             color="purple",
             label="Least-squares",
             index="end")
l_scaleto_world(p_names)
```

```
updateFits <- function(p, fit1, fit3) {
  ## use only the active points for regression
  sel <- p['active']
  ## which coordinates to use for regression
  ## We'll use the temporary locations if
  ## someone has moved the points
  ## otherwise the original coordiates.
  ##
  ## For x
  xnew <- p['xTemp']
  if (length(xnew) == 0) {
    xnew <- p['x']
  }
  ## For y
  ynew <- p['yTemp']
  if (length(ynew) == 0) {
    ynew <- p['y']
  }
  ##
  ## Get data
  ##
  newdata <- subset(data.frame(x=xnew, y=ynew), sel)
  xrng <- extendrange(xnew)

  xvals_curve <- seq(min(xrng), max(xrng), length.out=200 )

  ## Redo each fit
  formula1 <- formula(fit1)
  fit.ls1 <- lm(formula1, data=newdata)
  fit.lms1 <- lmsreg(formula1, data=newdata)
  fit.lts1 <- ltsreg(formula1, data=newdata)
  ## Redo each fit
  formula3 <- formula(fit3)
  fit.ls3 <- lm(formula3, data=newdata)
  fit.lms3 <- lmsreg(formula3, data=newdata)
  fit.lts3 <- ltsreg(formula3, data=newdata)
  ##
  ## update the fitted curves
  ##
  l_configure(curve.ls1,
              x=xvals_curve,
              y=predict(fit.ls1,
                        newdata=data.frame(x=xvals_curve)
              )
  )
  l_configure(curve.ls3,
              x=xvals_curve,
              y=predict(fit.ls3,
                        newdata=data.frame(x=xvals_curve)
              )
  )
```

```
  l_configure(curve.lms1,
              x=xvals_curve,
              y=predict(fit.lms1,
                        newdata=data.frame(x=xvals_curve)
              )
  )

  l_configure(curve.lms3,
              x=xvals_curve,
              y=predict(fit.lms3,
                        newdata=data.frame(x=xvals_curve)
              )
  )

  l_configure(curve.lts1,
              x=xvals_curve,
              y=predict(fit.lts1,
                        newdata=data.frame(x=xvals_curve)
              )
  )


  l_configure(curve.lts3,
              x=xvals_curve,
              y=predict(fit.lts3,
                        newdata=data.frame(x=xvals_curve)
              )
  )


  ## Update the tcl language's event handler
  tcl('update', 'idletasks')
}

# Here is we "bind" actions to state changes.
# Whenever the active state, or the temporary locations
# of the points change on the plot p,
# an anonymous function with no arguments will be called.
# This in turn will call the function "updateRegression"
# on the plot p, using the fit fb.fit3
#
l_bind_state(p,
             c("active","xTemp","yTemp"),
             function() {updateFits(p, fit_line, fit_cubic)}
)
```

These high breakdown fitting methods are rarely used on their own, but they are often used to provide starting points to give zero weights to outliers or to use in determining the first few steps of an M-estimate. Particularly for the latter case the least trimmed squares estimator is preferred to the least median squares estimator.

`R` offers another suite of robust regression methods to be found in the `robust` package. Here the main function is `lmRob` which tries to find highly efficient regression estimators that also have high breakdown. Note that from the help on `lmRob` we find

> "the solution returned here is an approximation to the true solution based upon a random algorithm (except when"Exhaustive" resampling is chosen). Hence you will get (slightly) different answers each time if you make the same call with a different seed."

A comparison might again be made interactively:

```
library(loon)
library(MASS)

library(robust) # contains lmRob

fit_line  <- lm(y~x, data=fb)
fit_cubic <- lm(y~x + I(x^2) + I(x^3), data=fb)
fit.lmRob1  <- lmRob(formula(fit_line),  data=fb)
fit.lmRob3  <- lmRob(formula(fit_cubic), data=fb)
fit.lts1  <- ltsreg(formula(fit_line), data=fb)
fit.lts3  <- ltsreg(formula(fit_cubic), data=fb)

p <- l_plot(fb$x, fb$y,
            linkingGroup="Facebook",
            xlabel="log(Impressions)",
            ylabel="log(like+1)",
            color = "firebrick",
            showScales=TRUE,
            showGuides=TRUE,
            title="Comparing estimators")


xlim <- extendrange(fb$x)
ylim <- extendrange(fb$y)

newX <- seq(min(xlim), max(xlim), length.out=200)


fits <- l_layer_group(p,
                      label="Various fitting methods",
                      index="end")

# then the individual fits
#
ls <- l_layer_group(p,
                    label="Least Squares",
                    parent=fits,
                    index="end")

curve.ls1 <- l_layer_line(p,
                          x=newX,
                          y=predict(fit_line,
                                    newdata=data.frame(x=newX)
                          ),
                          color="red",
                          linewidth=3,
                          dash=c(10,5),
                          label="LS line",
                          parent=ls,
```

```
                          index="end")

curve.ls3 <- l_layer_line(p,
                          x=newX,
                          y=predict(fit_cubic,
                                    newdata=data.frame(x=newX)
                          ),
                          color="red",
                          linewidth=3,
                          label="LS cubic",
                          parent=ls,
                          index="end")



lms <- l_layer_group(p,
                     label="lmRob",
                     parent=fits,
                     index="end")

curve.lmRob1 <- l_layer_line(p,
                             x=newX,
                             y=predict(fit.lmRob1,
                                       newdata=data.frame(x=newX)
                             ),
                             color="blue",
                             linewidth=3,
                             dash=c(10,5),
                             label="LMS line",
                             parent=lms,
                             index="end")

curve.lmRob3 <- l_layer_line(p,
                             x=newX,
                             y=predict(fit.lmRob3,
                                       newdata=data.frame(x=newX)
                             ),
                             color="blue",
                             linewidth=3,
                             label="LMS cubic",
                             parent=lms,
                             index="end")


lts <- l_layer_group(p,
                     label="Least Trimmed",
                     parent=fits,
                     index="end")

curve.lts1 <- l_layer_line(p,
                           x=newX,
                           y=predict(fit.lts1,
                                     newdata=data.frame(x=newX)
```

```
                           ),
                           color="purple",
                           linewidth=3,
                           dash=c(10,5),
                           label="LTS line",
                           parent=lts,
                           index="end")

curve.lts3 <- l_layer_line(p,
                           x=newX,
                           y=predict(fit.lts3,
                                     newdata=data.frame(x=newX)
                           ),
                           color="purple",
                           linewidth=3,
                           label="LTS cubic",
                           parent=lts,
                           index="end")



#  Plot for the names
#
p_names <- l_plot(title="Methods")
l_layer_text(p_names,
             x=0, y=8,
             text="Least-squares",
             size=20,
             color="red",
             label="Least-squares",
             index="end")
l_layer_text(p_names,
             x=0, y=6,
             text="lmRob",
             size=20,
             color="blue",
             label="Least-squares",
             index="end")
l_layer_text(p_names,
             x=0, y=4,
             text="Least-trimmed-squares",
             size=20,
             color="purple",
             label="Least-squares",
             index="end")
l_scaleto_world(p_names)



updateFits <- function(p, fit1, fit3) {
  ## use only the active points for regression
  sel <- p['active']
```

```
## which coordinates to use for regression
## We'll use the temporary locations if
## someone has moved the points
## otherwise the original coordiates.
##
## For x
xnew <- p['xTemp']
if (length(xnew) == 0) {
  xnew <- p['x']
}
## For y
ynew <- p['yTemp']
if (length(ynew) == 0) {
  ynew <- p['y']
}
##
## Get data
##
newdata <- subset(data.frame(x=xnew, y=ynew), sel)
xrng <- extendrange(xnew)

xvals_curve <- seq(min(xrng), max(xrng), length.out=200 )

## Redo each fit
formula1 <- formula(fit1)
fit.ls1 <- lm(formula1, data=newdata)
fit.lmRob1 <- lmsreg(formula1, data=newdata)
fit.lts1 <- ltsreg(formula1, data=newdata)
## Redo each fit
formula3 <- formula(fit3)
fit.ls3 <- lm(formula3, data=newdata)
fit.lmRob3 <- lmsreg(formula3, data=newdata)
fit.lts3 <- ltsreg(formula3, data=newdata)
##
## update the fitted curves
##
l_configure(curve.ls1,
            x=xvals_curve,
            y=predict(fit.ls1,
                      newdata=data.frame(x=xvals_curve)
            )
)
l_configure(curve.ls3,
            x=xvals_curve,
            y=predict(fit.ls3,
                      newdata=data.frame(x=xvals_curve)
            )
)


l_configure(curve.lmRob1,
            x=xvals_curve,
            y=predict(fit.lmRob1,
```

```r
                    newdata=data.frame(x=xvals_curve)
            )
  )

  l_configure(curve.lmRob3,
            x=xvals_curve,
            y=predict(fit.lmRob3,
                    newdata=data.frame(x=xvals_curve)
            )
  )

  l_configure(curve.lts1,
            x=xvals_curve,
            y=predict(fit.lts1,
                    newdata=data.frame(x=xvals_curve)
            )
  )


  l_configure(curve.lts3,
            x=xvals_curve,
            y=predict(fit.lts3,
                    newdata=data.frame(x=xvals_curve)
            )
  )


  ## Update the tcl language's event handler
  tcl('update', 'idletasks')
}

# Here is we "bind" actions to state changes.
# Whenever the active state, or the temporary locations
# of the points change on the plot p,
# an anonymous function with no arguments will be called.
# This in turn will call the function "updateRegression"
# on the plot p, using the fit fb.fit3
#
l_bind_state(p,
             c("active","xTemp","yTemp"),
             function() {updateFits(p, fit_line, fit_cubic)}
)
```

My experience suggests that `lmRob` is more resistant to outliers and more conservative in estimation. Also I have found cases where `rlm` has failed to converge but `lmRob` has not. I use `lmRob` in the package `qqtest`.

2.4 Power transformations

Outliers in data are sometimes indicative that the data might be better analysed on a transformed scale.

For example, consider the following data set from the `R` package `Sleuth`.

```r
library(loon)
```

```
library(Sleuth3)
data(case0902, package="Sleuth3")
head(case0902)

with(case0902,
     {
     l_plot(Body, Brain,
            itemlabel=paste(Species),
            showItemlabels=TRUE,
            title = "Body and brain weights",
            linkingGroup="Species")
     }
     )
```

In the special case where $x > 0$, one possible "family" of transformations is the power family.

For $x > 0$, we can write this family (over the power α) as

$$T_\alpha(x) = \begin{cases} ax^\alpha + b & (\alpha \neq 0) \\ c\log(x) + d & (\alpha = 0) \end{cases}$$

where a, b, c, d and α are real numbers with $a > 0$ when $\alpha > 0$, $a < 0$ when $\alpha < 0$, and $c > 0$ when $\alpha = 0$. The choices of a, b, c and d are somewhat arbitrary otherwise.

Since location and scale are not important in determining the shape of the distribution, we might settle on particular choices which are more mathematicallly convenient. For example,

$$T_\alpha(x) = \frac{x^\alpha - 1}{\alpha} \quad \forall\ \alpha$$

which requires no separate equation for the $x = 0$ case, since $lim_{\alpha \to 0} T_\alpha(x) = log(x)$.

The function which imlements it does of course require making a special case for $\alpha = 0$:

```
## Power Transformations
## =====================


power <- function(x, y, from=-5, to=5,
                  xlabel="x", ylabel="y",
                  linkingGroup="power", ...) {

  tt <- tktoplevel()
  tktitle(tt) <- "Box-Cox Power Transformation"
  p <- l_plot(x=x, y=y, xlabel=xlabel, ylabel=ylabel,
              parent=tt, linkingGroup=linkingGroup, ...)
  hx <- l_hist(x=x, xlabel=xlabel, yshows='density',
               linkingGroup=linkingGroup)
  hy <- l_hist(x=y, xlabel=ylabel, yshows='density',
               swapAxes=TRUE,
               linkingGroup=linkingGroup)

  lambda_x <- tclVar('1')
  lambda_y <- tclVar('1')
  sx <- tkscale(tt, orient='horizontal',
                variable=lambda_x, from=from, to=to, resolution=0.1)
  sy <- tkscale(tt, orient='vertical',
                variable=lambda_y, from=to, to=from, resolution=0.1)
```

```
  tkgrid(sy, row=0, column=0, sticky="ns")
  tkgrid(p, row=0, column=1, sticky="nswe")
  tkgrid(sx, row=1, column=1, sticky="we")
  tkgrid.columnconfigure(tt, 1, weight=1)
  tkgrid.rowconfigure(tt, 0, weight=1)

  powerfun <- function(x, lambda) {
    if (lambda == 0)
      log(x)
    else
      (x^lambda-1)/lambda
  }

  update <- function(...) {
    l_configure(p,
                x = powerfun(x, as.numeric(tclvalue(lambda_x))),
                y = powerfun(y, as.numeric(tclvalue(lambda_y))))
    l_scaleto_world(p)
    l_configure(hx,
                x = powerfun(x, as.numeric(tclvalue(lambda_x))))
    l_scaleto_world(hx)
    l_configure(hy,
                x = powerfun(y, as.numeric(tclvalue(lambda_y))))
    l_scaleto_world(hy)
  }

  tkconfigure(sx, command=update)
  tkconfigure(sy, command=update)

  invisible(list(plot=p,  histx = hx , histy = hy))
  # only returns value if it is assigned
}



ourPlot <- with(case0902,
                power(Body, Brain,
                      itemlabel=paste(Species),
                      showItemlabels=TRUE,
                      linkingGroup="Species")
                )

# We've kept the handle of the scatterplot for later reference.
ourPlot
```

First let's attach a few fitted lines and explore the effect of removing points.

```
library(robust)
library(MASS)
library(Sleuth3)

data <- case0902

fit_line <- lm(Brain ~ Body, data=data)
```

```
fit.lmRob <- lmRob(formula(fit_line), data=data)
fit.lts <- ltsreg(formula(fit_line), data=data)

#
# Get the plot handle
p <- ourPlot$plot

fits <- l_layer_group(p,
                      label="Various fitting methods",
                      index="end")

# then the individual fits
#
ls <- l_layer_group(p,
                    label="Least Squares",
                    parent=fits,
                    index="end")

curve.ls <- l_layer_line(p,
                         x=data$Body,
                         y=predict(fit_line),
                         color="firebrick",
                         linewidth=4,
                         label="LS line",
                         parent=ls,
                         index="end")


lmRob <- l_layer_group(p,
                        label="lmRob",
                        parent=fits,
                        index="end")

curve.lmRob <- l_layer_line(p,
                             x=data$Body,
                             y=predict(fit.lmRob),
                             color="blue",
                             linewidth=4,
                             label="lmRob line",
                             parent=lmRob,
                             index="end")

lts <- l_layer_group(p,
                      label="Least Trimmed",
                      parent=fits,
                      index="end")

curve.lts <- l_layer_line(p,
                           x=data$Body,
                           y=predict(fit.lts),
                           color="purple",
                           linewidth=4,
                           label="LTS line",
```

```
                            parent=lts,
                            index="end")




updateFits <- function(p, fit) {
  ## use only the active points for regression
  sel <- p['active']
  ## which coordinates to use for regression
  ## We'll use the temporary locations if
  ## someone has moved the points
  ## otherwise the original coordiates.
  ##
  ## For x
  xnew <- p['xTemp']
  if (length(xnew) == 0) {
    xnew <- p['x']
  }
  ## For y
  ynew <- p['yTemp']
  if (length(ynew) == 0) {
    ynew <- p['y']
  }
  ##
  ## Get data
  ##
  newdata <- subset(data.frame(Body=xnew, Brain=ynew), sel)
  ## Redo each fit
  formula <- formula(fit)
  fit.ls <- lm(formula, data=newdata)
  fit.lmRob <- lmRob(formula, data=newdata)
  fit.lts <- ltsreg(formula, data=newdata)
  ##
  ## update the fitted curves
  ##

  selX <- newdata$Body

  l_configure(curve.ls,
              x=selX,
              y=predict(fit.ls)
              )

  l_configure(curve.lmRob,
              x=selX,
              y=predict(fit.lmRob)
              )


  l_configure(curve.lts,
              x=selX,
              y=predict(fit.lts)
```

```
               )

  ## Update the tcl language's event handler
  tcl('update', 'idletasks')
}

# Here is we "bind" actions to state changes.
# Whenever the active state, or the temporary locations
# of the points change on the plot p,
# an anonymous function with no arguments will be called.
# This in turn will call the function "updateRegression"
# on the plot p, using the fit waldo.fit3
#
l_bind_state(p,
             c("active","xTemp","yTemp","x","y"),
             function() {updateFits(p, fit_line)}
             )
```

2.4.0.1 Questions/observations

- What is the effect on the fitted lines of deleting a few points?
  - Do things get better as we zoom in?
- The sliders show the power used in the transformations.
  - What happens as we move the sliders?
  - What is the response in the histograms, as we move the sliders?
  - What is the response in the scatterplot?
- How should we pick the powers?
  - to make a histogram look more symmetric?
  - to straighten the curve of a scatterplot

2.4.1 Tukey's ladder of power transformations

If we would like to have a more symmetric, looking distribution, we need only apply a power transformation.

John Tukey suggested imagining that the powers were arranged in a "ladder" with the smallest powers on the bottom and the largest on the top. For example,

```
#  A convenient mnemonic  ....
#
#  Tukey's "ladder" of power transformations
# ... -2, -1, -1/2, -1/3, 0, 1/3, 1/2, 1, 2, ...
#   <--- down            up -->        ^start
#
#    Or
#
#   Power
# _________
#     2
# _________
#     1      <-- raw data starts here
# _________
#    1/2
# _________
#     0
```

```
# _________
#   -1/2
# _________
#    -1
# _________
#    -2
# _________
#
#  Etc.
```

2.4.2 The bump rules

2.4.2.1 Bump rule 1: Making histograms more symmetric

The rule is that the location of the "bump" in the histogram (where the points are concentrated) tells you which way to "move" on the ladder. If the bump is on "lower" values, then move the power "lower" on the ladder.

Note: Data must be unimodal for the power transformations to have a chance of working.

2.4.2.2 Bump rule 2: Straightening scatterplots

Power transformations not only can be used to make the data distribution more (or less) symmetric, but it can also be used to "straighten" a plot of (x, y) data.

This is done by replacing the original data (x_i, y_i) for $i = 1, \ldots, n$ with the data $(T_{\alpha_x}(x_i), T_{\alpha_y}(y_i))$.

Note that each of the coordinates can have its own power transformation. This amounts to a different α for each variate and so also two different ladders of transformation – the x ladder and the y ladder.

Power transformations can be used to "straighten bivariate relationships" and to make univariate densities more symmetric. The location of the "bump" tells you which way to go on the ladder of transformations.

2.5 Selecting a subset

Another reason to use zero and one weights at least is to select a subset on which the fit is to be built.

For example, in conducting the interactive analysis on the `like` versus `Impressions` data we noticed that those posts which were of `Category` "Action" seemed to have a different relation than did those of either `Category` "Product" or "Inspiration". We might choose to fit separate functions $\mu_(x)$ for each category.

We could use weights to proceed as follows:

```
#
N <- length(fb$x)
# get  different weight vectors, initialized to be zeros
w_A <- rep(0, N)   ## Action
w_O <- rep(0, N)   ## Other (Not Action)
w_P <- rep(0, N)   ## Product
w_I <- rep(0, N)   ## Inspiration
#
# And we'll give a weight of one for pages belonging
# to each book in turn
w_A[fb$Category == "Action"] <- 1
w_O[fb$Category != "Action"] <- 1
w_P[fb$Category == "Product"] <- 1
w_I[fb$Category == "Inspiration"] <- 1
```

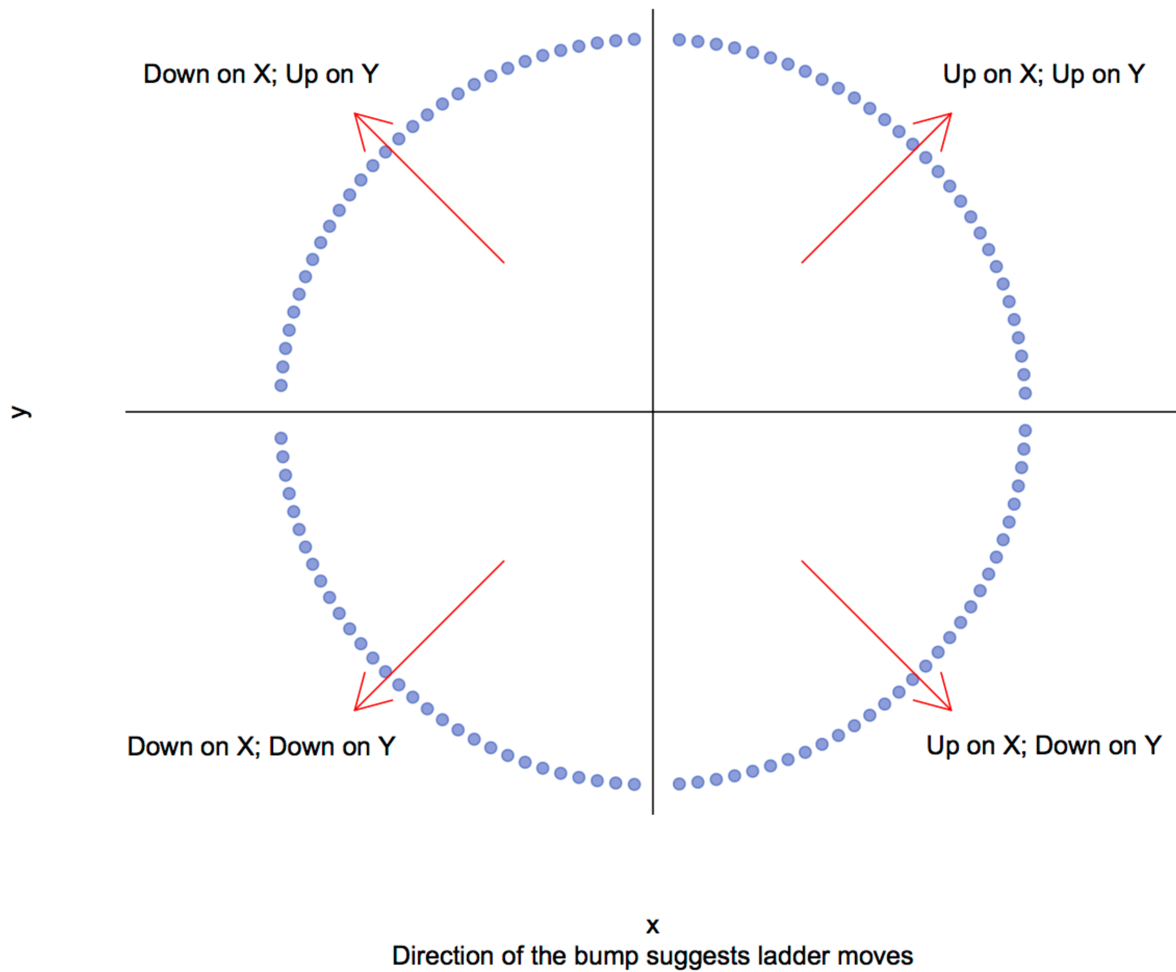


Figure 2: Rule of the bump

```r
# Number of "Action" posts are
sum(w_A)
```

```
## [1] 211
```

```r
# Number of other (non-"Action") posts are
sum(w_O)
```

```
## [1] 284
```

```r
# Number of "Product" posts are
sum(w_P)
```

```
## [1] 129
```

```r
# Number of "Inspiration" posts are
sum(w_I)
```

```
## [1] 155
```

We now call `lm(...)` to fit our model with these weights.

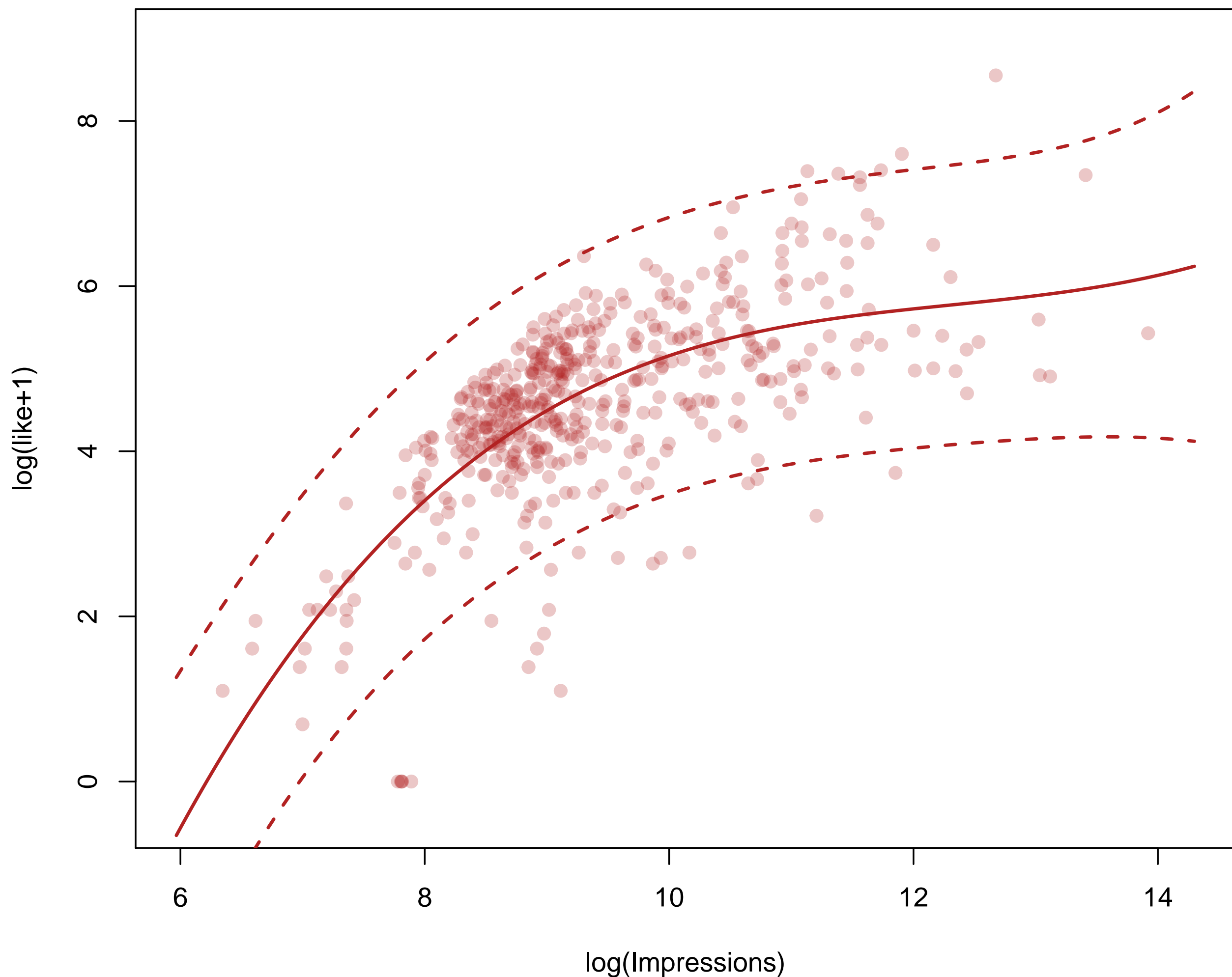
```r
# We'll work the same model we did for our last fit
# but now with weights according to which Coryateg
formula <- formula(facebook.fit3)

fitAction <-   lm(formula, weights=w_A, data=fb )
fitOther <-  lm(formula, weights=w_O, data=fb )
fitProduct <- lm(formula, weights=w_P, data=fb )
fitInspiration <- lm(formula, weights=w_I, data=fb )

xlim  <- extendrange(fb$x)
newX <- seq(min(xlim), max(xlim), length.out=200)
#
# Get 95% prediction intervals for each group
pred95.All <- predict(facebook.fit3,
                         newdata=data.frame(x=newX),
                         interval="prediction",
                         level=0.95)
pred95.Action <- predict(fitAction,
                         newdata=data.frame(x=newX),
                         interval="prediction",
                         level=0.95)

pred95.Other <- predict(fitOther,
                        newdata=data.frame(x=newX),
                        interval="prediction",
                        level=0.95)

pred95.Product <- predict(fitProduct,
                          newdata=data.frame(x=newX),
                          interval="prediction",
                          level=0.95)

pred95.Inspiration <- predict(fitInspiration,
                              newdata=data.frame(x=newX),
                              interval="prediction",
```

```
                              level=0.95)
#
# And we can plot them
#
# create a function that we can call
plotfacebook <- function(intervals, col="black", category,
                         prediction=TRUE,
                         title=NULL, add=FALSE) {
  # Get the selection
  if (category=="All") {
    sel <- rep(TRUE, length(fb$x))

  } else
  {if (category=="Other")
    {
    sel <- fb$Category != "Action"
    }
  else  sel <- fb$Category == category
  }

  if(is.null(title)){
    title <-  paste(category,
                    "posts ( ",
                    sum(sel),
                    "obs )"
    )
  }
  # Get the data
  x <- fb$x[sel]
  y <- fb$y[sel]

  if (!add){
    plot(x,y,
         xlim = extendrange(fb$x), ylim = extendrange(fb$y),
         main = title,
         xlab = "log(Impressions)",
         ylab = "log(like+1)",
         pch=19,
         col=adjustcolor("firebrick", 0.25)
    )
  } else
  {
    points(x, y,
           pch=19,
           col=adjustcolor(col, 0.25))
  }
  lines(newX,
        intervals[,"fit"],
        col=col,
        lwd=2)
  if(prediction){
  lines(newX,
        intervals[,"lwr"],
```

```
         lty=2,
         col=col,
         lwd=2)
  lines(newX,
         intervals[,"upr"],
         lty=2,
         col=col,
         lwd=2)}
  }
```

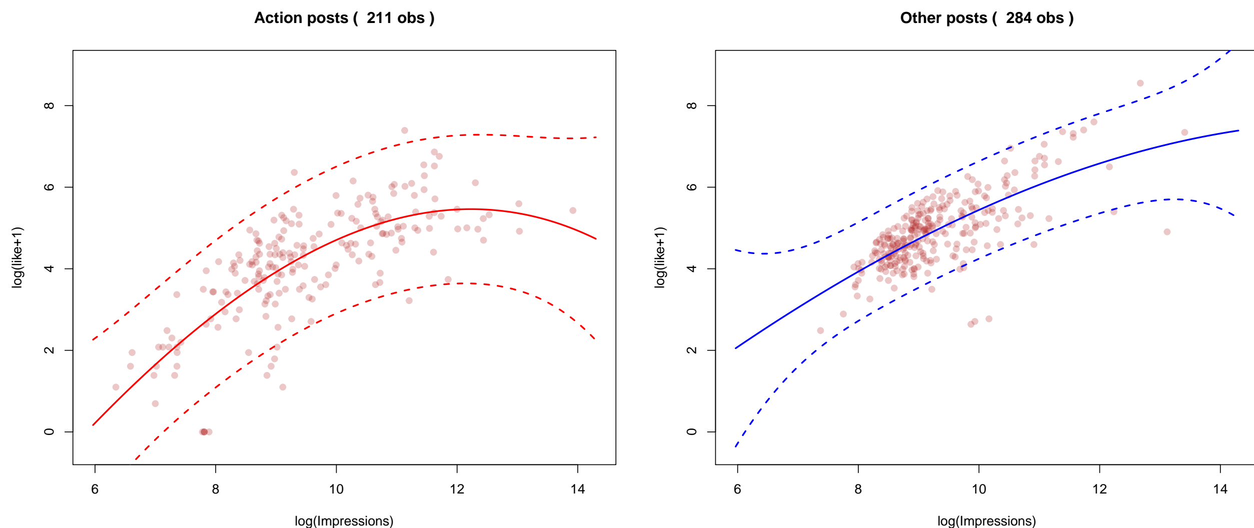
We can use the above function to construct a number of plots. First, let's look at the least-squares fit and its 95% prediction intervals for all posts.

```
#
# And now for all of the posts we plot the 95% prediction interval
#
plotfacebook(pred95.All, "firebrick", category="All", add=FALSE)
```



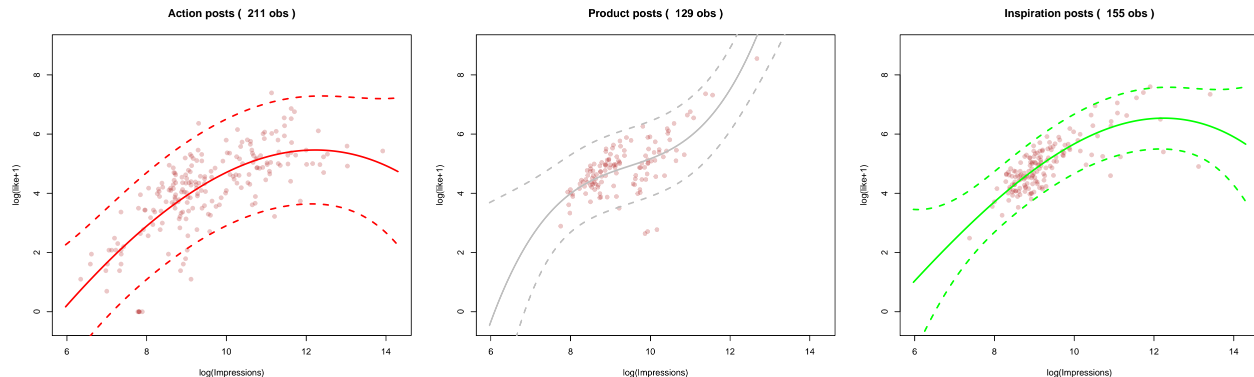
Comparing "Action" posts to all others:

```
savePar <- par(mfrow=c(1,2))
plotfacebook(pred95.Action, "red", category="Action", add=FALSE)
plotfacebook(pred95.Other, "blue", category="Other", add=FALSE)
```




```
par(savePar)
```

```
savePar <- par(mfrow=c(1,3))
plotfacebook(pred95.Action, "red", category="Action", add=FALSE)
plotfacebook(pred95.Product, "grey", category="Product", add=FALSE)
plotfacebook(pred95.Inspiration, "green", category="Inspiration", add=FALSE)
```

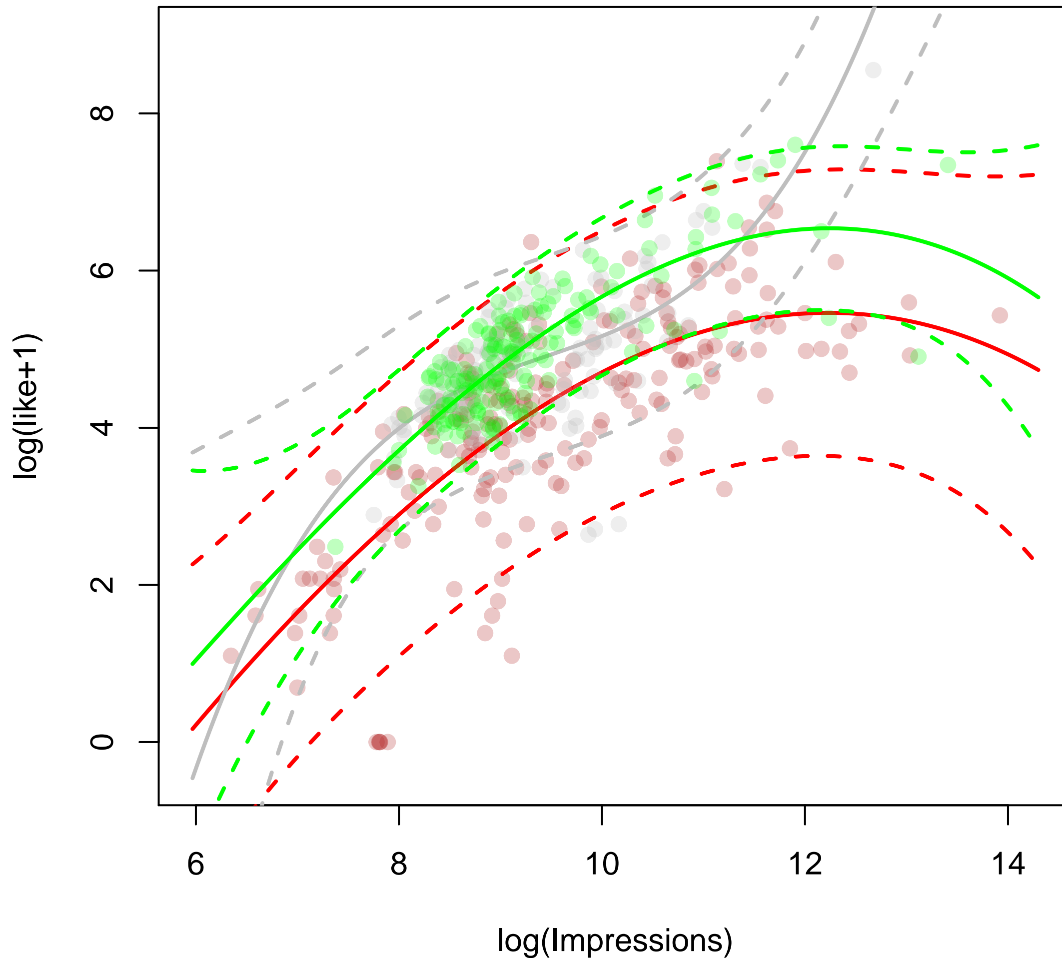



```
par(savePar)
```

Based on these predictions, we might ask how well they are doing? How do the fitted models differ in their interpretations? How might you assess whether the models agree?

We might on occasion consider overlaying the various predictions on the same plot.

```
#
# And we can plot them on the same plot
#
plotfacebook(pred95.Action, "red", category="Action", add=FALSE, title ="Each separate category")
plotfacebook(pred95.Product, "grey", category="Product", add=TRUE)
plotfacebook(pred95.Inspiration, "green", category="Inspiration", add=TRUE)
```



2.5.1 Nearest neighbours

Rather than select a sample of points (e.g. all posts that are "Product", or "Inspiration", or some other selection), we might be interested in only using points that are nearby (e.g. distance in the **x** space).

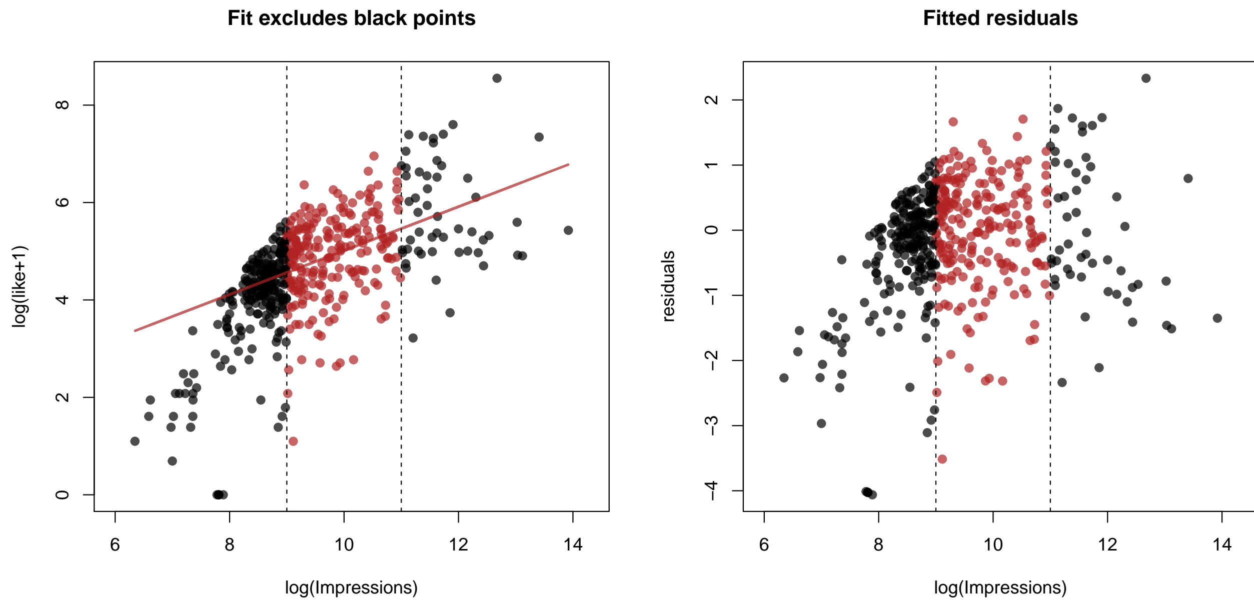
For example, suppose we were interested in predicting the number of likes for middling numbers of impressions? Then we might use only those values that had x values between say x_{low} and x_{high}, say. The fit could be had as

```
# Bounds of the neighbourhood:
xlow <- 9
xhigh <- 11


points <- {xlow <= fb$x} & {fb$x <= xhigh}
head(points)
```

```
## [1] FALSE  TRUE FALSE FALSE  TRUE  TRUE
```

```
# We can select these points via weights
w <- rep(0, N)
```

```r
w[points] <- 1


# Let's just fit a straight line
facebook.middle.w <- lm(y ~ x, data=fb, weights=w)

# alternatively, we could have done this with subsetting
facebook.middle.s <- lm(y ~ x, data=fb, subset = points)

#
# Get colours
col <- rep("black", N)
col[points] <- "firebrick"


# layout of plots in a row
parOptions <- par(mfrow=c(1,2))

# First for all points
#
#
plot(fb$x, fb$y,
     xlim = xlim,
     main = "Fit excludes black points",
     xlab = "log(Impressions)",
     ylab = "log(like+1)",
     pch=19,
     col=adjustcolor(col, 0.7)
     )

lines(fb$x, predict(facebook.middle.w),
      lwd=2,
      col=adjustcolor("firebrick", 0.7)
      )

# The neighbourhood is between the dashed vertical lines
abline(v=xlow, lwd=1, lty=2)
abline(v=xhigh, lwd=1, lty=2)

plot(fb$x, residuals(facebook.middle.w),
     xlim = xlim,
     main = "Fitted residuals",
     xlab = "log(Impressions)",
     ylab = "residuals",
     pch=19,
     col=adjustcolor(col, 0.7)
     )

# Again, the neighbourhood is between the dashed vertical lines
abline(v=xlow, lwd=1, lty=2)
abline(v=xhigh, lwd=1, lty=2)
```




```
# Set the layout options back
par(parOptions)
```

Clearly, this line fits best on the neighbourhood on which it is based, and not so well (as it turns out) elsewhere.

This is neighbourhood is contains all points within a region of $x = 10$. That is, it is the set of all points x_i such that $|x_i - x| \le \delta$ say, where in our example $x = 10$ and $\delta = 1$. If instead of scalar x_i s, we had vector $\mathbf{x}_i \in \mathbb{R}^p$, say, then a **Euclidean distance neighbourhood** of the p-dimensional point $\mathbf{x}$ would be

$$N_\delta(\mathbf{x}) = \{\mathbf{x}_i \mid \quad d(\mathbf{x}, \mathbf{x}_i) \le \delta, \forall i = 1, \ldots, N\}$$

where $d(\mathbf{x}, \mathbf{x}_i) = ||\mathbf{x} - \mathbf{x}_i||$ is the Euclidean distance between $\mathbf{x}$ and $\mathbf{x}_i$. That is, the Euclidean distance neighbourhood $N_b(\mathbf{x})$ is the set of all data points $\mathbf{x}_i$ lying within a ball of radius δ centred at $\mathbf{x}$.

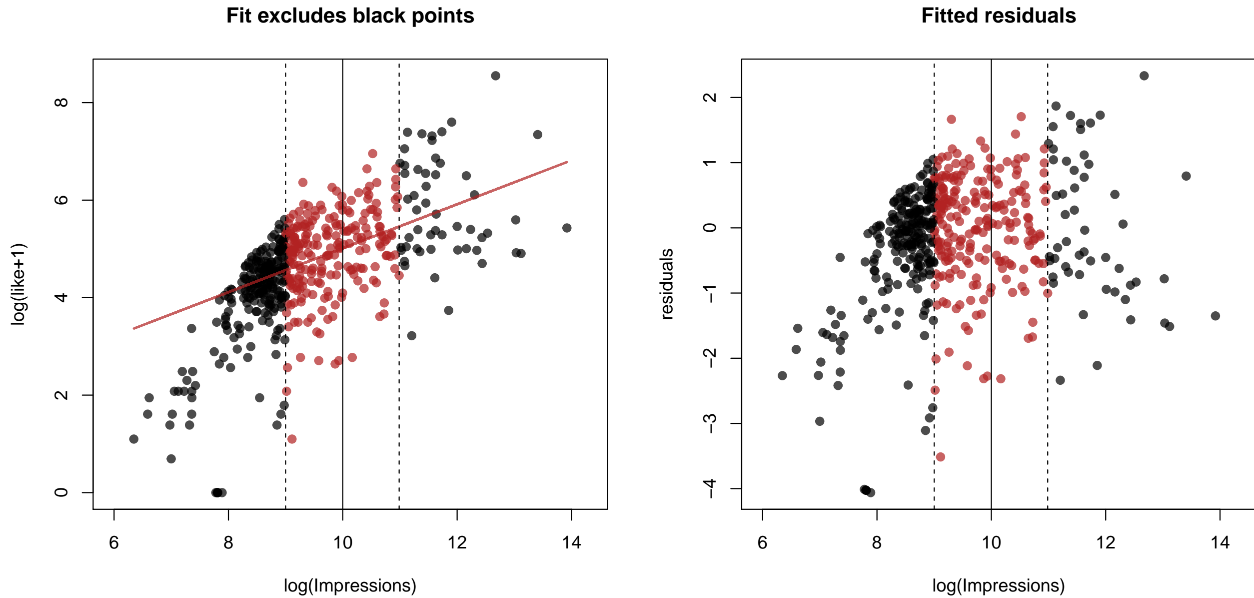
Another way of defining a neighbourhood would be to just expand the radius of this ball until it included k data points. This would be a k **nearest neighbourhood** of a point $\mathbf{x}$. Where the Euclidean distance neighbouthood could include anywhere from 0 to N points, the k nearest neighbourhood always includes exactly k points. So what is fixed is no longer δ but k.

In R, we can find any number of nearest neighbours to a point x in a given data set. As with the Euclidean distance

```
library("FNN")

# The FNN library has a number of Fast Nearest Neighbour search
# algorithms.

x <- 10
# Let's take k to be the same number of points
# that our Euclidean distance returned at x=10
#
k <- sum(points)

neighbours <- get.knnx(fb$x, query=x, k=k)$nn.index

# We can select these points via weights
w <- rep(0, N)
```

```r
w[neighbours] <- 1


# Let's just fit a straight line
facebook.kNN.w <- lm(y ~ x, data=fb, weights=w)

# alternatively, we could have done this with subsetting
facebook.kNN.s <- lm(y ~ x, data=fb, subset = neighbours)

#
# Get colours
col <- rep("black", N)
col[neighbours] <- "firebrick"


# layout of plots in a row
parOptions <- par(mfrow=c(1,2))

# First for all points
#
#
plot(fb$x, fb$y,
     xlim = xlim,
     main = "Fit excludes black points",
     xlab = "log(Impressions)",
     ylab = "log(like+1)",
     pch=19,
     col=adjustcolor(col, 0.7)
     )

lines(fb$x, predict(facebook.kNN.w),
      lwd=2,
      col=adjustcolor("firebrick", 0.7)
      )

# The neighbourhood is between the dashed vertical lines
abline(v=min(fb$x[neighbours]), lwd=1, lty=2)
abline(v=x, lwd=1, lty=1)
abline(v=max(fb$x[neighbours]), lwd=1, lty=2)

plot(fb$x, residuals(facebook.kNN.w),
     xlim = xlim,
     main = "Fitted residuals",
     xlab = "log(Impressions)",
     ylab = "residuals",
     pch=19,
     col=adjustcolor(col, 0.7)
     )

# Again, the neighbourhood is between the dashed vertical lines
# The neighbourhood is between the dashed vertical lines
abline(v=min(fb$x[neighbours]), lwd=1, lty=2)
abline(v=x, lwd=1, lty=1)
```

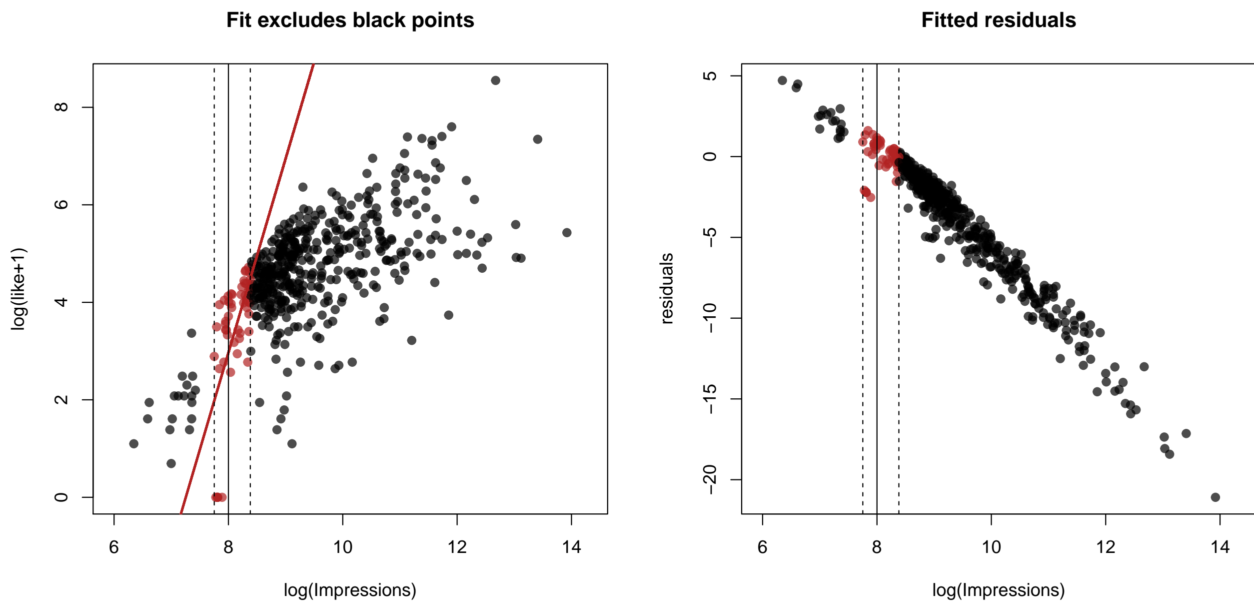
```r
abline(v=max(fb$x[neighbours]), lwd=1, lty=2)
```

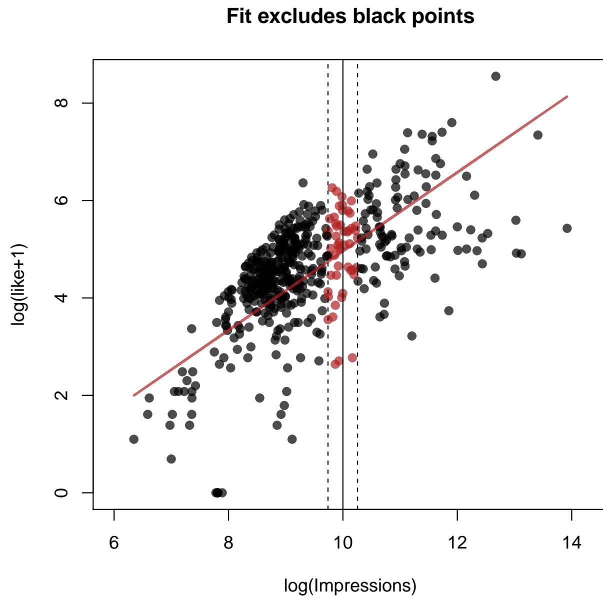


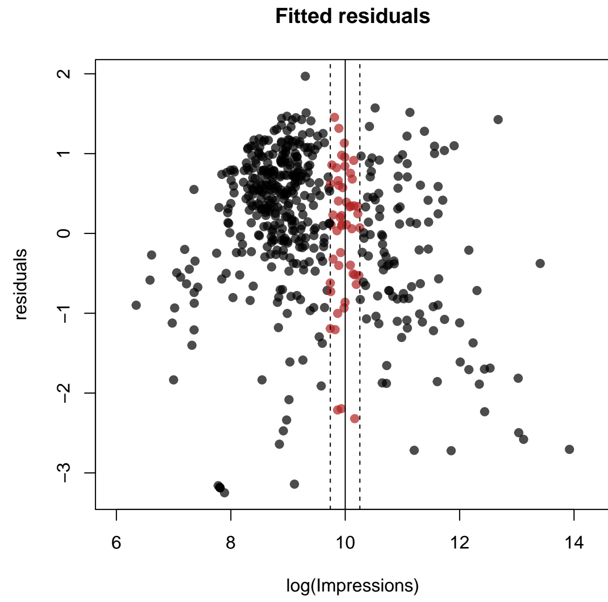

```r
# Set the layout options back
par(parOptions)
```

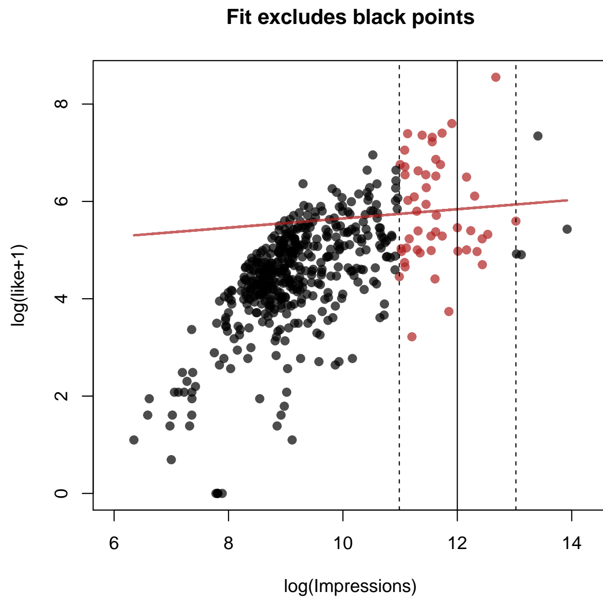
Note that the method returns the same points but that the neighbourhood regions are defined differently. We have just chosen the radius $\delta = 1$ and number of neighbours $k = 225$ to match when $x = 10$. Should we pick another value of x, the Euclidean ball may contain more or fewer points than 225.

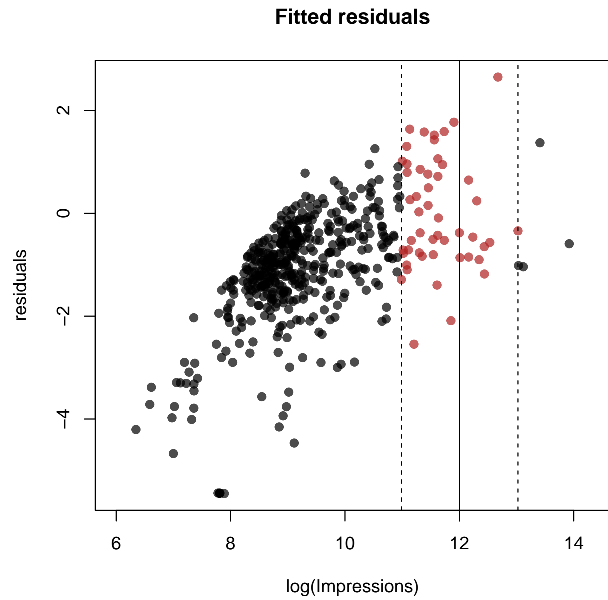
We might try to have a local fit that depended only on a local percentage of the data, say 10%, or $k \approx 50$. Here are the results at a few different locations.











Note that the local fitting really is **only** fitting the data **locally**. Its value is only reliable locally. Indeed, the fit is at a **single specified location**.

Again, *local* here means **local in x**. This removes the influence of far away **x** values on the fit at the specified point.

This kind of local fitting will be useful later. In combination with robust regression methods the effect of both outlying y and **x** can be substantially ameliorated by local robust fits.