# Fitting Models to Data

Statistical Computing, STA3005

Wednesday Mar 27, 2024

## Last chapter: Simulation

- Running simulations is an integral part of being a statistician in the 21st century
- R provides us with built-in functions for simulations from a wide variety of distributions
- To make your simulation results reproducible, you must set the seed, using `set.seed()`
- Inverse method generates random numbers based on a sequence of uniform random numbers
- There is a natural connection between iteration, functions, and simulations
- Saving and loading results can be done in two formats: rds and rdata formats

## Part I: *Reading in and writing out data*

## Reading in data from the outside

Before analyzing data, we first introduce how to read in data from the outside and write out to files, using:

- `readLines()` : reading in lines of text from a file or webpage; returns vector of strings
- `read.table()` : read in a data table from a file or webpage; returns data frame
- `read.csv()` : like the above, but assumes comma separated data; returns data frame

## Reading in data from a previous R session

Here are the ways to read/write in data in specialized R formats:

- `readRDS()` , `saveRDS()` : functions for reading/writing single R objects from/to a file
- `load()` , `save()` : functions for reading/writing any number of R objects from/to a file

直接 save 成 R object

- `save.image()` : functions for saving the current environment to a file
- Advantage: these can be a lot more memory efficient than storing in `txt` or `csv` files.
- Disadvantage: they're limited in scope in the sense that they can only communicate with R

## `read.table()` and `read.csv()`

Have a table full of data, just not in the R file format?

Then `read.table()` is the function for you. It works as in:

- `read.table(file=file.name, sep=" ")` , to read data from a local file on your computer called `file.name` , assuming (say) space separated data
- `read.table(file=webpage.link, sep="\t")` , to read data from a webpage up at `webpage.link` , assuming (say) tab separated data

The function `read.csv()` is just a shortcut for using `read.table()` with `sep=","` . (But note: these two actually differ on some of the default inputs!)

一般还需要设置 header = TRUE

## Examples of reading in data

注：sep="" 相当于 sep=" " 或 "\t"

```r
# This data table is comma separated, so we can use read.csv()
strike.df = read.csv("strikes.csv")
head(strike.df)
```

```
##     country year strike.volume unemployment inflation left.parliament
## 1 Australia 1951           296          1.3      19.8            43.0
## 2 Australia 1952           397          2.2      17.2            43.0
## 3 Australia 1953           360          2.5       4.3            43.0
## 4 Australia 1954             3          1.7       0.7            47.0
## 5 Australia 1955           326          1.4       2.0            38.5
## 6 Australia 1956           352          1.8       6.3            38.5
##   centralization density
## 1      0.3748588      NA
## 2      0.3751829      NA
## 3      0.3745076      NA
## 4      0.3710170      NA
## 5      0.3752675      NA
## 6      0.3716072      NA
```

```r
sapply(strike.df, class)
```

```
##         country            year   strike.volume    unemployment       inflation
##     "character"       "integer"       "integer"       "numeric"       "numeric"
## left.parliament  centralization         density
##       "numeric"       "numeric"       "numeric"
```

```r
# This data table is tab separated, so let's specify sep="\t"
anss.df = read.table(file="anss.dat", sep="\t")
head(anss.df)
```

```
##           V1          V2       V3        V4     V5   V6
## 1       Date        Time      Lat       Lon  Depth  Mag
## 2 2002/01/01 10:39:06.82 -55.2140 -129.0000  10.00 6.00
## 3 2002/01/01 11:29:22.73   6.3030  125.6500 138.10 6.30
## 4 2002/01/02 14:50:33.49 -17.9830  178.7440 665.80 6.20
## 5 2002/01/02 17:22:48.76 -17.6000  167.8560  21.00 7.20
## 6 2002/01/03 07:05:27.67  36.0880   70.6870 129.30 6.20
```

```r
sapply(anss.df, class)
```

```
##          V1          V2          V3          V4          V5          V6
## "character" "character" "character" "character" "character" "character"
```

```r
# Oops! It comes with column names, so let's set header=TRUE
anss.df = read.table(file="anss.dat", sep="\t",
                     header=TRUE)
head(anss.df)
```

```
##         Date        Time     Lat      Lon Depth Mag
## 1 2002/01/01 10:39:06.82 -55.214 -129.000  10.0 6.0
## 2 2002/01/01 11:29:22.73   6.303  125.650 138.1 6.3
## 3 2002/01/02 14:50:33.49 -17.983  178.744 665.8 6.2
## 4 2002/01/02 17:22:48.76 -17.600  167.856  21.0 7.2
## 5 2002/01/03 07:05:27.67  36.088   70.687 129.3 6.2
## 6 2002/01/03 10:17:36.30 -17.664  168.004  10.0 6.6
```

```
sapply(anss.df, class)
```

```
##       Date       Time        Lat        Lon      Depth        Mag
## "character" "character"  "numeric"  "numeric"  "numeric"  "numeric"
```

## Helpful input arguments

The above inputs apply to either `read.table()` or `read.csv()` (though these two functions actually have different default inputs in general—e.g., `header` defaults to TRUE in `read.csv()` but FALSE in `read.table()`)

- `header`: Boolean, TRUE means that the first line should be interpreted as column names
- `sep`: string, specifies what separates the entries; empty string "" means any white space
- `quote`: string, specifies what set of characters signify the beginning and end of quotes; empty string "" disables quotes altogether

Other helpful inputs: `skip`, `row.names`, `col.names`. You can read about them in the help file for `read.table()`

## `write.table()` and `write.csv()`

To write a data frame (or matrix) to a text file, use `write.table()` or `write.csv()`. These are the counterparts to `read.table()` and `read.csv()`, and they work as in:

- `write.table(my.df, file="my.df.txt", sep=" ", quote=FALSE)`, to write `my.dat` to the text file "my.df.txt" (to be created in your working directory), with (say) space separation, and no quotes around the entries
- `write.csv(my.df, file="my.df.csv", quote=FALSE)`, to write `my.dat` to the text file "my.df.csv" (to be created in your working directory), with comma separation, and no quotes

Note that `quote=FALSE`, signifying that no quotes should be put around the printed data entries, seems always preferable (default is `quote=TRUE`). Also, setting `row.names=FALSE` and `col.names=FALSE` will disable the printing of row and column names (defaults are `row.names=TRUE` and `col.names=TRUE`)

# Part II: *Exploratory data analysis*

## Why fit statistical (regression) models?

You have some data $X_1, \ldots, X_p, Y$: the variables $X_1, \ldots, X_p$ are called predictors, and $Y$ is called a response. You're interested in the relationship that governs them

So you posit that $Y|X_1, \ldots, X_p \sim P_\theta$, where $\theta$ represents some unknown parameters. This is called **regression model** for $Y$ given $X_1, \ldots, X_p$. Goal is to estimate parameters.

- **inference**: to assess model validity, predictor importance
- **prediction**: to predict future $Y$'s from future $X_1, \ldots, X_p$'s

## Prostate cancer data

Recall the data set on 97 men who have prostate cancer. The measured variables:

- `lpsa`: log Prostate-Specific Antigen (PSA) score, often elevated in people with prostate cancer
- `lcavol`: log cancer volume
- `lweight`: log prostate weight
- `age`: age of patient
- `lbph`: log of the amount of benign prostatic hyperplasia
- `svi`: seminal vesicle invasion

- `lcp` : log of capsular penetration
- `gleason` : Gleason score
- `pgg45` : percent of Gleason scores 4 or 5

```
pros.df = read.table("pros.dat")
dim(pros.df)
```

```
## [1] 97  9
```

```
head(pros.df)
```

```
##       lcavol  lweight age      lbph svi       lcp gleason pgg45       lpsa
## 1 -0.5798185 2.769459  50 -1.386294   0 -1.386294       6     0 -0.4307829
## 2 -0.9942523 3.319626  58 -1.386294   0 -1.386294       6     0 -0.1625189
## 3 -0.5108256 2.691243  74 -1.386294   0 -1.386294       7    20 -0.1625189
## 4 -1.2039728 3.282789  58 -1.386294   0 -1.386294       6     0 -0.1625189
## 5  0.7514161 3.432373  62 -1.386294   0 -1.386294       6     0  0.3715636
## 6 -1.0498221 3.228826  50 -1.386294   0 -1.386294       6     0  0.7654678
```

Some example questions we might be interested in:

- What is the relationship between `lcavol` and `lweight`?
- What is the relationship between `svi` and `lcavol`, `lweight`?
- Can we predict `lpsa` from the other variables?
- Can we predict whether `lpsa` is high or low, from other variables?

## Exploratory data analysis

Before pursuing a specific model, it's generally a good idea to look at your data. When done in a structured way, this is called **exploratory data analysis**. E.g., you might investigate:

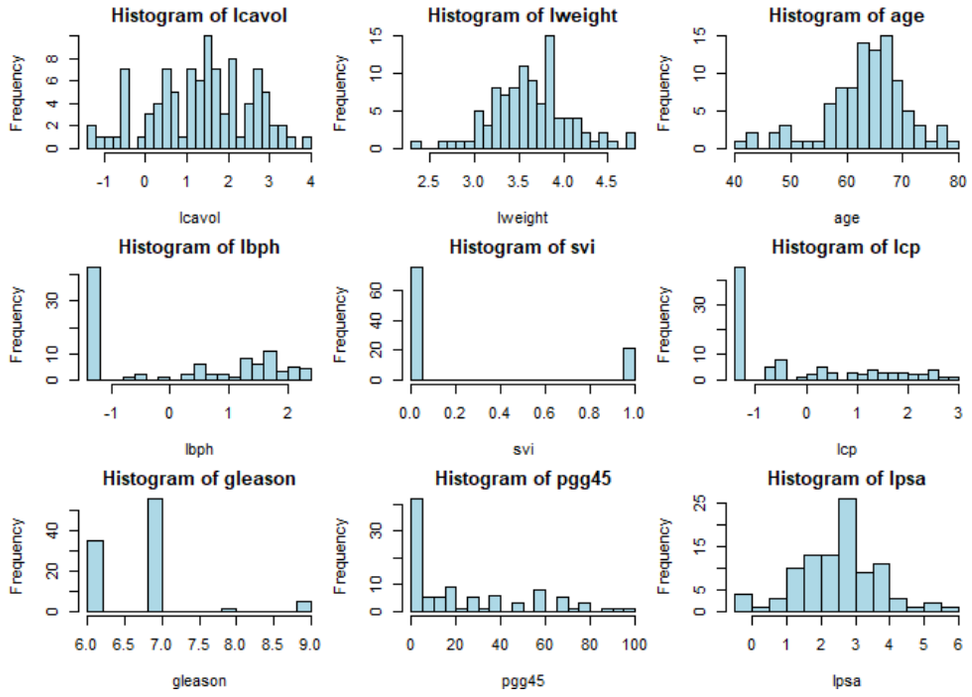
- What are the distributions of the variables?
- Are there distinct subgroups of samples?
- Are there any noticeable outliers?
- Are there interesting relationship/trends to model?

## Distributions of prostate cancer variables

```
colnames(pros.df) # These are the variables
```

```
## [1] "lcavol"  "lweight" "age"     "lbph"    "svi"     "lcp"     "gleason"
## [8] "pgg45"   "lpsa"
```

```
par(mfrow=c(3,3), mar=c(4,4,2,0.5)) # Setup grid, margins
for (j in 1:ncol(pros.df)) {
  hist(pros.df[,j], xlab=colnames(pros.df)[j],
       main=paste("Histogram of", colnames(pros.df)[j]),
       col="lightblue", breaks=20)
}
```




What did we learn? A bunch of things! E.g.,

- `svi`, the presence of seminal vesicle invasion, is binary
- `lcp`, the log amount of capsular penetration, is very skewed, a bunch of men with little (or none?), then a big spread; why is this?
- `gleason`, takes integer values of 6 and larger; how does it relate to `pgg45`, the percentage of Gleason scores 4 or 5?
- `lpsa`, the log PSA score, is close to normally distributed

After asking doctors some questions, we learn:

- When the actual capsular penetration is very small, it can't be properly measured, so it just gets arbitrarily set to 0.25 (and we can check that `min(pros.df$lcp)` $\approx \log 0.25$)
- The variable `pgg45` measures the percentage of 4 or 5 Gleason scores that were recorded over their visit history *before* their final current Gleason score, stored in `gleason`; a higher Gleason score is worse, so `pgg45` tells us something about the severity of their cancer in the past

## Correlations between prostate cancer variables

```
pros.cor = cor(pros.df)
round(pros.cor,3)
```

```
##         lcavol lweight   age   lbph    svi    lcp gleason pgg45  lpsa
## lcavol   1.000   0.281 0.225  0.027  0.539  0.675   0.432 0.434 0.734
## lweight  0.281   1.000 0.348  0.442  0.155  0.165   0.057 0.107 0.433
## age      0.225   0.348 1.000  0.350  0.118  0.128   0.269 0.276 0.170
## lbph     0.027   0.442 0.350  1.000 -0.086 -0.007   0.078 0.078 0.180
## svi      0.539   0.155 0.118 -0.086  1.000  0.673   0.320 0.458 0.566
## lcp      0.675   0.165 0.128 -0.007  0.673  1.000   0.515 0.632 0.549
## gleason  0.432   0.057 0.269  0.078  0.320  0.515   1.000 0.752 0.369
## pgg45    0.434   0.107 0.276  0.078  0.458  0.632   0.752 1.000 0.422
## lpsa     0.734   0.433 0.170  0.180  0.566  0.549   0.369 0.422 1.000
```

Some strong correlations! Let's find the biggest (in absolute value):

```
pros.cor[lower.tri(pros.cor,diag=TRUE)] = 0 # Why only upper triangle part?
pros.cor.sorted = sort(abs(pros.cor),decreasing=T)
pros.cor.sorted[1]
```

```
## [1] 0.7519045
```

which 作用于 underlying vector，返回 vector 中的 index

```
vars.big.cor = arrayInd(which(abs(pros.cor)==pros.cor.sorted[1]),
                        dim(pros.cor)) # Note: arrayInd() is useful
colnames(pros.df)[vars.big.cor]
```

```
## [1] "gleason" "pgg45"
```

This is not surprising, given what we know about `pgg45` and `gleason`; essentially this is saying: if their Gleason score is high now, then they likely had a bad history of Gleason scores

Let's find the second biggest correlation (in absolute value):

```
pros.cor.sorted[2]
```

```
## [1] 0.7344603
```

```
vars.big.cor = arrayInd(which(abs(pros.cor)==pros.cor.sorted[2]),
                        dim(pros.cor))
colnames(pros.df)[vars.big.cor]
```

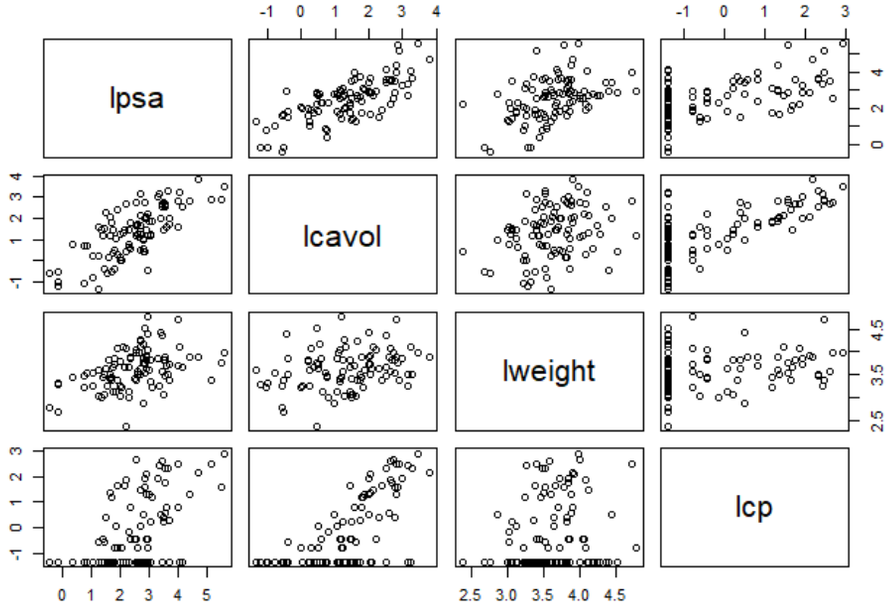
```
## [1] "lcavol" "lpsa"
```

This is more interesting! If we want to predict `lpsa` from the other variables, then it seems like we should at least include `lcavol` as a predictor

## Visualizing relationships among variables, with `pairs()`

Can easily look at multiple scatter plots at once, using the `pairs()` function. The first argument is written like a **formula**, with no response variable. We'll hold off on describing more about formulas until we learn `lm()`, shortly

```
pairs(~ lpsa + lcavol + lweight + lcp, data=pros.df)
```



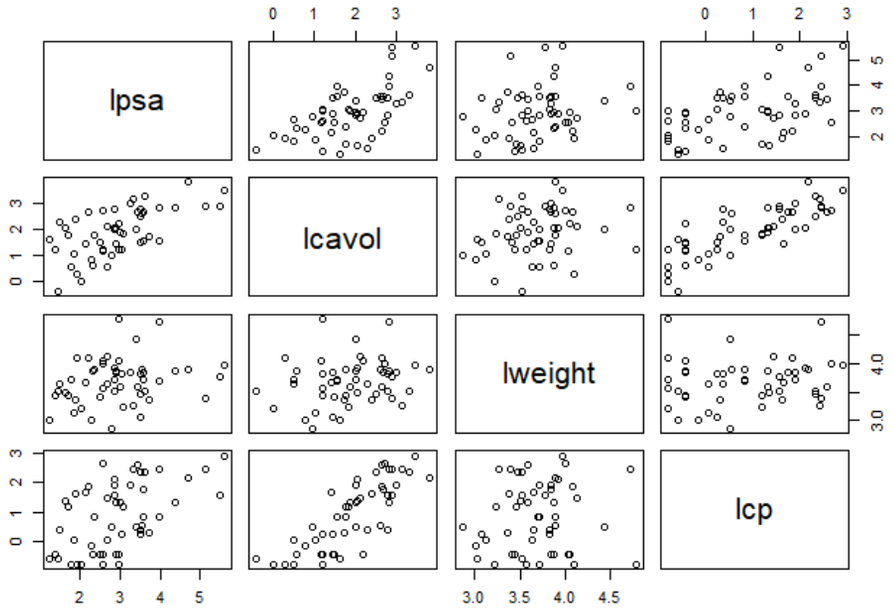
## Inspecting relationships over a subset of the observations

As we've seen, the `lcp` takes a bunch of really low values, that don't appear to have strong relationships with other variables. Let's get rid of them and see what the relationships look like

```
pros.df.subset = pros.df[pros.df$lcp > min(pros.df$lcp),]
nrow(pros.df.subset) # Beware, we've lost a half of our data!
```

```
## [1] 52
```

```
pairs(~ lpsa + lcavol + lweight + lcp, data=pros.df.subset)
```




## Testing means between two different groups

Recall that `svi`, the presence of seminal vesicle invasion, is binary:

```
table(pros.df$svi)
```

```
##
##  0  1
## 76 21
```

From http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1476128/:

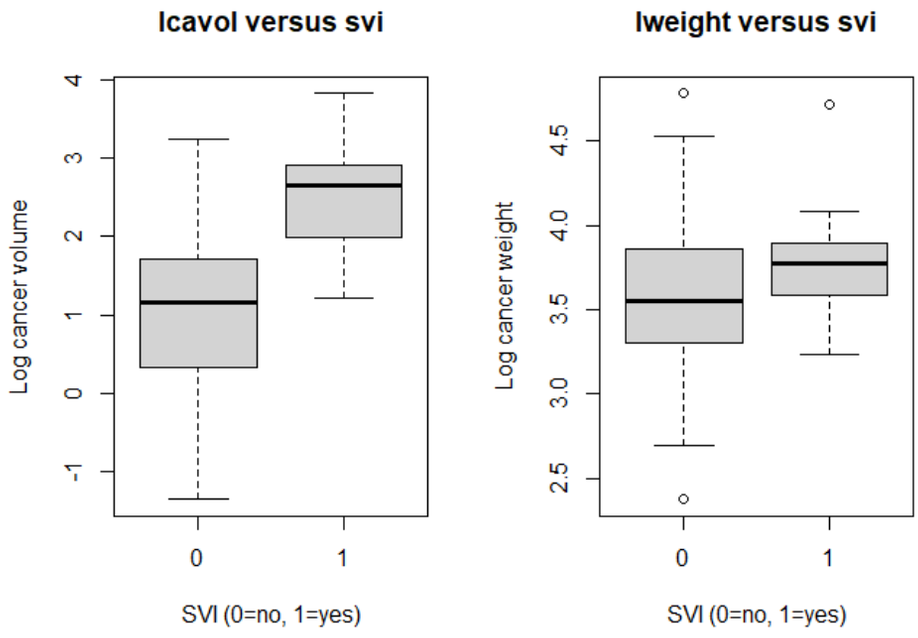
> "When the pathologist's report following radical prostatectomy describes seminal vesicle invasion (SVI) ... prostate cancer in the areolar connective tissue around the seminal vesicles and outside the prostate ...generally the outlook for the patient is poor."

Does seminal vesicle invasion relate to the volume of cancer? Weight of cancer?

Let's do some plotting first:

```
pros.df$svi = factor(pros.df$svi)
par(mfrow=c(1,2))
plot(pros.df$svi, pros.df$lcavol, main="lcavol versus svi",
     xlab="SVI (0=no, 1=yes)", ylab="Log cancer volume")
```

```
plot(pros.df$svi, pros.df$lweight, main="lweight versus svi",
     xlab="SVI (0=no, 1=yes)", ylab="Log cancer weight")
```




Visually, `lcavol` looks like it has a big difference, but `lweight` perhaps does not

Now let's try simple two-sample t-tests:

```
t.test(pros.df$lcavol[pros.df$svi==0],
       pros.df$lcavol[pros.df$svi==1])
```

```
##
##  Welch Two Sample t-test
##
## data:  pros.df$lcavol[pros.df$svi == 0] and pros.df$lcavol[pros.df$svi == 1]
## t = -8.0351, df = 51.172, p-value = 1.251e-10
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.917326 -1.150810
## sample estimates:
## mean of x mean of y
##  1.017892  2.551959
```

```
t.test(pros.df$lweight[pros.df$svi==0],
       pros.df$lweight[pros.df$svi==1])
```

```
##
##  Welch Two Sample t-test
##
## data:  pros.df$lweight[pros.df$svi == 0] and pros.df$lweight[pros.df$svi == 1]
## t = -1.8266, df = 42.949, p-value = 0.07472
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.33833495  0.01674335
## sample estimates:
## mean of x mean of y
##  3.594131  3.754927
```

Confirms what we saw visually

# Part III: *Linear models*

## Linear regression modeling

The linear model is arguably the **most widely used** statistical model, has a place in nearly every application domain of statistics

Given response $Y$ and predictors $X_1, \ldots, X_p$, in a **linear regression model**, we posit:

$$Y = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p + \epsilon, \quad \text{where } \epsilon \sim N(0, \sigma^2)$$

Goal is to estimate parameters, also called coefficients $\beta_0, \beta_1, \ldots, \beta_p$

## Fitting a linear regression model with `lm()`

We can use `lm()` to fit a linear regression model. The first argument is a **formula**, of the form `Y ~ X1 + X2 + ... + Xp`, where `Y` is the response and `X1`, ..., `Xp` are the predictors. These refer to column names of variables in a data frame, that we pass in through the `data` argument

E.g., for the prostate data, to regress the response variable `lpsa` (log PSA score) onto the predictors variables `lcavol` (log cancer volume) and `lweight` (log cancer weight) :

```
pros.lm = lm(lpsa ~ lcavol + lweight, data=pros.df)
class(pros.lm) # Really, a specialized list
```

```
## [1] "lm"
```

```
names(pros.lm) # Here are its components
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

```
pros.lm # It has a special way of printing
```

```
##
## Call:
## lm(formula = lpsa ~ lcavol + lweight, data = pros.df)
##
## Coefficients:
## (Intercept)       lcavol      lweight
##     -0.8134       0.6515       0.6647
```

## Utility functions

Linear models in R come with a bunch of **utility** functions, such as `coef()`, `fitted()`, `residuals()`, `summary()`, `plot()`, `predict()`, for retrieving coefficients, fitted values, residuals, producing summaries, producing diagnostic plots, making predictions, respectively

These tasks can also be done manually, by extracting at the components of the returned object from `lm()`, and manipulating them appropriately. But this is **discouraged**, because:

- The manual strategy is more tedious and error prone

- Once you master the utility functions, you'll be able to retrieve coefficients, fitted values, make predictions, etc., in the same way for model objects returned by `glm()`, `gam()`, and many others

## Retrieving estimated coefficients with `coef()`

So, what were the regression coefficients that we estimated? Use the `coef()` function, to retrieve them:

```
pros.coef = coef(pros.lm) # Vector of 3 coefficients
pros.coef
```

```
## (Intercept)      lcavol     lweight
##  -0.8134373   0.6515421   0.6647215
```

What does a linear regression coefficient mean, i.e., how do you interpret it? Note, from our linear model:

$$\mathbb{E}(Y|X) = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p$$

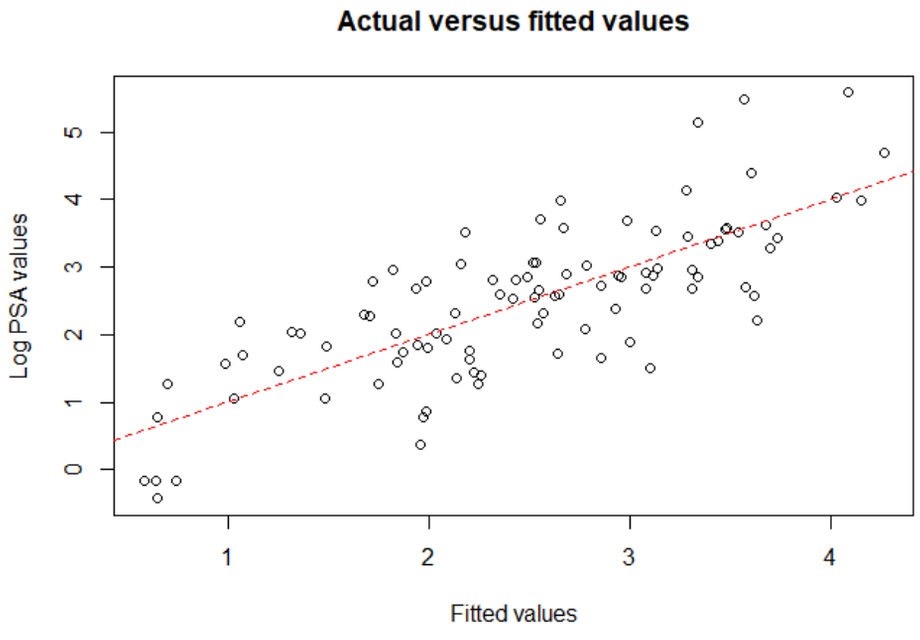
So, increasing predictor $X_j$ by one unit, while holding all other predictors fixed, increases the expected response by $\beta_j$

E.g., increasing `lcavol` (log cancer volume) by one unit (one cc), while holding `lweight` (log cancer weight) fixed, increases the expected value of `lpsa` (log PSA score) by $\approx 0.65$

## Retrieving fitted values with `fitted()`

What does our model predict for the log PSA scores of the 97 mean in question? And how do these compare to the actual log PSA scores? Use the `fitted()` function, then plot the actual values versus the fitted ones:

```
pros.fits = fitted(pros.lm) # Vector of 97 fitted values
plot(pros.fits, pros.df$lpsa, main="Actual versus fitted values",
     xlab="Fitted values", ylab="Log PSA values")
abline(a=0, b=1, lty=2, col="red")
```




## Displaying an overview with `summary()`

The function `summary()` gives us a nice summary of the linear model we fit:

```
summary(pros.lm) # Special way of summarizing
```

```
##
## Call:
## lm(formula = lpsa ~ lcavol + lweight, data = pros.df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.61051 -0.44135 -0.04666  0.53542  1.90424
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.81344    0.65309  -1.246 0.216033
## lcavol       0.65154    0.06693   9.734 6.75e-16 ***
## lweight      0.66472    0.18414   3.610 0.000494 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7419 on 94 degrees of freedom
## Multiple R-squared:  0.5955, Adjusted R-squared:  0.5869
## F-statistic: 69.19 on 2 and 94 DF,  p-value: < 2.2e-16
```

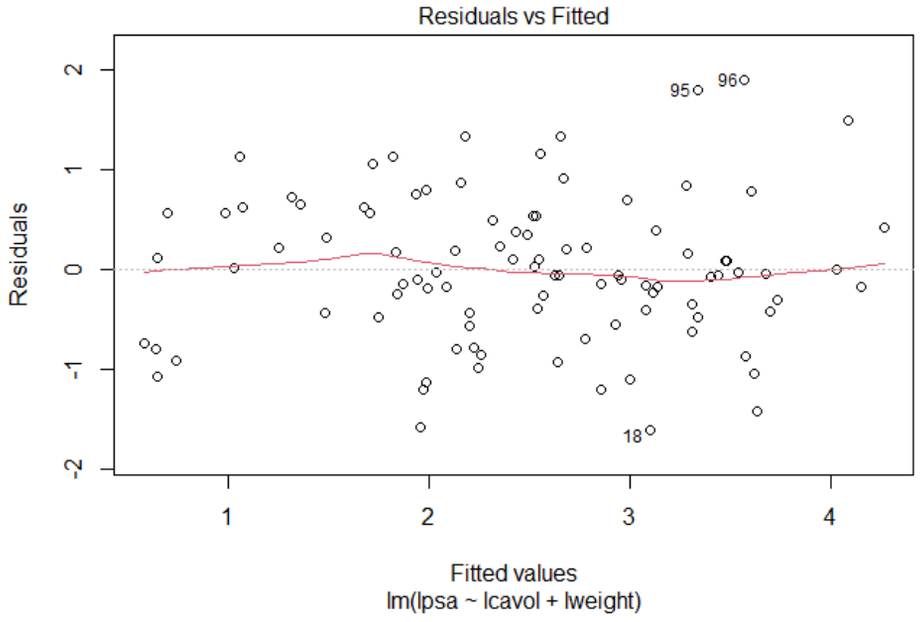
This tells us:

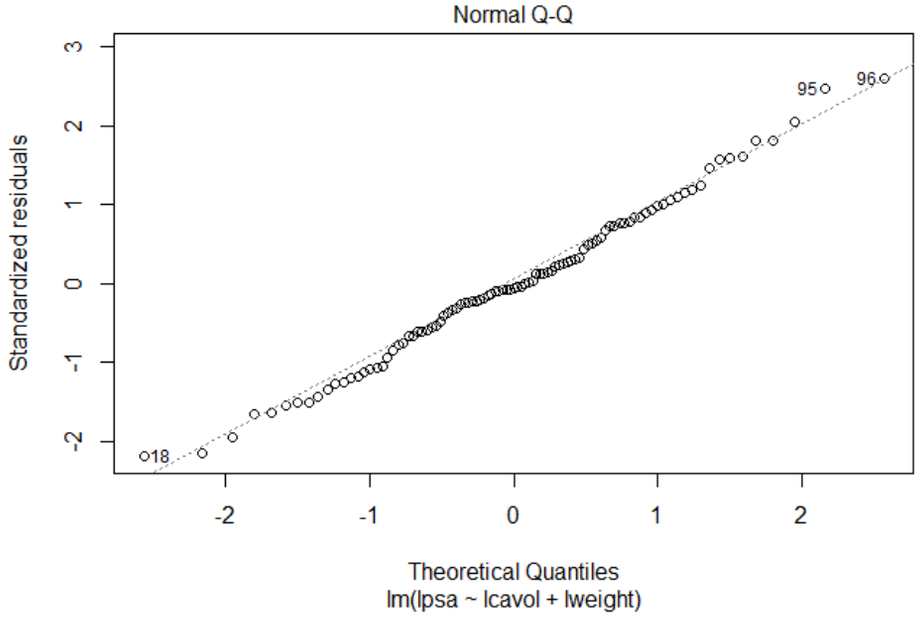
- The quantiles of the residuals: ideally, this is a perfect normal distribution
- The coefficient estimates
- Their standard errors
- P-values for their individual significances
- (Adjusted) R-squared value: how much variability is explained by the model?
- F-statistic for the significance of the overall model

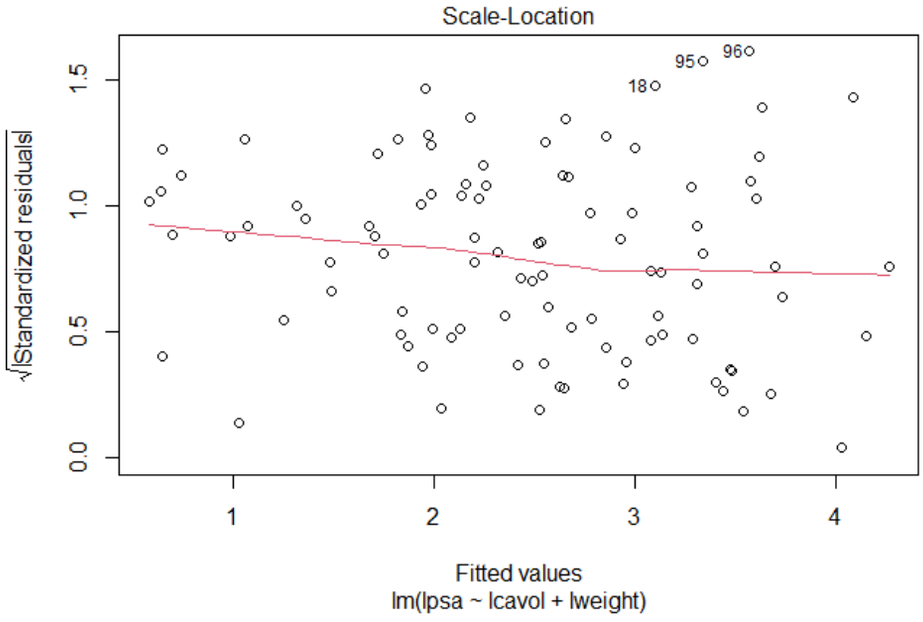
## Running diagnostics with `plot()`

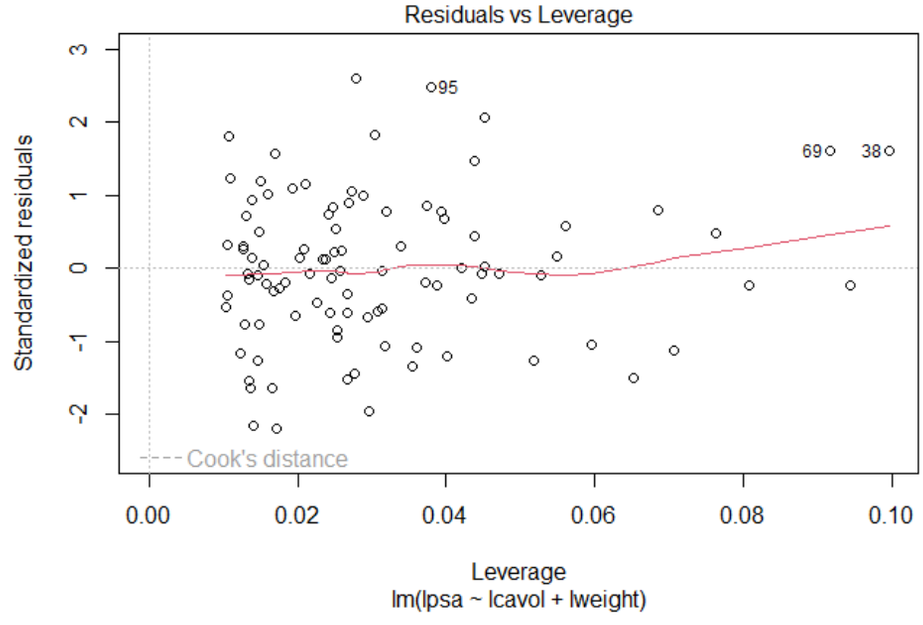
We can use the `plot()` function to run a series of diagnostic tests for our regression:

```
plot(pros.lm) # Special way of plotting
```


Residuals vs Fitted
Residuals
Fitted values
lm(lpsa ~ lcavol + lweight)
95
96
18


Normal Q-Q
Standardized residuals
Theoretical Quantiles
lm(lpsa ~ lcavol + lweight)
95
96
18




The results are pretty good:

- "Residuals vs Fitted" plot: points appear randomly scattered, no particular pattern
- "Normal Q-Q" plot: points are mostly along the 45 degree line, so residuals look close to a normal distribution
- "Scale-Location" and "Residuals vs Leverage" plots: no obvious groups, no points are too far from the center

## Making predictions with `predict()`

Suppose we had a new observation from a man whose log cancer volume was 4.1, and log cancer weight was 4.5. What would our linear model estimate his log PSA score would be? Use `predict()`:

```
pros.new = data.frame(lcavol=4.1, lweight=4.5) # Must set up a new data frame
pros.pred = predict(pros.lm, newdata=pros.new) # Now call predict with new df
pros.pred
```

```
##        1
## 4.849132
```

We'll learn much more about making/evaluating statistical predictions later in the course

## Some handy shortcuts

Here are some handy shortcuts, for fitting linear models with `lm()` (there are also many others):

- No intercept (no $\beta_0$ in the mathematical model): use `0 +` on the right-hand side of the formula, as in:

```
summary(lm(lpsa ~ 0 + lcavol + lweight, data=pros.df))
```

```
##
## Call:
## lm(formula = lpsa ~ 0 + lcavol + lweight, data = pros.df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.63394 -0.51181  0.00925  0.49705  1.90715
##
## Coefficients:
##         Estimate Std. Error t value Pr(>|t|)
## lcavol   0.66394    0.06638   10.00   <2e-16 ***
## lweight  0.43894    0.03249   13.51   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7441 on 95 degrees of freedom
## Multiple R-squared:  0.9273, Adjusted R-squared:  0.9258
## F-statistic: 606.1 on 2 and 95 DF,  p-value: < 2.2e-16
```

- Include all predictors: use `.` on the right-hand side of the formula, as in:

```
summary(lm(lpsa ~ ., data=pros.df))
```

```
##
## Call:
## lm(formula = lpsa ~ ., data = pros.df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.76644 -0.35510 -0.00328  0.38087  1.55770
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.181561   1.320568   0.137  0.89096
## lcavol       0.564341   0.087833   6.425 6.55e-09 ***
## lweight      0.622020   0.200897   3.096  0.00263 **
## age         -0.021248   0.011084  -1.917  0.05848 .
## lbph         0.096713   0.057913   1.670  0.09848 .
## svi1         0.761673   0.241176   3.158  0.00218 **
## lcp         -0.106051   0.089868  -1.180  0.24115
## gleason      0.049228   0.155341   0.317  0.75207
## pgg45        0.004458   0.004365   1.021  0.31000
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6995 on 88 degrees of freedom

## Multiple R-squared:  0.6634, Adjusted R-squared:  0.6328
## F-statistic: 21.68 on 8 and 88 DF,  p-value: < 2.2e-16
```

- Include interaction terms: use `:` between two predictors of interest, to include the interaction between them as a predictor, as in:

```
summary(lm(lpsa ~ lcavol + lweight + lcavol:svi, data=pros.df))
```

```
##
## Call:
## lm(formula = lpsa ~ lcavol + lweight + lcavol:svi, data = pros.df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.77358 -0.47304  0.00016  0.41458  1.52657
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.75666    0.62507  -1.211 0.229142
## lcavol       0.51193    0.07816   6.550 3.15e-09 ***
## lweight      0.66234    0.17617   3.760 0.000297 ***
## lcavol:svi1  0.25406    0.08156   3.115 0.002445 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7098 on 93 degrees of freedom
## Multiple R-squared:  0.6337, Adjusted R-squared:  0.6219
## F-statistic: 53.64 on 3 and 93 DF,  p-value: < 2.2e-16
```

# Part IV: *Beyond linear models*

## What other kinds of models are there?

Linear regression models, as we've said, are useful and ubiquitous. But there's a lot else out there. What else?

- Logistic regression
- Poisson regression
- Generalized linear models
- Generalized additive models
- Many, many others (generative models, Bayesian models, nonparametric models, machine learning "models", etc.)

Today we'll quickly visit logistic regression and generalized additive models.

## Logistic regression modeling

Given response $Y$ and predictors $X_1, \ldots, X_p$, where $Y \in \{0, 1\}$ is a **binary outcome**. In a **logistic regression model**, we posit the relationship:

$$\log \frac{\mathbb{P}(Y = 1|X)}{\mathbb{P}(Y = 0|X)} = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p$$

(where $Y|X$ is shorthand for $Y|X_1, \ldots, X_p$). Goal is to estimate parameters, also called coefficients $\beta_0, \beta_1, \ldots, \beta_p$

## Fitting a logistic regression model with `glm()`

We can use `glm()` to fit a logistic regression model. The arguments are very similar to `lm()`

The first argument is a formula, of the form `Y ~ X1 + X2 + ... + Xp`, where `Y` is the response and `X1`, ..., `Xp` are the predictors. These refer to column names of variables in a data frame, that we pass in through the `data` argument. We must also specify `family="binomial"` to get logistic regression

E.g., for the prostate data, suppose we add a column `lpsa.high` to our data frame `pros.df`, as the indicator of whether the `lpsa` variable is larger than log(10) (equivalently, whether the PSA score is larger than 10). To regress the binary response variable `lpsa.high` onto the predictor variables `lcavol` and `lweight`:

```
pros.df$lpsa.high = as.numeric(pros.df$lpsa > log(10)) # New binary outcome
table(pros.df$lpsa.high) # There are 56 men with high PSA scores
```

```
##
##  0  1
## 41 56
```

```
pros.glm = glm(lpsa.high ~ lcavol + lweight, data=pros.df, family="binomial")
class(pros.glm) # Really, a specialized list
```

```
## [1] "glm" "lm"
```

```
pros.glm # It has a special way of printing
```

```
##
## Call:  glm(formula = lpsa.high ~ lcavol + lweight, family = "binomial",
##     data = pros.df)
##
## Coefficients:
## (Intercept)       lcavol      lweight
##
##     -12.551        1.520        3.034
##
## Degrees of Freedom: 96 Total (i.e. Null);  94 Residual
## Null Deviance:       132.1
## Residual Deviance: 75.44     AIC: 81.44
```

## Utility functions work as before

For retrieving coefficients, fitted values, residuals, summarizing, plotting, making predictions, the utility functions `coef()`, `fitted()`, `residuals()`, `summary()`, `plot()`, `predict()` work pretty much just as with `lm()`. E.g.,

```
coef(pros.glm) # Logisitic regression coefficients
```

```
## (Intercept)      lcavol     lweight
##  -12.551478    1.520006    3.034018
```

```
summary(pros.glm) # Special way of summarizing
```

```
##
## Call:
## glm(formula = lpsa.high ~ lcavol + lweight, family = "binomial",
##     data = pros.df)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.7957  -0.6413   0.2192   0.5608   2.2209
##
## Coefficients:
```

```
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -12.5515     3.2422  -3.871 0.000108 ***
## lcavol        1.5200     0.3604   4.218 2.47e-05 ***
## lweight       3.0340     0.8615   3.522 0.000429 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 132.142  on 96  degrees of freedom
## Residual deviance:  75.436  on 94  degrees of freedom
## AIC: 81.436
##
## Number of Fisher Scoring iterations: 5
```

```
p.hat = fitted(pros.glm) # These are probabilities! Not binary outcomes
y.hat = round(p.hat) # This is one way we'd compute fitted outcomes
table(y.hat, y.true=pros.df$lpsa.high) # This is a 2 x 2 "confusion matrix"
```

```
##      y.true
## y.hat  0  1
##     0 33  5
##     1  8 51
```

## What does a logistic regression coefficient mean?

How do you interpret a logistic regression coefficient? Note, from our logistic model:

$$\frac{\mathbb{P}(Y = 1|X)}{\mathbb{P}(Y = 0|X)} = \exp(\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p)$$

So, increasing predictor $X_j$ by one unit, while holding all other predictor fixed, multiplies the odds by $e^{\beta_j}$. E.g.,

```
coef(pros.glm)
```

```
## (Intercept)      lcavol     lweight
##  -12.551478    1.520006    3.034018
```

So, increasing `lcavol` (log cancer volume) by one unit (one cc), while holding `lweight` (log cancer weight) fixed, multiplies the odds of `lpsa.high` (high PSA score, over 10) by $\approx e^{1.52} \approx 4.57$

## Creating a binary variable "on-the-fly"

We can easily create a binary variable "on-the-fly" by using the `I()` function inside a call to `glm()`:

```
pros.glm = glm(I(lpsa > log(10)) ~ lcavol + lweight, data=pros.df,
               family="binomial")
summary(pros.glm) # Same as before
```

```
##
## Call:
## glm(formula = I(lpsa > log(10)) ~ lcavol + lweight, family = "binomial",
##     data = pros.df)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.7957  -0.6413   0.2192   0.5608   2.2209
```

```
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -12.5515     3.2422  -3.871 0.000108 ***
## lcavol        1.5200     0.3604   4.218 2.47e-05 ***
## lweight       3.0340     0.8615   3.522 0.000429 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 132.142  on 96  degrees of freedom
## Residual deviance:  75.436  on 94  degrees of freedom
## AIC: 81.436
##
## Number of Fisher Scoring iterations: 5
```

## Generalized additive modeling

**Generalized additive models** allow us to do something that is like linear regression or logistic regression, but with a more flexible way of modeling the effects of predictors (rather than limiting their effects to be linear). For a continuous response $Y$, our model is:

$$\mathbb{E}(Y|X) = \beta_0 + f_1(X_1) + \ldots + f_p(X_p)$$

and the goal is to estimate $\beta_0, f_1, \ldots, f_p$. For a binary response $Y$, our model is:

$$\log \frac{\mathbb{P}(Y = 1|X)}{\mathbb{P}(Y = 0|X)} = \beta_0 + f_1(X_1) + \ldots + f_p(X_p)$$

and the goal is again to estimate $\beta_0, f_1, \ldots, f_p$

## Fitting a generalized additive model with `gam()`

We can use the `gam()` function, from the `gam` package, to fit a generalized additive model. The arguments are similar to `glm()` (and to `lm()`), with a key distinction

The formula is now of the form `Y ~ s(X1) + X2 + ... + s(Xp)`, where `Y` is the response and `X1`, ..., `Xp` are the predictors. The notation `s()` is used around a predictor name to denote that we want to model this as a smooth effect (nonlinear); without this notation, we simply model it as a linear effect

So, e.g., to fit the model

$$\mathbb{E}(\text{lpsa} \mid \text{lcavol}, \text{lweight}) = \beta_0 + f_1(\text{lcavol}) + \beta_2 \text{lweight}$$

we use:

```
library(gam, quiet=TRUE)
```

```
## Warning: package 'gam' was built under R version 4.2.3
```

```
## Warning: package 'foreach' was built under R version 4.2.3
```

```
## Loaded gam 1.22-2
```

```
pros.gam = gam(lpsa ~ s(lcavol) + lweight, data=pros.df)
class(pros.gam) # Again, a specialized list
```

```
## [1] "Gam" "glm" "lm"
```

```
pros.gam # It has a special way of printing
```

```
## Call:
## gam(formula = lpsa ~ s(lcavol) + lweight, data = pros.df)
##
## Degrees of Freedom: 96 total; 91.00006 Residual
## Residual Deviance: 49.40595
```

## Most utility functions work as before

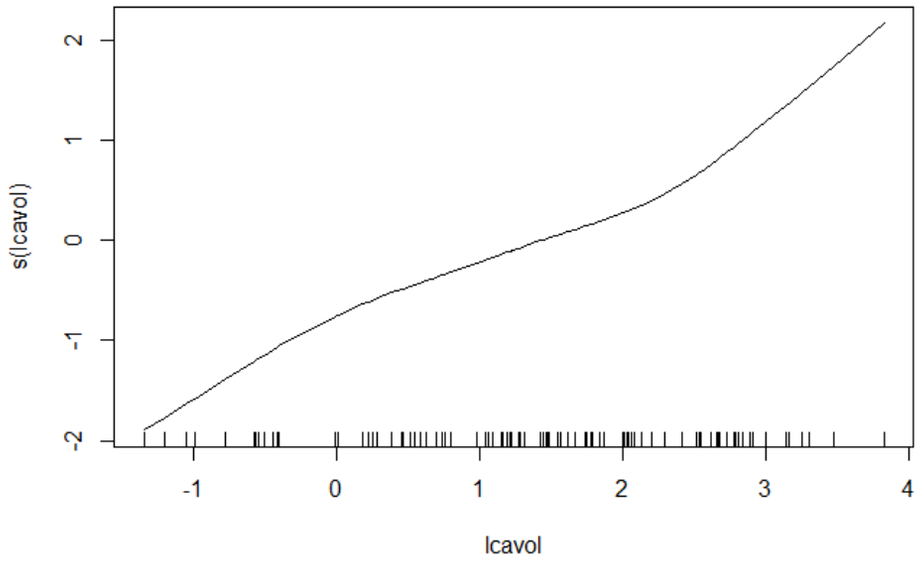
Most of our utility functions work just as before. E.g.,

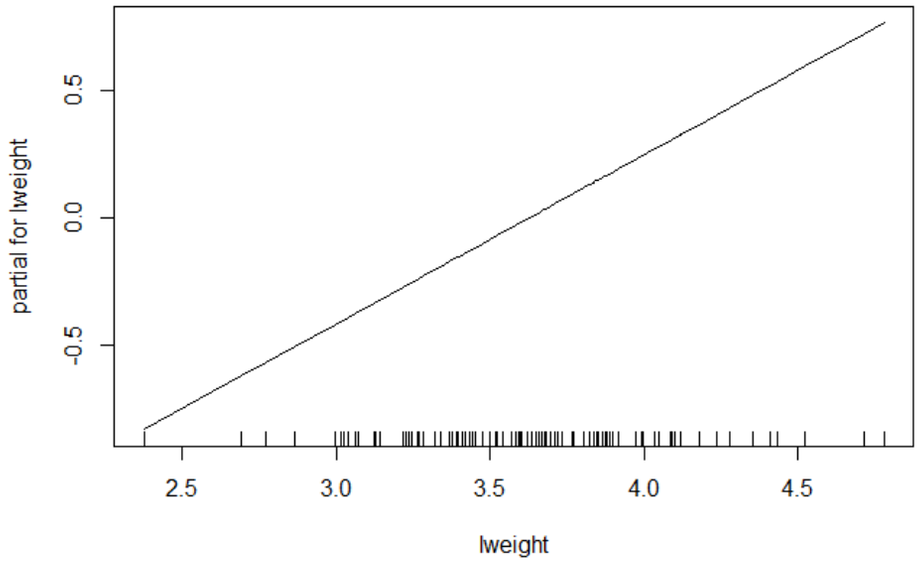
```
summary(pros.gam)
```

```
##
## Call: gam(formula = lpsa ~ s(lcavol) + lweight, data = pros.df)
## Deviance Residuals:
##        Min         1Q     Median         3Q        Max
## -1.6343202 -0.4601627  0.0004965  0.5121757  1.8516801
##
## (Dispersion Parameter for gaussian family taken to be 0.5429)
##
##     Null Deviance: 127.9177 on 96 degrees of freedom
## Residual Deviance: 49.406 on 91.0001 degrees of freedom
## AIC: 223.8339
##
## Number of Local Scoring Iterations: NA
##
## Anova for Parametric Effects
##           Df Sum Sq Mean Sq F value    Pr(>F)
## s(lcavol)  1 69.003  69.003 127.095 < 2.2e-16 ***
## lweight    1  7.181   7.181  13.227 0.0004571 ***
## Residuals 91 49.406   0.543
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Anova for Nonparametric Effects
##             Npar Df Npar F  Pr(F)
## (Intercept)
## s(lcavol)         3 1.4344 0.2379
## lweight
```

But now, `plot()`, instead of producing a bunch of diagnostic plots, shows us the effects that were fit to each predictor (nonlinear or linear, depending on whether or not we used `s()`):

```
plot(pros.gam)
```




We can see that, even though we allowed `lcavol` to have a nonlinear effect, this didn't seem to matter much as its estimated effect came out to be pretty close to linear anyway!

## Summary

- The first step of data analysis is to read in data, also to write out data after analysis
- Fitting models is critical to both statistical inference and prediction
- Exploratory data analysis is a very good first step and gives you a sense of what you're dealing with before you start modeling
- Linear regression is the most basic modeling tool of all, and one of the most ubiquitous
- `lm()` allows you to fit a linear model by specifying a formula, in terms of column names of a given data frame
- Utility functions `coef()`, `fitted()`, `residuals()`, `summary()`, `plot()`, `predict()` are very handy and should be used over manual access tricks
- Logistic regression is the natural extension of linear regression to binary data; use `glm()` with `family="binomial"` and all the same utility functions
- Generalized additive models add a level of flexibility in that they allow the predictors to have nonlinear effects; use `gam()` and utility functions