2 | Understand Your Data

GOOD data is the basis of any sort of regression model, because we use this data to actually construct the model. If the data is flawed, the model will be flawed. It is the old maxim of *garbage in, garbage out.* Thus, the first step in regression modeling is to ensure that your data is reliable. There is no universal approach to verifying the quality of your data, unfortunately. If you collect it yourself, you at least have the advantage of knowing its provenance. If you obtain your data from somewhere else, though, you depend on the source to ensure data quality. Your job then becomes verifying your source's reliability and correctness as much as possible.

2.1 || Missing Values

Any large collection of data is probably incomplete. That is, it is likely that there will be cells without values in your data table. These missing values may be the result of an error, such as the experimenter simply forgetting to fill in a particular entry. They also could be missing because that particular system configuration did not have that parameter available. For example, not every processor tested in our example data had an L2 cache. Fortunately, R is designed to gracefully handle missing values. R uses the notation *NA* to indicate that the corresponding value is not available.

Most of the functions in R have been written to appropriately ignore *NA* values and still compute the desired result. Sometimes, however, you must explicitly tell the function to ignore the NA values. For example, calling the `mean()` function with an input vector that contains NA values causes it to return *NA* as the result. To compute the mean of the input vector while ignoring the *NA* values, you must explicitly tell the function to remove the *NA* values using `mean(x, na.rm=TRUE)` .

2.2 || Sanity Checking and Data Cleaning

Regardless of where you obtain your data, it is important to do some *sanity checks* to ensure that nothing is drastically flawed. For instance, you can check the minimum and maximum values of key input parameters (i.e., columns) of your data to see if anything looks obviously wrong. One of the exercises in Chapter 8 encourages you explore other approaches for verifying your data. R also provides good plotting functions to quickly obtain a visual indication of some of the key relationships in your data set. We will see some examples of these functions in Section 3.1.

If you discover obvious errors or flaws in your data, you may have to eliminate portions of that data. For instance, you may find that the perfor- mance reported for a few system configurations is hundreds of times larger than that of all of the other systems tested. Although it is possible that this data is correct, it seems more likely that whoever recorded the data simply made a transcription error. You may decide that you should delete those results from your data. It is important, though, not to throw out data that looks strange without good justification. Sometimes the most interesting conclusions come from data that on first glance appeared flawed, but was actually hiding an interesting and unsuspected phenomenon. This process of checking your data and putting it into the proper format is often called *data cleaning.*

It also is always appropriate to use your knowledge of the system and the relationships between the inputs and the output to inform your model building. For instance, from our experience, we expect that the clock rate will be a key parameter in any regression model of computer systems per- formance that we construct. Consequently, we will want to make sure that our models include the clock parameter. If the modeling methodology sug- gests that the clock is not important in the model, then using the method- ology is probably an error. We additionally may have deeper insights into the physical system that suggest how we should proceed in developing a model. We will see a specific example of applying our insights about the effect of caches on system performance when we begin constructing more complex models in Chapter 4.

These types of sanity checks help you feel more comfortable that your data is valid. However, keep in mind that it is impossible to prove that your data is flawless. As a result, you should always look at the results of any regression modeling exercise with a healthy dose of skepticism and think carefully about whether or not the results make sense. Trust your intuition. If the results don't feel right, there is quite possibly a problem lurking somewhere in the data or in your analysis.

2.3 || The Example Data

I obtained the input data used for developing the regression models in the subsequent chapters from the publicly available *CPU DB* database [2]. This database contains design characteristics and measured performance results for a large collection of commercial processors. The data was col- lected over many years and is nicely organized using a common format and a standardized set of parameters. The particular version of the database used in this book contains information on 1,525 processors.

Many of the database's parameters (columns) are useful in understand- ing and comparing the performance of the various processors. Not all of these parameters will be useful as predictors in the regression models, how- ever. For instance, some of the parameters, such as the column labeled *Instruction set width*, are not available for many of the processors. Oth- ers, such as the *Processor family*, are common among several processors and do not provide useful information for distinguishing among them. As a result, we can eliminate these columns as possible predictors when we develop the regression model.

On the other hand, based on our knowledge of processor design, we know that the clock frequency has a large effect on performance. It also seems likely that the parallelism-related parameters, specifically, the num- ber of threads and cores, could have a significant effect on performance, so we will keep these parameters available for possible inclusion in the regression model.

Technology-related parameters are those that are directly determined by the particular fabrication technology used to build the processor. The num- ber of transistors and the die size are rough indicators of the size and com- plexity of the processor's logic. The feature size, channel length, and FO4 (fanout-of-four) delay are related to gate delays in the processor's logic. Because these parameters both have a direct effect on how much process- ing can be done per clock cycle and effect the critical path delays, at least some of these parameters could be important in a regression model that describes performance.

Finally, the memory-related parameters recorded in the database are the separate L1 instruction and data cache sizes, and the unified L2 and L3 cache sizes. Because memory delays are critical to a processor's perfor- mance, all of these memory-related parameters have the potential for being important in the regression models.

The reported performance metric is the score obtained from the SPEC CPU integer and floating-point benchmark programs from 1992, 1995, 2000, and 2006 [6–8]. This performance result will be the regression model's output. Note that performance results are not available for every processor running every benchmark. Most of the processors have perfor- mance results for only those benchmark sets that were current when the processor was introduced into the market. Thus, although there are more than 1,500 lines in the database representing more than 1,500 unique pro- cessor configurations, a much smaller number of results are reported for each individual benchmark.

2.4 || Data Frames

The *fundamental* object used for storing tables of data in R is called a *data frame.* We can think of a data frame as a way of organizing data into a large table with a row for each system measured and a column for each parameter. An interesting and useful feature of R is that all the columns in a data frame do not need to be the same data type. Some columns may consist of numerical data, for instance, while other columns contain textual data. This feature is quite useful when manipulating large, heterogeneous data files.

To access the CPU DB data, we first must read it into the R environ- ment. R has built-in functions for reading data directly from files in the csv (comma separated values) format and for organizing the data into data frames. The specifics of this reading process can get a little messy, depend- ing on how the data is organized in the file. We will defer the specifics of reading the CPU DB file into R until Chapter 6. For now,

we will use a function called `extract_data()` , which was specifically written for reading the CPU DB file.

To use this function, copy both the **all-data.csv** and **read-data.R** files into a directory on your computer (you can download both of these files from this book's web site shown on p. ii). Then start the R environment and set the local directory in R to be this directory using the *File -> Change dir* pull-down menu. Then use the *File -> Source R code* pull-down menu to read the **read-data.R** file into R. When the R code in this file completes, you should have six new data frames in your R environment workspace: `int92.dat` , `fp92.dat` , `int95.dat` , `fp95.dat` , `int00.dat` , `fp00.dat` , `int06.dat` , and `fp06.dat` .

The data frame `int92.dat` contains the data from the CPU DB database for all of the processors for which performance results were available for the SPEC Integer 1992 (Int1992) benchmark program. Similarly, fp92.dat contains the data for the processors that executed the Floating-Point 1992 (Fp1992) benchmarks, and so on. I use the `.dat` suffix to show that the corresponding variable name is a data frame.

Simply typing the name of the data frame will cause R to print the en- tire table. For example, here are the first few lines printed after I type `int92.dat` , truncated to fit within the page:

	nperf	perf	clock	threads	cores	...
1	9.662070	68.60000	100	1	1	...
2	7.996196	63.100000	125	1	1	...
3	16.363872	90.72647	166	1	1	...
4	13.720745	82.00000	175	1	1	...
...						

The first row is the header, which shows the name of each column. Each subsequent row contains the data corresponding to an individual processor. The first column is the index number assigned to the processor whose data is in that row. The next columns are the specific values recorded for that parameter for each processor. The function `head(int92.dat)` prints out just the header and the first few rows of the corresponding data frame. It gives you a quick glance at the data frame when you interact with your data.

Table 2.1 shows the complete list of column names available in these data frames. Note that the column names are listed vertically in this table, simply to make them fit on the page.

Table 2.1: The names and definitions of the columns in the data frames containing the data from CPU DB.

Column Number	Column name	Definition
1	(blank)	Processor index number
2	nperf	Normalized performance
3	perf	SPEC performance
4	clock	Clock frequency (MHz)
5	threads	Number of hardware threads available
6	cores	Number of hardware cores available
7	TDP	Thermal design power
8	transistors	Number of transistors on the chip(M)
9	dieSize	The size fo the chip
10	voltage	Nominal operating voltage
11	featureSize	Fabrication feature size
12	channel	Fabrication channel size
13	FO4delay	Fan-out-four delay
14	L1icache	Level 1 instruction cache size
15	L1dcache	Level 1 data cache size
16	L2cache	Level 2 cache size
17	L3cache	Level 3 cache size

2.5 || Accessing a Data Frame

We access the individual elements in a data frame using square brackets to identify a specific cell. For instance, the following accesses the data in the cell in row 15, column 12:

```
int92.dat[15,12]
```

```
## [1] 180
```

We can also access cells by name by putting quotes around the name:

```
int92.dat["71","perf"]
```

```
## [1] 105.1
```

This expression returns the data in the row labeled 71 and the column labeled perf . Note that this is not row 71, but rather the row that contains the data for the processor whose name is 71 .

We can access an entire column by leaving the first parameter in the square brackets empty. For instance, the following prints the value in every row for the column labeled clock :

```
int92.dat[,"clock"]
```

```
##  [1] 100 125 166 175 190 200 225 233 266 275 231 233  99 250 266 291 300
## [18] 333 350 110  60  70  85 101 118  50 100 125  50  80  90 100  48  60
## [35]  64  80  96 125  99 100 120  66  77  66  75  66 100 120 133  66 100
## [52] 100 120 133  60  66  75  90 150 166 180 200 200 250 100 150 175 200
## [69] 100 133 133  75 100 125 150  50  60  75
```

Similarly, this expression prints the values in all of the columns for row 36:

```
int92.dat[36,]
```

```
##       nperf     perf clock threads cores TDP transistors dieSize voltage
## 36 13.07378 79.86399    80       1     1  NA          NA      NA      NA
##    featureSize channel FO4delay L1icache L1dcache L2cache L3cache
## 36        0.75    0.75      270        1       NA      NA      NA
```

The functions nrow() and ncol() return the number of rows and columns, respectively, in the data frame:

```
nrow(int92.dat)
```

```
## [1] 78
```

```
ncol(int92.dat)
```

```
## [1] 16
```

Because R functions can typically operate on a vector of any length, we can use built-in functions to quickly compute some useful results. For ex- ample, the following expressions compute the minimum, maximum, mean, and standard deviation of the perf column in the int92.dat data frame:

```
min(int92.dat[,"perf"])
```

```
## [1] 36.7
```

```
max(int92.dat[,"perf"])
```

```
## [1] 366.857
```

```
mean(int92.dat[,"perf"])
```

```
## [1] 124.2859
```

```
sd(int92.dat[,"perf"])
```

```
## [1] 78.0974
```

This square-bracket notation can become cumbersome when you do a substantial amount of interactive computation within the R environment. R provides an alternative notation using the $ symbol to more easily access a column. Repeating the previous example using this notation:

```
min(int92.dat$perf)
```

```
## [1] 36.7
```

```
max(int92.dat$perf)
```

```
## [1] 366.857
```

```
mean(int92.dat$perf)
```

```
## [1] 124.2859
```

```
sd(int92.dat$perf)
```

```
## [1] 78.0974
```

This notation says to use the data in the column named perf from the data frame named int92.dat . We can make yet a further simplification using the attach function. This function makes the corresponding data frame local to the current workspace, thereby eliminating the need to use the potentially awkward $ or square-bracket indexing notation. The following example shows how this works:

```
attach(int92.dat)
min(perf)
```

```
## [1] 36.7
```

```
max(perf)
```

```
## [1] 366.857
```

```
mean(perf)
```

```
## [1] 124.2859
```

```
sd(perf)
```

```
## [1] 78.0974
```

To change to a different data frame within your local workspace, you must first detach the current data frame:

```
detach(int92.dat)
attach(fp00.dat)
min(perf)
```

```
## [1] 87.54153
```

```
max(perf)
```

```
## [1] 3369
```

```
mean(perf)
```

```
## [1] 1217.282
```

```
sd(perf)
```

```
## [1] 787.4139
```

Now that we have the necessary data available in the R environment, and some understanding of how to access and manipulate this data, we are ready to generate our first regression model.

3 | One-Factor Regression

The simplest linear regression model finds the relationship between one input variable, which is called the *predictor* variable, and the output, which is called the system's *response*. This type of model is known as a *one-factor* linear regression. To demonstrate the regression-modeling process, we will begin developing a one-factor model for the SPEC Integer 2000 (Int2000) benchmark results reported in the CPU DB data set. We will expand this model to include multiple input variables in Chapter 4.

3.1 || Visualize the Data

The first step in this one-factor modeling process is to determine whether or not it *looks* as though a linear relationship exists between the predictor and the output value. From our understanding of computer system design - that is, from our *domain-specific knowledge* - we know that the clock frequency strongly influences a computer system's performance. Consequently, we must look for a roughly linear relationship between the processor's per- formance and its clock frequency. Fortunately, R provides powerful and flexible plotting functions that let us visualize this type relationship quite easily.

This R function call:

```
plot(int00.dat[,"clock"],int00.dat[,"perf"], main="Int2000",xlab="Clock", ylab="Performance")
```

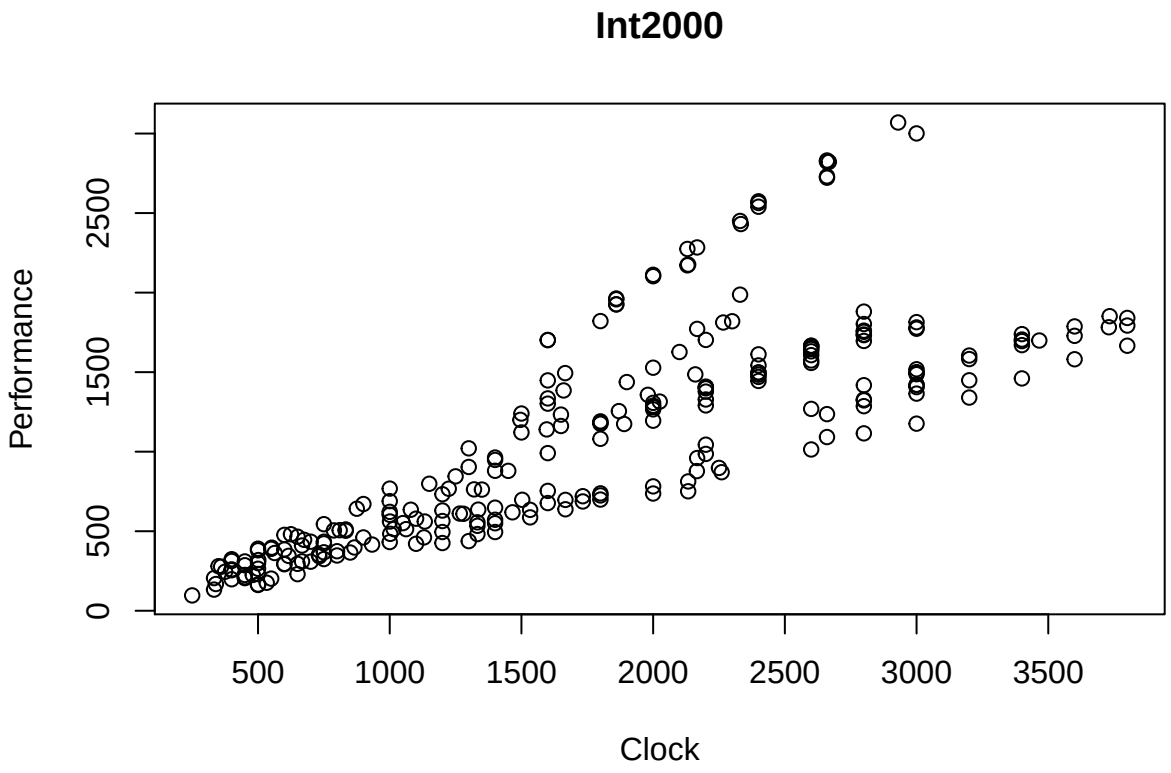

Figure 3.1: A scatter plot of the performance of the processors we tested using the Int2000 benchmark versus the clock frequency.

generates the plot shown in Figure 3.1. The first parameter in this func- tion call is the value we will plot on the x-axis. In this case, we will plot the `clock` values from the `int00.dat` data frame as the independent variable on the x-axis. The dependent variable is the `perf` column from `int00.dat`, which we plot on the y-axis. The function argument `main="Int2000"` provides a title for the plot, while `xlab="Clock"` and `ylab="Performance"` pro vide labels for the x- and y-axes, respectively.

This figure shows that the performance tends to increase as the clock fre- quency increases, as we expected. If we superimpose a straight line on this scatter plot, we see that the relationship between the predictor (the clock frequency) and the output (the performance) is roughly linear. It is not per- fectly linear, however. As the clock frequency increases, we see a larger spread in performance values. Our next step is to develop a regression model that will help us quantify the degree of linearity in the relationship between the output and the predictor.

3.2 || The Linear Model Function

We use regression models to predict a system's behavior by extrapolating from previously measured output values when the system is tested with known input parameter values. The simplest regression model is a straight line. It has the mathematical form:

$$y = a0 + a1x1 \qquad (3.1)$$

where `x1` is the input to the system, `a0` is the y-intercept of the line, `a1` is the slope, and `y` is the output value the model predicts.

R provides the function `lm()` that generates a linear model from the data contained in a data frame. For this one-factor model, R computes the val- ues of `a0` and `a1` using the method of least squares. This method finds the line that most closely fits the measured data by minimizing the dis- tances between the line and the individual data points. For the data frame `int00.dat` , we compute the model as follows:

```
detach(fp00.dat) # objects from int00.dat would be masked otherwise
attach(int00.dat)
int00.lm = lm(perf ~ clock)
```

The first line in this example attaches the `int00.dat` data frame to the current workspace. The next line calls the `lm()` function and assigns the resulting *linear* model object to the variable `int00.lm` . We use the suffix `.lm` to emphasize that this variable contains a linear model. The argument in the `lm()` function, `(perf ~ clock)` , says that we want to find a model where the the predictor `clock` explains the output `perf` . Typing the variable's name, `int00.lm` , by itself causes R to print the ar- gument with which the function `lm()` was called, along with the computed coefficients for the regression model.

```
int00.lm
```

```
##
## Call:
## lm(formula = perf ~ clock)
##
## Coefficients:
## (Intercept)        clock
##     51.7871       0.5863
```

In this case, the y-intercept is `a0` = 51.7871 and the slope is `a1` = 0.5863.

Thus, the final regression model is:

$$perf = 51.7871 + 0.5863 * clock. \qquad (3.2)$$

The following code plots the original data along with the fitted line, as shown in Figure 3.2. The function `abline()` is short for *(a,b)-line*. It plots a line on the active plot window, using the slope and intercept of the linear model given in its argument.

```
plot(clock,perf)
abline(int00.lm)
```

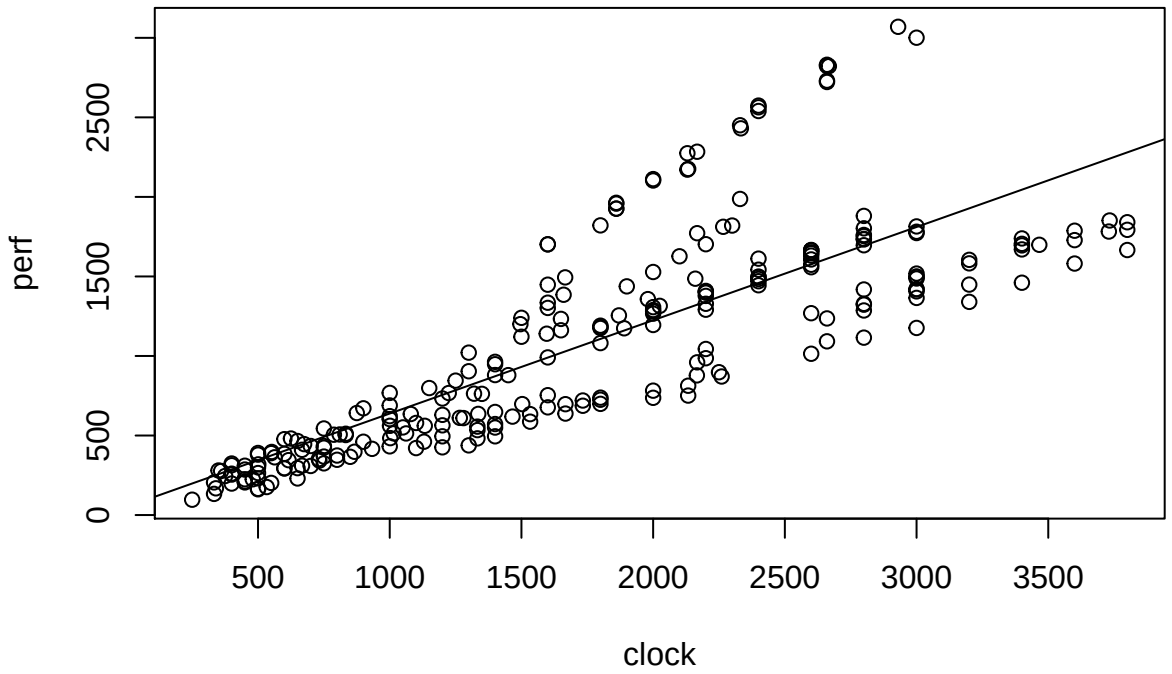

Figure 3.2: The one-factor linear regression model superimposed on the data from Figure 3.1.

3.3 || Evaluating the Quality of the Model

The information we obtain by typing `int00.lm` shows us the regression model's basic values, but does not tell us anything about the model's quality. In fact, there are many different ways to evaluate a regression model's quality. Many of the techniques can be rather technical, and the details of them are beyond the scope of this tutorial. However, the function `summary()` extracts some additional information that we can use to determine how well the data fit the resulting model. When called with the model object `int00.lm` as the argument, `summary()` produces the following information:

```
summary(int00.lm)
```

```
##
## Call:
## lm(formula = perf ~ clock)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -634.61 -276.17  -30.83   75.38 1299.52
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 51.78709   53.31513   0.971    0.332
## clock        0.58635    0.02697  21.741   <2e-16 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 396.1 on 254 degrees of freedom
## Multiple R-squared:  0.6505, Adjusted R-squared:  0.6491
## F-statistic: 472.7 on 1 and 254 DF,  p-value: < 2.2e-16
```

Let's examine each of the items presented in this summary in turn.

```
> summary(int00.lm)
Call:
lm(formula = perf ~ clock)
```

These first few lines simply repeat how the lm() function was called. It is useful to look at this information to verify that you actually called the function as you intended.

```
Residuals:
  Min     1Q      Median    3Q    Max
-634.61 -276.17 -30.83    75.38   1299.52
```

The *residuals* are the differences between the actual measured values and the corresponding values on the fitted regression line. In Figure 3.2, each data point's residual is the distance that the individual data point is above (positive residual) or below (negative residual) the regression line. `Min` is the minimum residual value, which is the distance from the regression line o the point furthest below the line. Similarly, `Max` is the distance from the regression line of the point furthest above the line. `Median` is the median value of all of the residuals. The `1Q` and `3Q` values are the points that mark the first and third quartiles of all the sorted residual values.

How should we interpret these values? If the line is a good fit with the data, we would expect residual values that are normally distributed arounda mean of zero. (Recall that a normal distribution is also called a Gaussian distribution.) This distribution implies that there is a decreasing probability of finding residual values as we move further away from the mean. That is, a good model's residuals should be roughly balanced around and not too far away from the mean of zero. Consequently, when we look at the residual values reported by `summary()` , a good model would tend to have a median value near zero, minimum and maximum values of roughly the same magnitude, and first and third quartile values of roughly the same magnitude. For this model, the residual values are not too far off what we would expect for Gaussian-distributed numbers. In Section 3.4, we present a simple visual test to determine whether the residuals appear to follow a normal distribution.

```
Coefficients:
            Estimate      Std. Error  t value Pr(>|t|)
(Intercept) 51.78709      53.31513    0.971   0.332
clock       0.58635       0.02697     21.741  <2e-16 ***
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

This portion of the output shows the estimated coefficient values. These values are simply the fitted regression model values from Equation 3.2. The `Std. Error` column shows the statistical standard error for each of the coefficients. For a good model, we typically would like to see a standard error that is at least five to ten times smaller than the corresponding coeffi- cient. For example, the standard error for `clock` is 21.7 times smaller than the coefficient value (0.58635/0.02697 = 21.7). This large ratio means that there is relatively little variability in the slope estimate, `a1` . The standard error for the intercept, `a0` , is 53.31513, which is roughly the same as the es- timated value of 51.78709 for this coefficient. These similar values suggest that the estimate of this coefficient for this model can vary significantly.

The last column, labeled `Pr(>|t|)` , shows the probability that the corre- sponding coefficient is not relevant in the model. This value is also known as the significance or p-value of the coefficient. In this example, the prob- ability that `clock` is not relevant in this model is $2 \times 10 - 16$ - a tiny value. The probability that

the intercept is not relevant is 0.332, or about a one-in- three chance that this specific intercept value is not relevant to the model. There is an intercept, of course, but we are again seeing indications that the model is not predicting this value very well.

The symbols printed to the right in this summary - that is, the asterisks, periods, or spaces - are intended to give a quick visual check of the coeffi- cients' significance. The line labeled Signif. codes: gives these symbols' meanings. Three asterisks (*) **means 0 < p 0.001, two asterisks (**) means 0.001 < p 0.01, and so on.

R uses the column labeled t value to compute the p-values and the cor- responding significance symbols. You probably will not use these values directly when you evaluate your model's quality, so we will ignore this column for now.

```
Residual standard error: 396.1 on 254 degrees of freedom
Multiple R-squared: 0.6505, Adjusted R-squared: 0.6491
F-statistic: 472.7 on 1 and 254 DF, p-value: < 2.2e-16
```

These final few lines in the output provide some statistical information about the quality of the regression model's fit to the data. The `Residual standard error` is a measure of the total variation in the residual values. If the residuals are distributed normally, the first and third quantiles of the previous residuals should be about 1.5 times this standard error.

The number of `degrees of freedom` is the total number of measurements or *observations* used to generate the model, minus the number of coeffi- cients in the model. This example had 256 unique rows in the data frame, corresponding to 256 independent measurements. We used this data to pro- duce a regression model with two coefficients: the slope and the intercept. Thus, we are left with (256 - 2 = 254) degrees of freedom.

The `Multiple R-squared` value is a number between 0 and 1. It is a statis- tical measure of how well the model describes the measured data. We com- pute it by dividing the total variation that the model explains by the data's total variation. Multiplying this value by 100 gives a value that we can interpret as a percentage between 0 and 100. The reported *R^2* of 0.6505 for this model means that the model explains 65.05 percent of the data's variation. Random chance and measurement errors creep in, so the model will never explain all data variation. Consequently, you should not ever expect an *R^2* value of exactly one. In general, values of *R^2* that are closer to one indicate a better-fitting model. However, a good model does not necessarily require a large *R^2* value. It may still accurately predict future observations, even with a small *R^2* value.

`The Adjusted R-squared` value is the *R^2* value modified to take into ac- count the number of predictors used in the model. The adjusted *R^2* is always smaller than the *R^2* value. We will discuss the meaining of the ad- justed *R^2* in Chapter 4, when we present regression models that use more than one predictor.

The final line shows the `F-statistic` . This value compares the current model to a model that has one fewer parameters. Because the one-factor model already has only a single parameter, this test is not particularly use- ful in this case. It is an interesting statistic for the multi-factor models, however, as we will discuss later.

3.4 || Residual Analysis

The `summary()` function provides a substantial amount of information to help us evaluate a regression model's fit to the data used to develop that model. To dig deeper into the model's quality, we can analyze some addi- tional information about the observed values compared to the values that the model predicts. In particular, residual analysis examines these residual values to see what they can tell us about the model's quality.

Recall that the residual value is the difference between the actual mea- sured value stored in the data frame and the value that the fitted regression line predicts for that corresponding data point. Residual values greater than zero mean that the regression model predicted a value that was too small compared to the actual measured value, and negative values indicate that the regression model predicted a value that was too large. A model that fits the data well would tend to over-predict as often as it under-predicts. Thus, if

we plot the residual values, we would expect to see them distributed uniformly around zero for a well-fitted model.

The following function calls produce the residuals plot for our model, shown in Figure 3.3.

```
plot(fitted(int00.lm),resid(int00.lm))
```

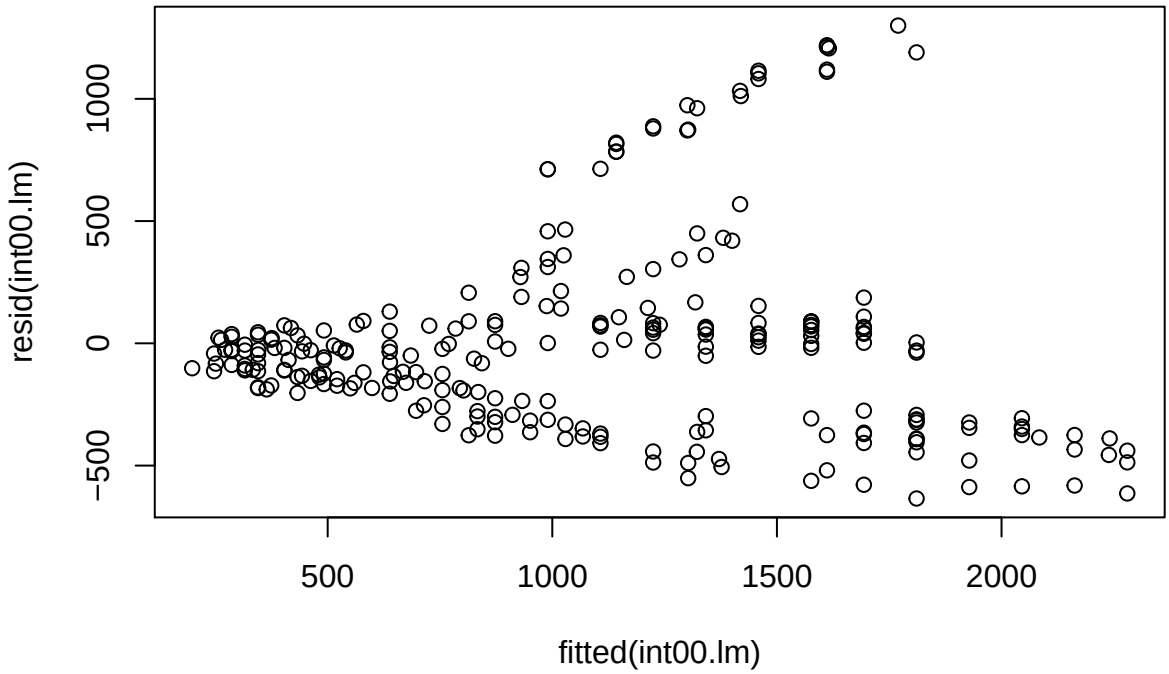

Figure 3.3: The residual values versus the input values for the one-factor model developed using the Int2000 data.

In this plot, we see that the residuals tend to increase as we move to the right. Additionally, the residuals are not uniformly scattered above and below zero. Overall, this plot tells us that using the clock as the sole predictor in the regression model does not sufficiently or fully explain the data. In general, if you observe any sort of clear trend or pattern in the residuals, you probably need to generate a better model. This does not mean that our simple one-factor model is useless, though. It only means that we may be able to construct a model that produces tighter residual values and better predictions.

Another test of the residuals uses the *quantile-versus-quantile*, or Q-Q, plot. Previously we said that, if the model fits the data well, we would expect the residuals to be normally (Gaussian) distributed around a mean of zero. The Q-Q plot provides a nice visual indication of whether the residuals from the model are normally distributed. The following function calls generate the Q-Q plot shown in Figure 3.4:

```
qqnorm(resid(int00.lm))
qqline(resid(int00.lm))
```

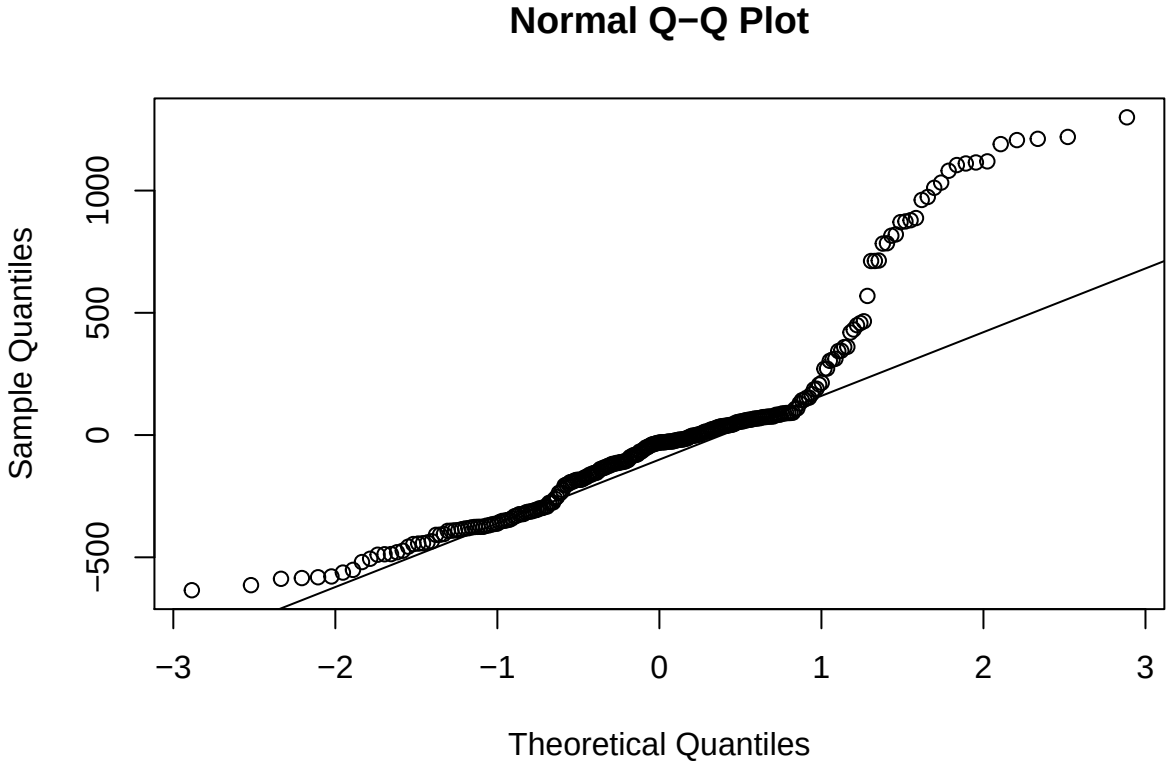

Figure 3.4: The Q-Q plot for the one-factor model developed using the Int2000 data.

If the residuals were normally distributed, we would expect the points plotted in this figure to follow a straight line. With our model, though, we see that the two ends diverge significantly from that line. This behavior indicates that the residuals are not normally distributed. In fact, this plot suggests that the distribution's tails are "heavier" than what we would ex- pect from a normal distribution. This test further confirms that using only the clock as a predictor in the model is insufficient to explain the data.

Our next step is to learn to develop regression models with multiple input factors. Perhaps we will find a more complex model that is better able to explain the data