Introduction to R

Patrick O. Perry

Reading Data into R

Preparing Data

Believe it or not, reading in data is often the hardest part of working with R. If you collect and store your data in Excel or Google Docs, you will need to carefully format your spreadsheet. It should obey the following rules:

1. The spreadsheet should contain a single sheet.
2. Row 1 should contain variable names in consecutive cells, starting with Cell A1. For convenience, the names should be comprised of lowercase words and contain no symbols or punctuation.
3. Subsequent rows (starting with Row 2) should contain your observations (data).
4. If the value of a variable is missing for a particular observation, the coressponding cell in the spreadsheet should be empty.
5. All other cells in the spreadsheet should be empty.

This sounds straightforward, but many spreadsheets that you find "in the wild" do not obey these rules. You will have to reformat these spreadsheets, usually by deleting empty rows and columns and by deleting notes and other annotations.

Even if your spreadsheet is formatted as above, R cannot open Excel files. To save your data to an R-compatible file format, exportyour data as a "Column-Separated Value" (CSV) file. You can do this from the File menu in Excel.

Reading CSV Files

If you have a property-formatted CSV file, you can read it into R using the `read.csv` function. There are two ways to specify the file. To use your system's file chooser, run the command.

```
data <- read.csv(file.choose())
```

Alternatively, if you know the name of the file, you can pass it directly to the `read.csv` function. Note that if you pass the file name directly, you must either specify the full path to the file, or you must set the "working directory" to be the directory thatcontains the file. To set the working directory, either use the `setwd` function or run the *Set Working Directory* command from RStudio's **Session** menu.

Suppose that I want to open a file named "bikedata.csv", which is stored in the "~/Datasets" directory on my system. I first set the working directory to "~/Datasets" by Choosing **Session** > *Set Working Directory* > *Choose Directory* .... This will execute the command

```
setwd("~/Datasets")
```

(In fact, if I do not want to use the menu system, then I can just type this command directly to achieve the same effect.) Once the working directory is set, I can read in the data to a variable named `data` by executing the command

```
bikedata <- read.csv("bikedata.csv")
```

One nice feature of the `read.csv` function is that it correctly web addresses. For example to read the file "bikedata.csv" from the course website, you can run the command

```
bikedata <- read.csv("http://ptrckprry.com/course/forecasting/data/bikedata.csv")
```

This downloads the file and reads the data into the `bikedata` variable.

Basic Types and Operations

Variables

In R, we use the term "variable" to refer to a name-value pair. You should not confuse this concept with the types of variables you have seen in your math classes (they are similar in some ways, but different in others).

In the last section, when we ran the command `data <- read.csv(file.choose())` we created a variable with name `data` and value equal to the contents of the chosen file.

To create a variable or to assign a new value to an existing variable, use the assignment command (`<-`), which is meant to look like an arrow pointing from the value to the variable name. For example, the command

```
a <- 2.7
```

means "assign the value `2.7` to the variable named `a`". Another way to read this is "variable `a` gets the value `2.7`".

When you have a variable, you can use the name in place of the value:

```
a + 10
```

```
## [1] 12.7
```

```
5 * a
```

```
## [1] 13.5
```

You can see the value of a variable, by typing its name and pressing enter:

```
a
```

```
## [1] 2.7
```

An alternative way to read in a file to the variable named `data` is to run the following sequence of commands:

```
filename <- file.choose()
data <- read.csv(filename)
```

The first command asks the user to choose a file, and stores the resulting name in the `filename` variable. This variable contains the name of the file, but not the actual contents. The second command takes the name of the file, opens it, reads the contents into memory, and stores the result in the `data` variable.

Functions

Besides variables, the other main concept you need to learn in R is that of a *function*. You are probably familiar with the concept of a function from your mathematics courses, and a function in R is very similar: a function is something that takes zero or more values, then performs a sequence of actions and returns a result.

We have already seen three functions: `file.choose`, `read.csv`, and `setwd`. We *call* a function by putting a pair of parentheses `()` after the function name. Many functions, including `read.csv` and `setwd` require one or more values as input. We refer to these values as *arguments*, and we specify them by putting the values inside the parentheses. When we do so, we say that we are *passing the value* of the argument to the function.

Sometimes a function will have optional arguments. These are arguments that, if left unspecified, will be given reasonable default values. For example, by default, the `file.choose` function forces the user to choose an existing file. To allow the user to choose a name for new file, pass the argument `new=TRUE` to file `file.choose` function:

```
file.choose(new=TRUE)
```

Before, we did not specify the `new` argument, and it defaulted to the value `FALSE`.

Data Frames

The `read.csv` command opens the file and reads the data into a type of object called a "dataframe". Conceptually, a data frame is just like a spreadsheet: it has columns, corresponding to variables, and rows, corresponding to observations. Each row and column has a name. Usually, the row names are the character strings "1", "2", etc., but this is not always the case.

To see the first 6 rows in the `bikedata` data frame, run the command

```
head(bikedata)
```

```
##   vehicle colour passing.distance street helmet kerb            datetime
## 1     Car   Blue            2.114  Urban      N  0.5 2006-05-11 16:30:00
## 2     HGV    Red            0.998  Urban      N  0.5 2006-05-11 16:30:00
## 3     LGV   Blue            1.817  Urban      N  0.5 2006-05-11 16:30:00
## 4     Car   <NA>            1.640  Urban      N  0.5 2006-05-11 16:30:00
## 5     Bus  Other            1.544  Urban      N  0.5 2006-05-11 16:30:00
## 6     Car   Grey            1.509  Urban      N  0.5 2006-05-11 16:30:00
##   bikelane      city
## 1        N Salisbury
## 2        N Salisbury
## 3        N Salisbury
## 4        N Salisbury
## 5        N Salisbury
## 6        N Salisbury
```

We can see that there are nine columns, named `vehicle`, `colour`, `passing.distance`, `street`, `helmet`, `kerb`, `datetime`, `bikelane`, and `city`.

To see a summary of the entire data frame, use the `summary` function:

```
summary(bikedata)
```

```
##  vehicle         colour    passing.distance         street     helmet
##  Bus :  46   Blue   :636   Min.   :0.394    Main       :1637   N:1206
##  Car :1708   Grey   :531   1st Qu.:1.303    OneWay1    :   9   Y:1149
##  HGV :  82   Red    :378   Median :1.529    OneWay2    :  13
##  LGV : 293   White  :333   Mean   :1.564    Residential:  39
##  PTW :  34   Black  :262   3rd Qu.:1.790    Rural      :   2
##  SUV : 143   (Other):201   Max.   :3.787    Urban      : 655
##  Taxi:  49   NA's   : 14
##       kerb                       datetime    bikelane        city
##  Min.   :0.2500   2006-05-20 16:21:00:  93   N:2305   Bristol  : 450
##  1st Qu.:0.2500   2006-05-20 15:48:00:  75   Y:  50   Salisbury:1905
##  Median :0.5000   2006-05-31 09:04:00:  70
##  Mean   :0.6702   2006-05-20 15:34:00:  65
##  3rd Qu.:1.0000   2006-05-27 10:01:00:  64
##  Max.   :1.2500   2006-05-27 09:25:00:  63
##                   (Other)            :1925
```

Extracting Columns

Let's say we want to investigate the `passing.distance` variable. To do this, we must first *extract* that column from the `bikedata` dataframe. There are three ways to do this:

```
x <- bikedata$passing.distance
x <- bikedata[["passing.distance"]]
x <- bikedata[,"passing.distance"]
```

All three commands are equivalent ways to extract the `passing.distance` column and store it in a variable named `x`. The `$` form is the most common, but you will sometimes see the other two forms, as well.

Vectors

Data frame columns are stored in a data type called a "vector". Conceptually, a vector is a one-dimensional array of values, indexed by integers starting at `1`. Most functions in R operate on vectors.

You can access individual values by using double square-brackets. For example, to see the first element of the vector, type the command

```
x[[1]]
```

```
## [1] 2.114
```

To see the fifth value, type the command

```
x[[5]]
```

```
## [1] 1.544
```

To see how many elements are contained in the vector, use the `length` function:

```
length(x)
```

```
## [1] 2355
```

To see the last element, type

```
x[[length(x)]]
```

```
## [1] 1.031
```

To extract a subvector, use single square brackets. For example, the subvector consisting of the first 25 elements is

```
x[1:25]
```

```
##  [1] 2.114 0.998 1.817 1.640 1.544 1.509 1.290 1.512 1.049 1.932 1.145
## [12] 1.410 1.428 1.494 1.570 2.103 0.896 1.160 1.290 1.963 2.436 2.304
## [23] 1.482 1.492 1.432
```

Here, `1:25` is shorthand for "integers 1 to 25". Since not all 25 values fit onto a single line, R wraps the values. At the start of each line, R prints the index of the first value on the line in square brackets. Looking at the output above, we can see that `1.410` is the 12th element and `1.492` is the 24th element of the result.

You may have asked yourself earlier why the output of `x[[1]]` and other similar commands was prefixed by `[1]`. The reason for this is that R doesn't have the concept of a "single value" or "scalar". The only way to represent the value of `x[[1]]` is as a length-one vector. The output

denotes a vector with a single value (`2.114`), stored at index `1`.

Since there is no concept of a "scalar" in R, the command `x[[1]]` is equivalent to `x[1:1]`, which is also the same as `x[1]`. In other programming languages, `x[[1]]`, "the first element of `x`", and `x[1]`, "the subvector of `x` starting and ending at index `1`" would be different; in R, these are identical. Because of this, most people use single brackets instead of double brackets when indexing vectors, writing `x[1]` and `x[5]` instead of `x[[1]]` and `x[[5]]`.

Plots and Descriptive Statistics

Descriptive Statistics for Numeric Variables

There are a variety of functions for computing descriptive statistics for the values stored in a vector.

- Sum of values:

  ```
  sum(x)
  ```

  ```
  ## [1] 3683.013
  ```

- Measures of central tendency (sample mean and median):

```r
mean(x)
```

```
## [1] 1.563912
```

```r
median(x)
```

```
## [1] 1.529
```

- Measures of variability (sample standard deviation and sample variance):

```r
sd(x)
```

```
## [1] 0.3834545
```

```r
var(x)
```

```
## [1] 0.1470373
```

- Extreme values (minimum and maximum):

```r
min(x)
```

```
## [1] 0.394
```

```r
max(x)
```

```
## [1] 3.787
```

- Quantiles:

```r
quantile(x, .25) # first quartile
```

```
##   25%
## 1.303
```

```r
quantile(x, .75) # third quartile
```

```
##    75%
## 1.7905
```

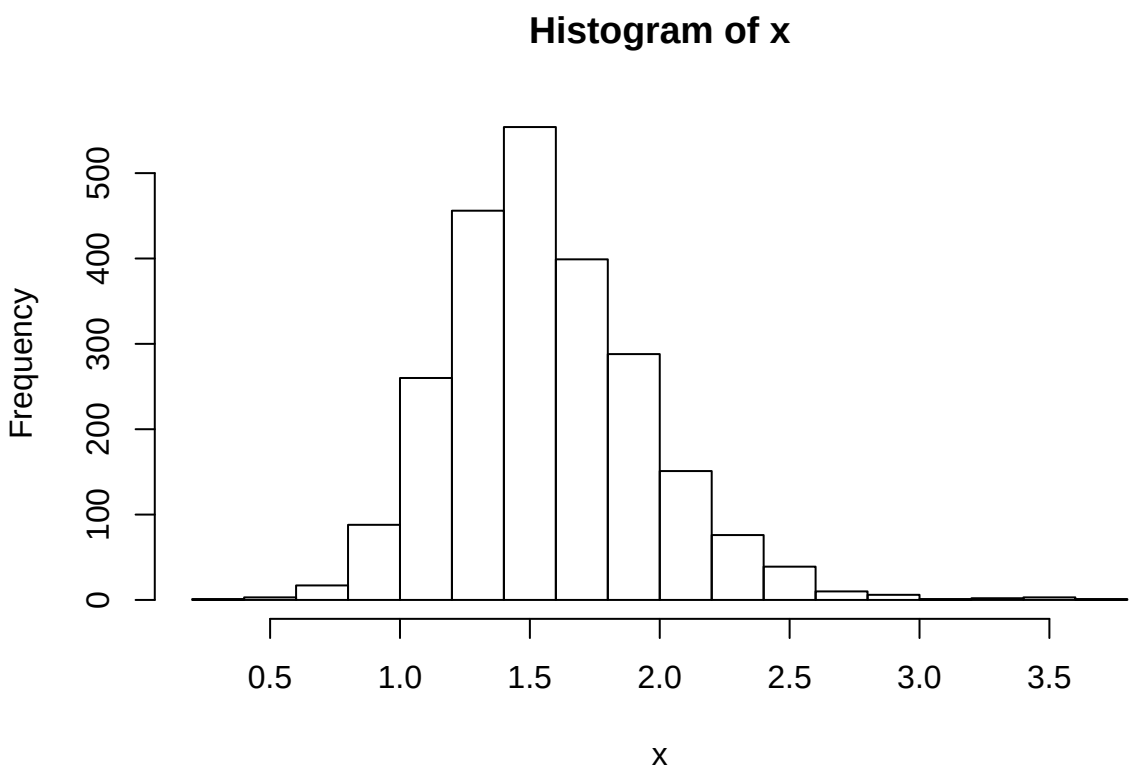
```r
quantile(x, .99) # 99th percentile
```

```
##     99%
## 2.58006
```

Plots for Numeric Variables

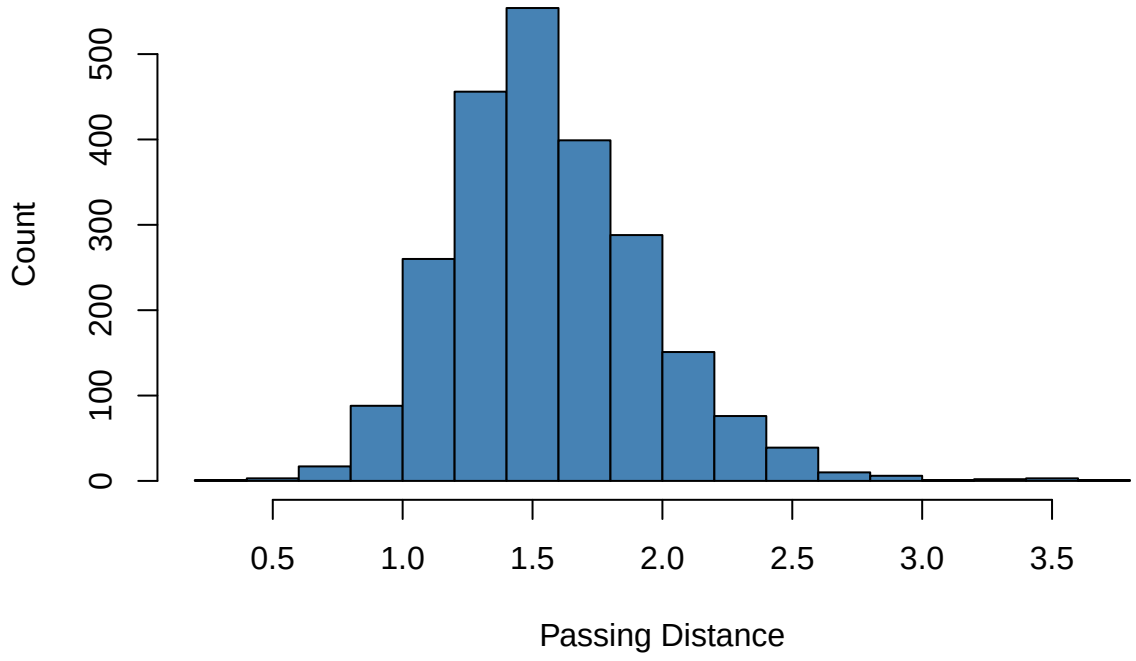
We can use the `hist` command to make a histogram of the values stored in a vector:

```
hist(x)
```



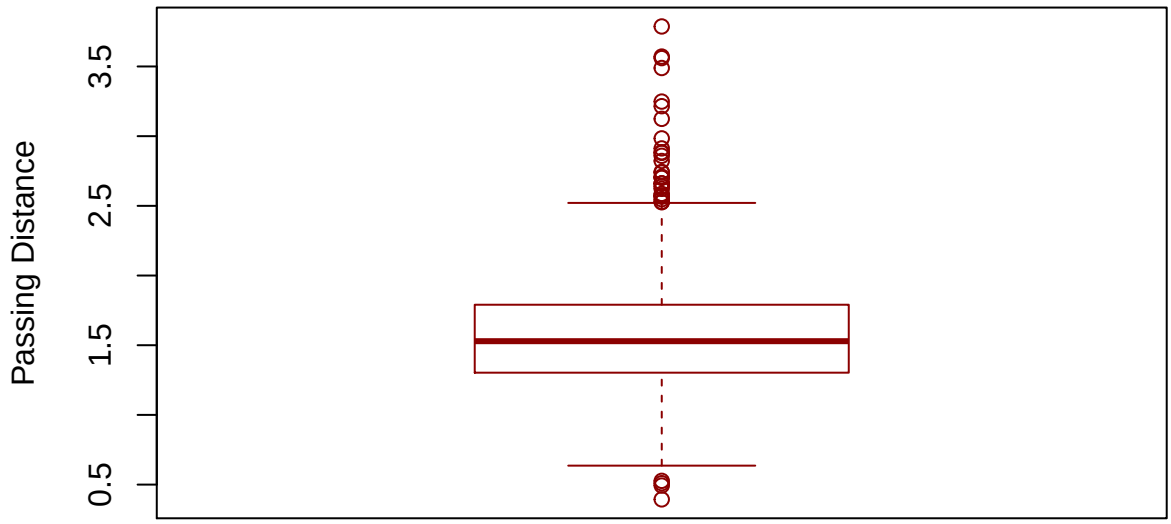
By default, the output looks fine when printed in black and white, but it isn't very pretty. We can specify the bin color, change the axis labels, and omit the main title by passing additional arguments to this `hist` function

```
hist(x, col="steelblue", xlab="Passing Distance", ylab="Count",
     main="")
```

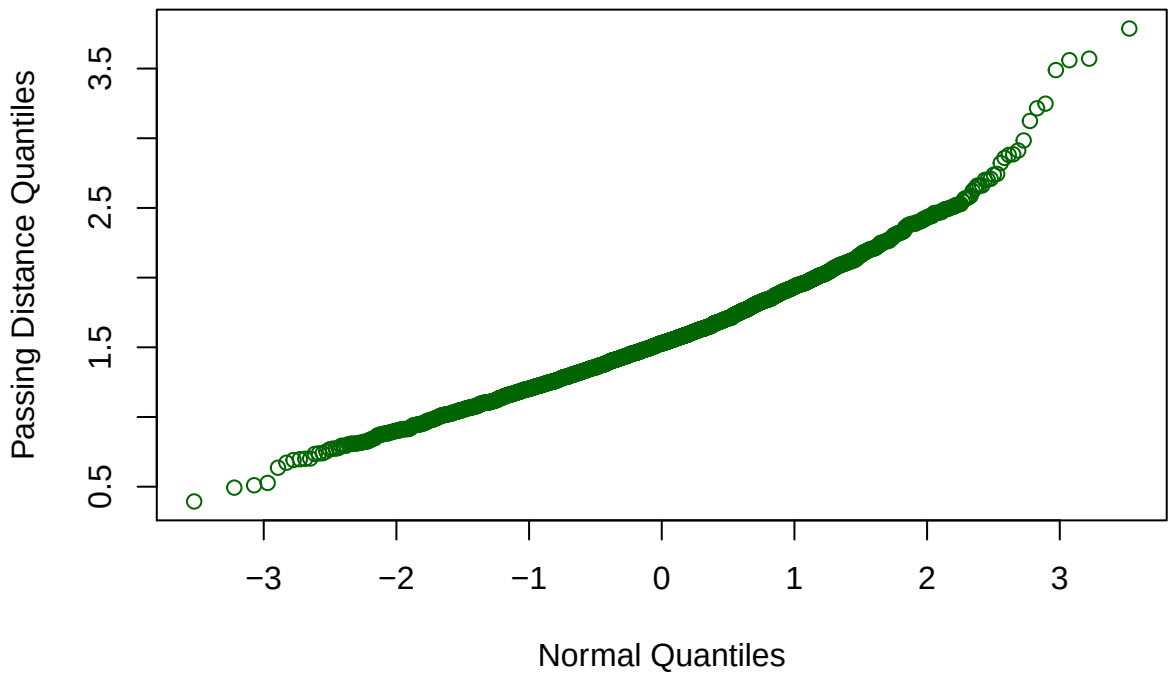


Use the `boxplot` and `qqnorm` commands to make boxplots and normal probability plots, as in the following examples:

```
boxplot(x, border="darkred", ylab="Passing Distance")
```



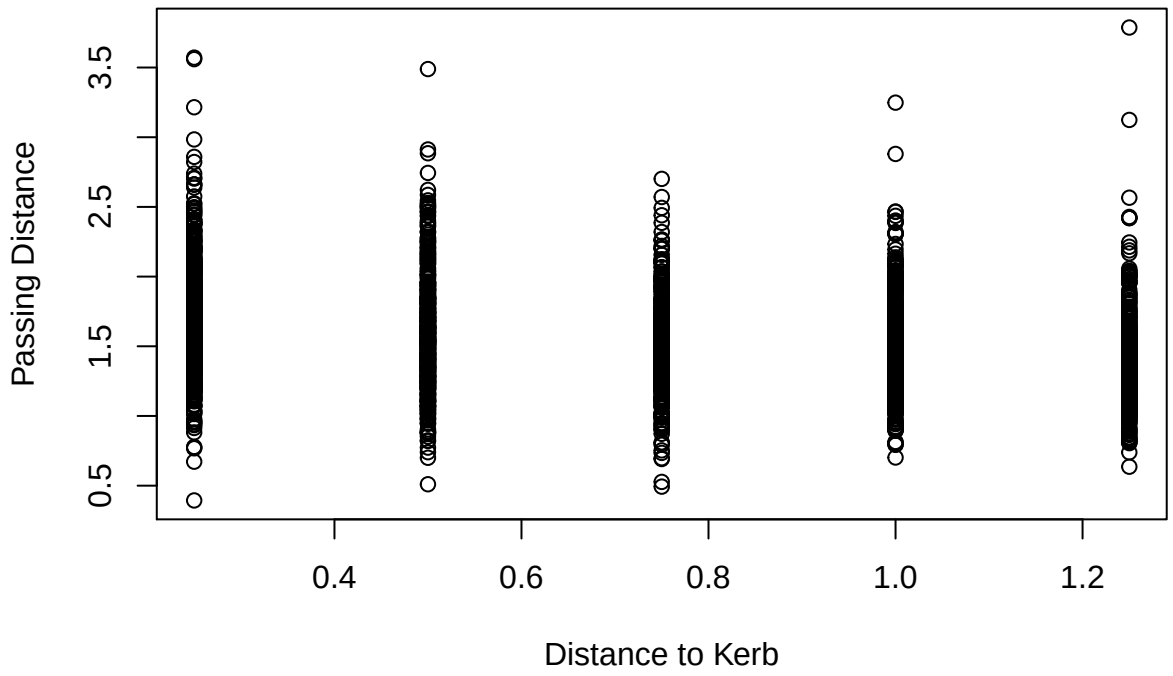

```
qqnorm(x, col="darkgreen", xlab="Normal Quantiles",
       ylab="Passing Distance Quantiles",
       main="")
```



Scatterplots

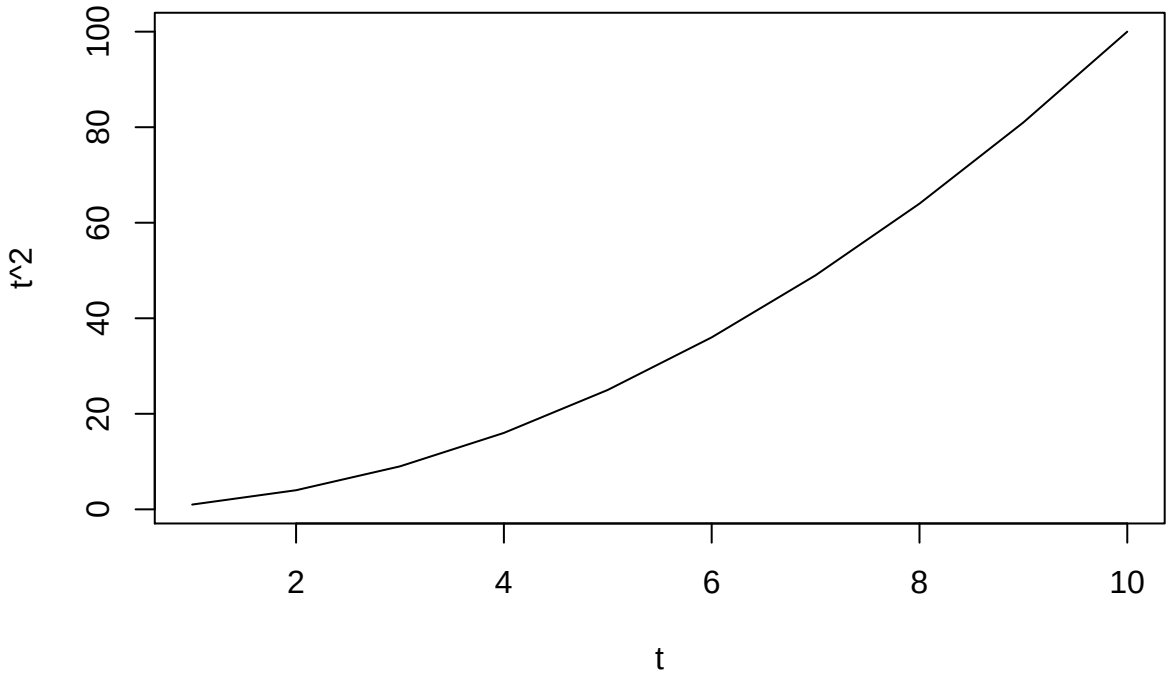
We can make a scatter plot using the `plot` command. For example, to plot passing distance versus distance to the kerb, run the command

```
plot(bikedata$kerb, bikedata$passing.distance,
     xlab="Distance to Kerb",
     ylab="Passing Distance")
```



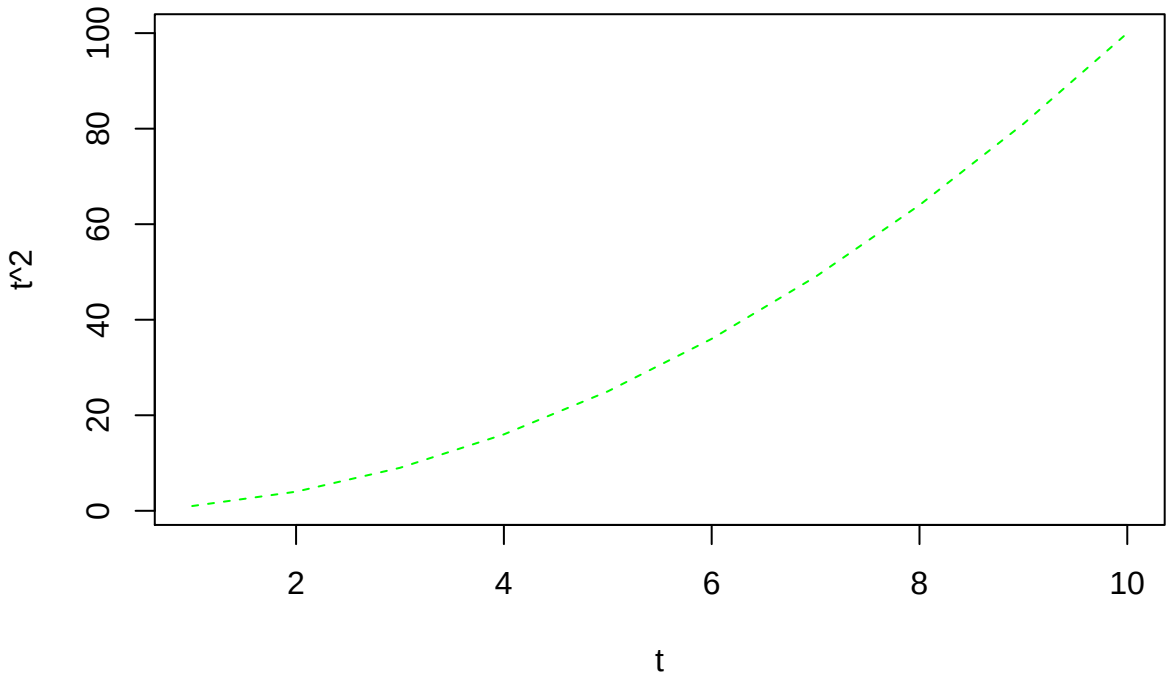
To connect the points with lines, use the `type="l"` argument. For example

```
t <- 1:10
plot(t, t^2, type="l")
```



We can use the `lty` and `col` arguments to change the style and color of the line:

```
t <- 1:10
plot(t, t^2, type="l", lty=2, col="green")
```



Categorical Variables

So far, we have seen how to use R to summarize and plot a numeric (quantitative) variable. R also has very good support for categorical (qualitative) variables, referred to as *factors*.

To see levels, the set of possible values for a factor variable, use the `levels` function. For example, to see the levels of the `colour` variable:

```
levels(bikedata$colour)
```

```
## [1] "Black" "Blue"  "Green" "Grey"  "Other" "Red"   "White"
```

To tabulate the values of the variable, use the `table` command, as in

```
table(bikedata$colour)
```

```
##
## Black  Blue Green  Grey Other   Red White
##   262   636   149   531    52   378   333
```

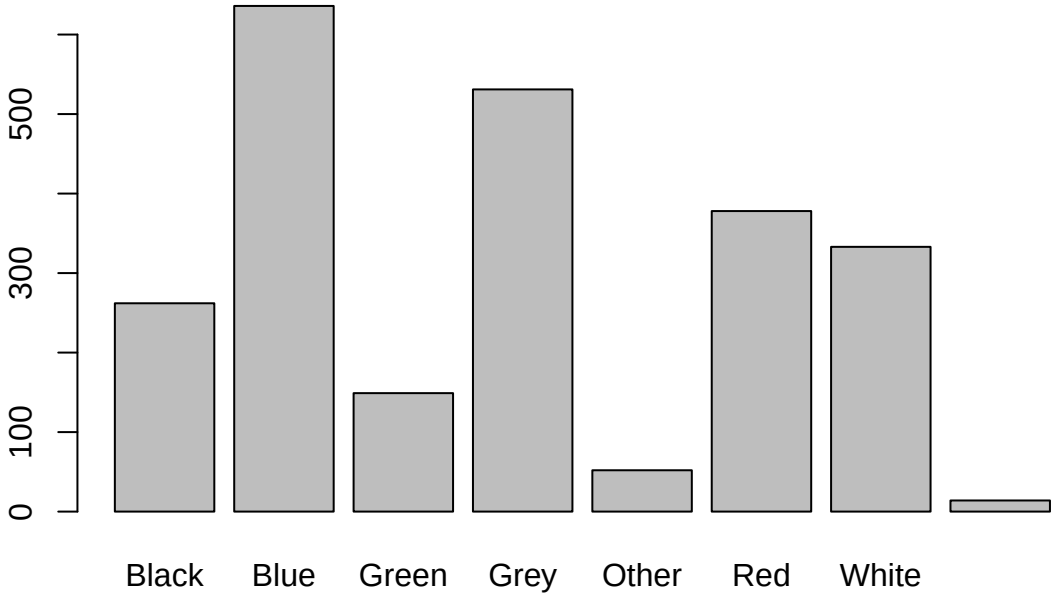
Note: be default, the `table` command omits missing values. To include these values in the output, include `useNA="ifany"` in the call to `table`:

```
table(bikedata$colour, useNA="ifany")
```

```
##
## Black  Blue Green  Grey Other   Red White  <NA>
##   262   636   149   531    52   378   333    14
```

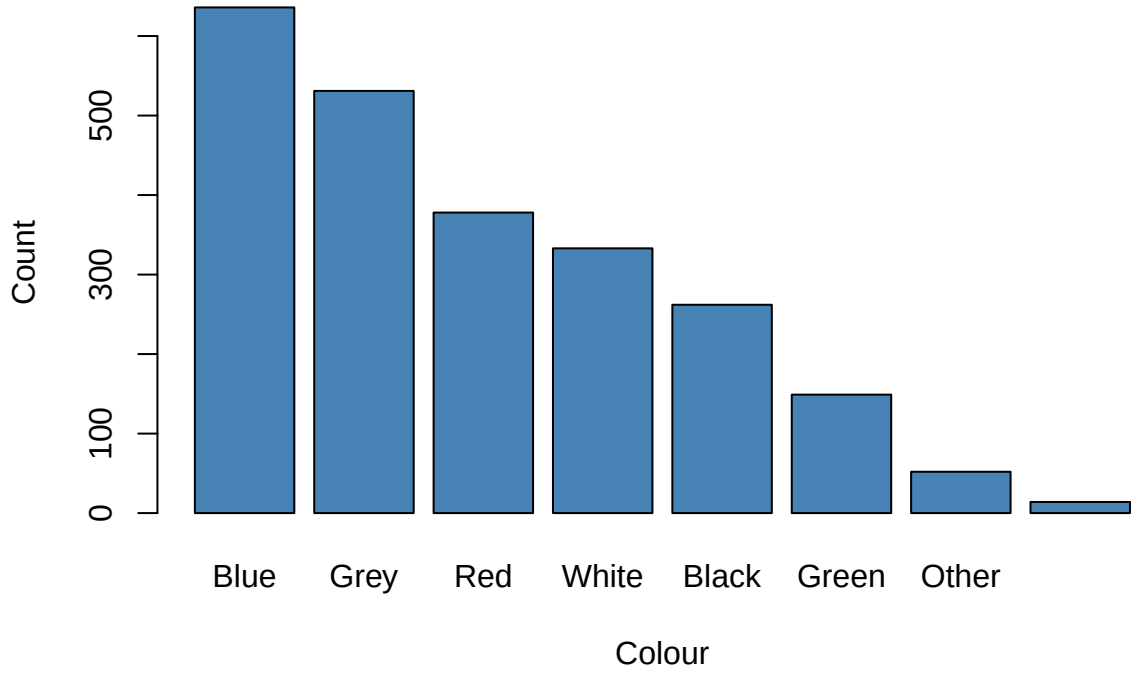
We can present tabulated counts in a bar plot using the following commands

```
tab <- table(bikedata$colour, useNA="ifany")
barplot(tab)
```



Usually, it makes sense to arrange the table values in decreasing order. Here is an example with sorted counts that adds axis labels and changes the bar colors:

```
barplot(sort(tab, decreasing=TRUE), xlab="Colour", ylab="Count",
        col="steelblue")
```



Inference

Inference for a Population Mean

We can use the `t.test` function to test a hypothesis about a population mean.

```
t.test(bikedata$passing.distance)
```

```
##
##  One Sample t-test
##
## data:  bikedata$passing.distance
## t = 197.922, df = 2354, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  1.548417 1.579407
## sample estimates:
## mean of x
##  1.563912
```

This reports the t statistic, the degrees of freedom, the p-value, and the sample mean. The command also reports a 95% confidence interval for the population mean. To change the confidence level, use the `conf.level` argument, as in

```
t.test(bikedata$passing.distance, conf.level=0.99)
```

```
##
##  One Sample t-test
##
## data:  bikedata$passing.distance
## t = 197.922, df = 2354, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 99 percent confidence interval:
##  1.543542 1.584282
## sample estimates:
## mean of x
##  1.563912
```

By default, the null hypothesis is that the true (population) mean is equal to 0, and the alternative hypothesis is that the true mean is not equal to 0. To use a different null, pass the `mu` argument. To use a different alternative, pass `alternative="less"` or `alternative="greater"`. For example, to test the null hypothesis that the true mean is equal to 1.5 against the alternative that it is greater, run the command

```
t.test(bikedata$passing.distance, alternative="greater", mu=1.5)
```

```
##
##  One Sample t-test
##
```

```
## data:  bikedata$passing.distance
## t = 8.0884, df = 2354, p-value = 4.786e-16
## alternative hypothesis: true mean is greater than 1.5
## 95 percent confidence interval:
##  1.55091     Inf
## sample estimates:
## mean of x
##  1.563912
```

Note that for a one-sided alternative, the confidence interval is one-sided, as well.

Inference for a Population Proportion

To perform a test on a population proportion, use the `prop.test` function. This performs a test on a population proportion that is slightly different than the one we cover in the core statistics course, but it will give you a very similar answer.

In the first argument, specify `x`, the number of successes; in the second argument, specify `n`, the number of trials. By default, the null value of the population proportion is `0.5`; to specify a different value, use the `p` argument.

For example, to test the null hypothesis that the true proportion of cars passing the rider on his route is exactly equal to 40%, we first tabulate the `colour` variable,

```
table(bikedata$colour)
```

```
##
## Black  Blue Green  Grey Other   Red White
##   262   636   149   531    52   378   333
```

In this instance, the number of "successes" is equal to the number of blue cars, `636`. Recall that some of the values for the `colour` variable are missing. If the missingness is unrelated to the actual color, then we can safely ifnore these values; in this case, the number of "trials" is equal to the sumof the counts for all of the non-missing values.

```
sum(table(bikedata$colour))
```

```
## [1] 2341
```

Now, to test the proportion, we run the command:

```
prop.test(636, 2341, p=0.40)
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  636 out of 2341, null probability 0.4
## X-squared = 160.0812, df = 1, p-value < 2.2e-16
## alternative hypothesis: true p is not equal to 0.4
## 95 percent confidence interval:
##  0.2538356 0.2902788
## sample estimates:
##         p
## 0.2716788
```

As with the `t.test` function, we can use a one-sided alternative or specify a different confidence level for the interval by using the `alternative` or `conf.level` argument, respectively. Here is a test of the null that the true proportion is equal to `0.5` against the alternative that it is less, along with a one-sided 99% confidence interval:

```
prop.test(636, 2341, p=0.50, alternative="less", conf.level=0.99)
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  636 out of 2341, null probability 0.5
## X-squared = 487.2379, df = 1, p-value < 2.2e-16
## alternative hypothesis: true p is less than 0.5
## 99 percent confidence interval:
##  0.0000000 0.2937932
## sample estimates:
##         p
## 0.2716788
```

Linear Regression

Model Fitting

We fit a linear regression model using the `lm` command. For example, suppose we want to fit a with response variable `sqrt(passing.distance)` and predictors `helmet`, `vehicle`, and `kerb`. We would use the following command:

```
model <- lm(sqrt(passing.distance) ~ helmet + vehicle + kerb,
            data = bikedata)
```

The formula syntax `y ~ x1 + x2 + x3` means use `y` as the response variable, use `x1`, `x2`, and `x3` as predictor variables, and include an intercept in the model. We can either store all of the variables in a data frame and use the `data` argument as above, or we can extract the variable first. For example, we could run the following commands to fit an equivalent model:

```
sqrt.passing.distance <- sqrt(bikedata$passing.distance)
helmet <- bikedata$helmet
vehicle <- bikedata$vehicle
kerb <- bikedata$kerb
model1 <- lm(sqrt.passing.distance ~ helmet + vehicle + kerb)
```

In the latter instance, there is no need to pass the `data` argument to `lm`; the response and the predictor variables exist in the environment.

Inference for Regression Parameters

When we have a fitted value, we can get the coefficient estimates, their standard errors, t statistics, and p values with the `summary` command.

```
summary(model)
```

```
##
## Call:
## lm(formula = sqrt(passing.distance) ~ helmet + vehicle + kerb,
##     data = bikedata)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.67612 -0.09319 -0.00633  0.08825  0.74546
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.217556   0.022035  55.255  < 2e-16 ***
## helmetY     -0.022548   0.006028  -3.741 0.000188 ***
## vehicleCar   0.112074   0.021642   5.179 2.43e-07 ***
## vehicleHGV   0.048953   0.026739   1.831 0.067261 .
## vehicleLGV   0.091336   0.022986   3.973 7.30e-05 ***
## vehiclePTW   0.097733   0.032753   2.984 0.002875 **
## vehicleSUV   0.111934   0.024561   4.557 5.45e-06 ***
## vehicleTaxi  0.068841   0.029768   2.313 0.020833 *
## kerb        -0.103258   0.008497 -12.152  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1448 on 2346 degrees of freedom
## Multiple R-squared:  0.09062,    Adjusted R-squared:  0.08752
## F-statistic: 29.22 on 8 and 2346 DF,  p-value: < 2.2e-16
```

This also reports some information about the residuals, along with the R^2 and the adjusted R^2 values.

To get a confidence interval for a population regression coefficient, use the `confint` command. For example, we can get a 99% confidence interval for coefficient of `helmetY` with

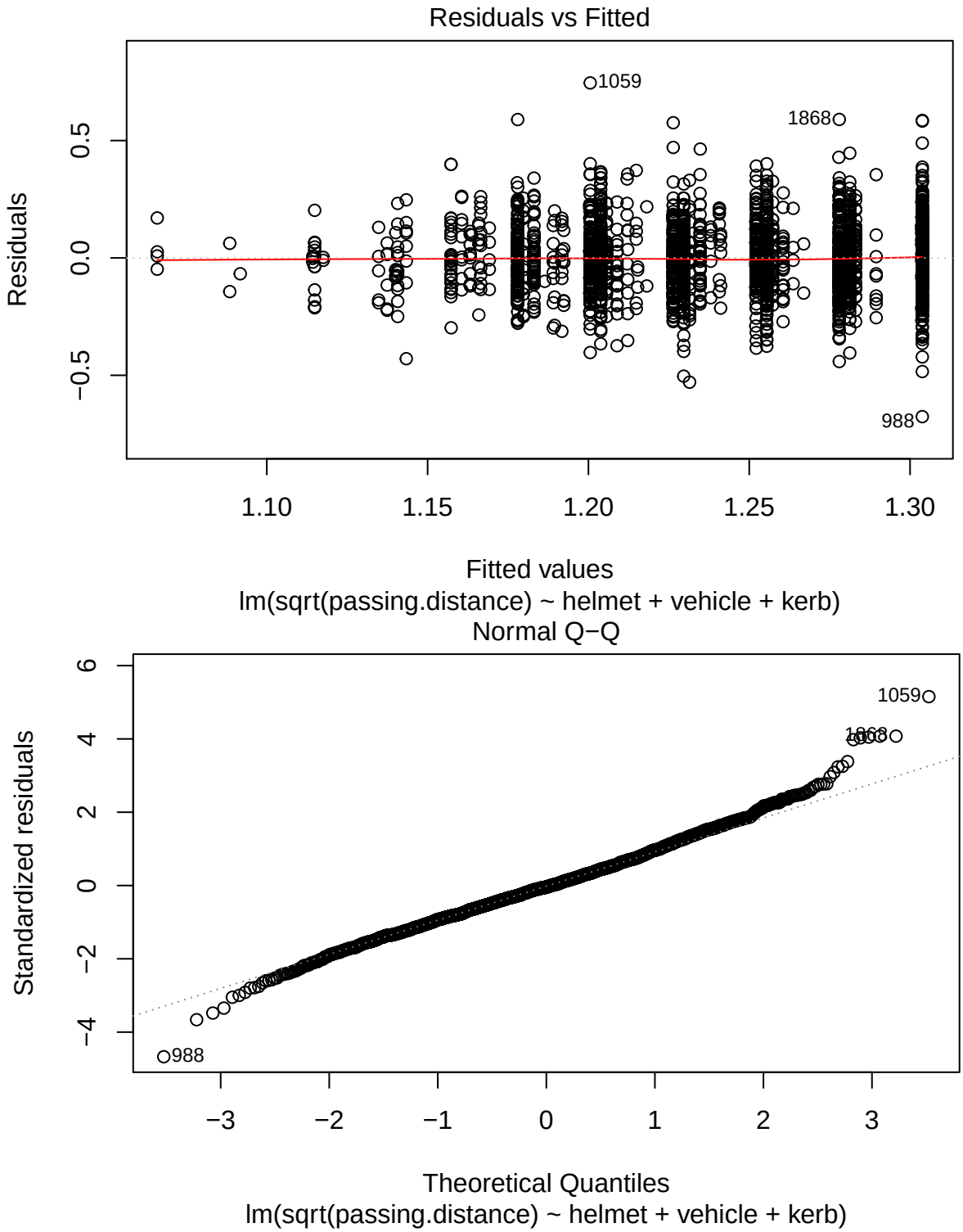
```
confint(model, "helmetY", 0.99)
```

```
##              0.5 %       99.5 %
## helmetY -0.0380878 -0.007008733
```

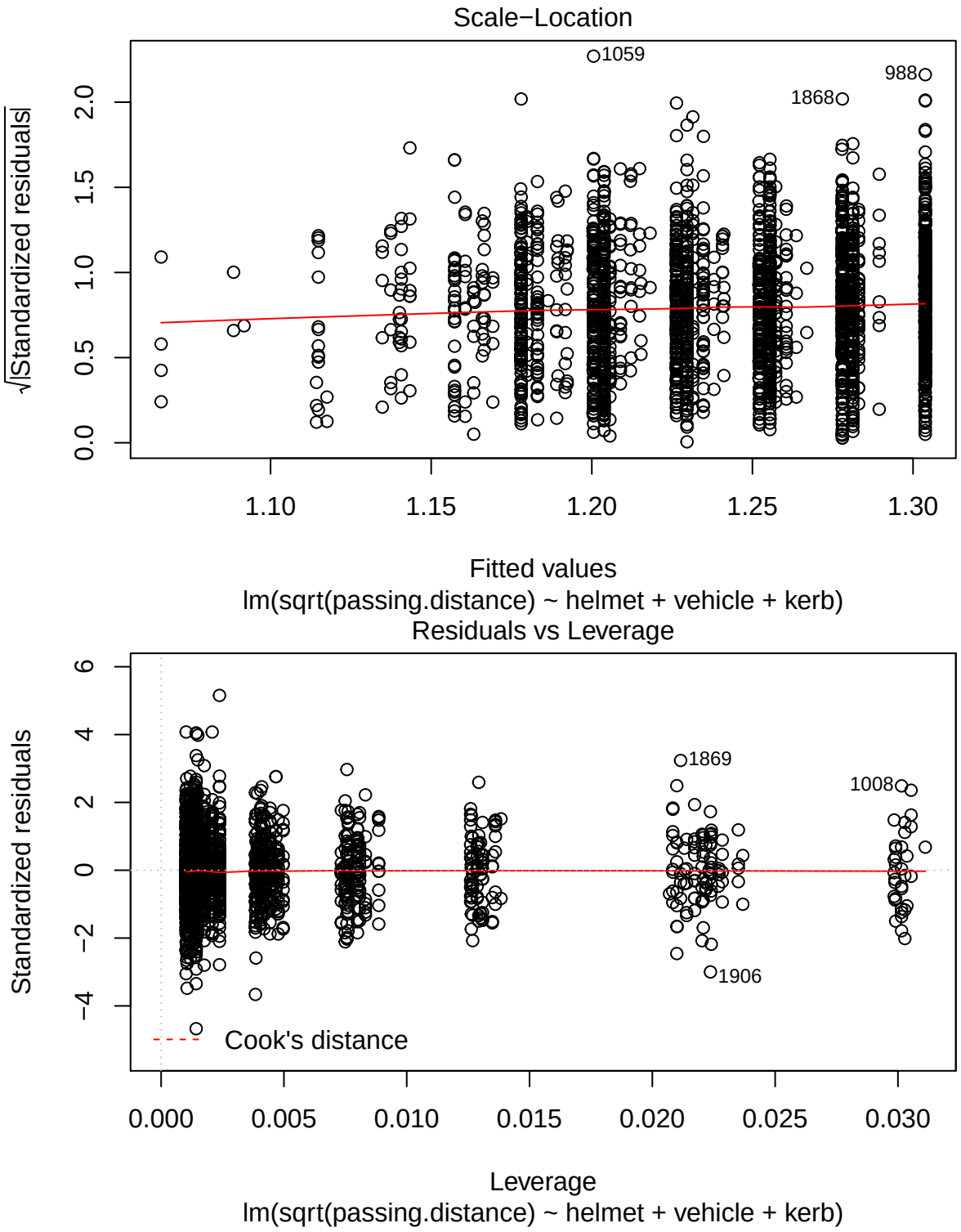
Regression Diagnostics

Plotting a fitted model shows us regression diagnostics:

```
plot(model)
```

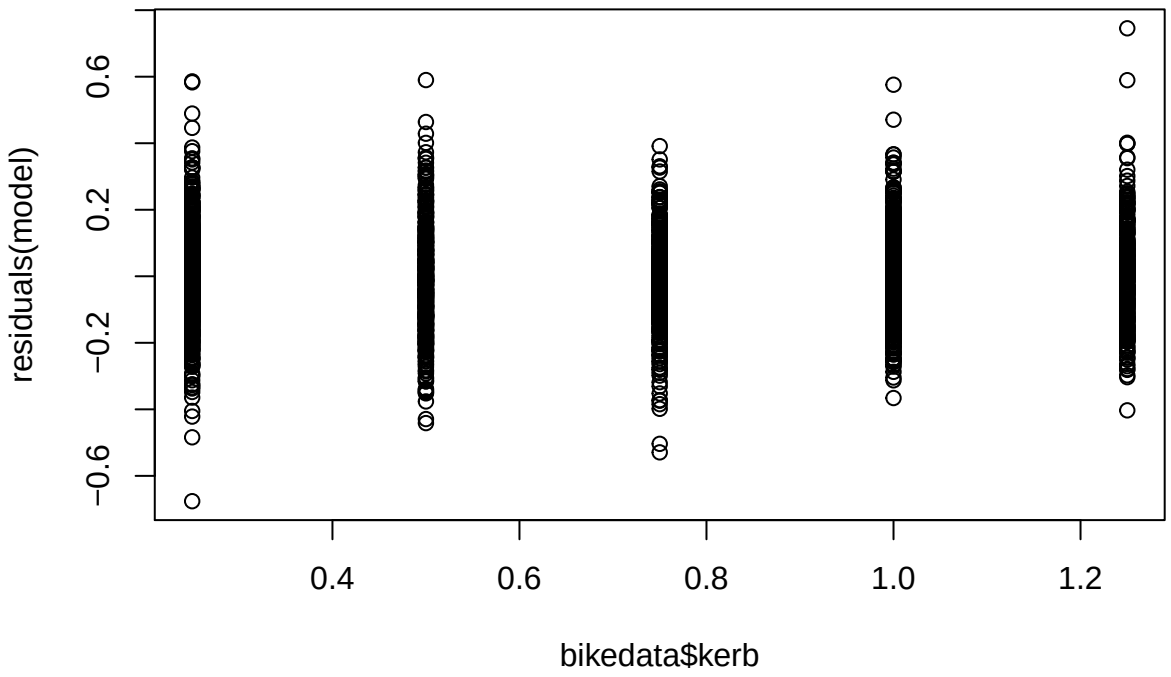

Residuals vs Fitted
Residuals
Fitted values
lm(sqrt(passing.distance) ~ helmet + vehicle + kerb)
1059
1868
988
1.10
1.15
1.20
1.25
1.30
−0.5
0.0
0.5

Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(sqrt(passing.distance) ~ helmet + vehicle + kerb)
1059
1868
988
−3
−2
−1
0
1
2
3
−4
−2
0
2
4
6



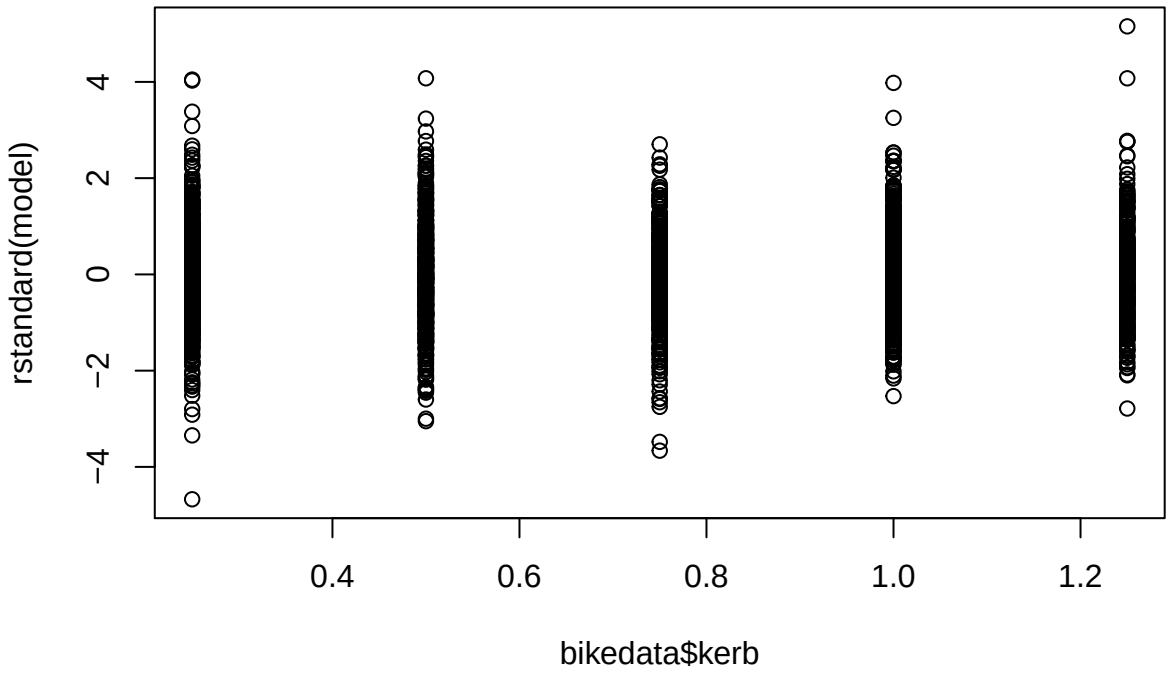
We can extract the raw or standardized residuals with the `residuals` or `rstandard` command. For example, to make scatter plots of residuals versus `kerb`, use

```
plot(bikedata$kerb, residuals(model))
```



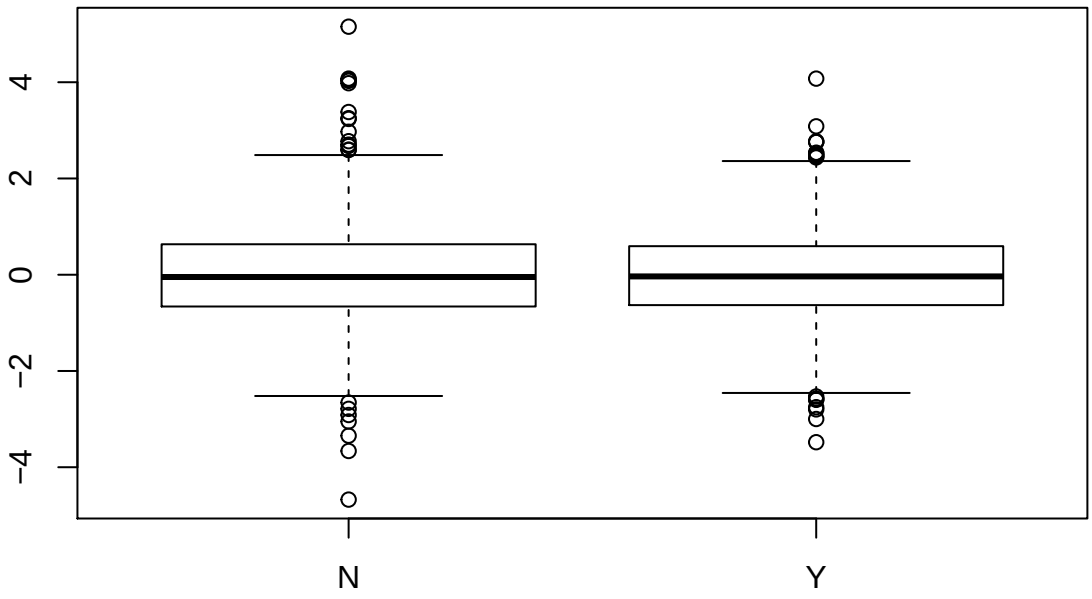
For standardized residuals, use

```
plot(bikedata$kerb, rstandard(model))
```



Here is a boxplots of standardized residual versus `helmet`:

```
boxplot(rstandard(model) ~ bikedata$helmet)
```



Forecasting

We use the `predict` command to forecast the response values for new observations. To use this command, we first make a data frame with the predictors for the new observations, then we pass this data frame and the model to the `predict` command.

Suppose we want to get a prediction `sqrt(passing.distance)` when riding with a helmet, being passed by an SUV, and having a kerb distance of 1.2 meters. In this case, we run the commands

```
newdata <- data.frame(helmet="Y", vehicle="SUV", kerb=1.2)
predict(model, newdata)
```

```
##        1
## 1.183032
```

We can also ask for the standard error of the fit:

```
predict(model, newdata, se.fit = TRUE)
```

```
## $fit
##        1
## 1.183032
##
## $se.fit
## [1] 0.0130666
##
## $df
## [1] 2346
##
## $residual.scale
## [1] 0.1448078
```

We can get a 95% confidence interval for the mean with

```r
predict(model, newdata, interval = "confidence", level = 0.95)
```

```
##        fit      lwr      upr
## 1 1.183032 1.157409 1.208656
```

Here, the reported confidence interval is approximately `(1.157, 1.209)`.

Finally, we can get a prediction interval for the response with

```r
predict(model, newdata, interval = "prediction", level = 0.95)
```

```
##        fit      lwr      upr
## 1 1.183032 0.897914 1.468151
```

Here, the reported prediction interval is approximately `(0.897, 1.469)`.