Chapter 11

Data Processing

In many cases, data is available in a form that makes its analysis inconvenient. Some frequent situations are listed below.

Missing data. Some of the measurements are not available due to data corruption or difficulty in obtaining the data.

Outliers. Some of the measurements are highly atypical of the data distribution.

Skewed data. The data is highly skewed making its visualization and analysis difficult.

In the first three sections we discuss such situations and how to handle them. In the final section we discuss how to manipulate data in general, and specifically how to manipulate data in R, using the `reshape2` and `plyr` packages and in Python using the `pandas` module.

11.1 Missing Data

Data may be missing for a variety of reasons. Perhaps it was corrupted during its transfer or storage. Perhaps some instances in the data collection process were skipped due to difficulty or price associated with obtaining the data. Or perhaps the data was simply unavailable for some other reason.

Denoting n data instances by $\boldsymbol{x}^{(i)}, i = 1, \ldots, n$ (corresponding for example to dataframe rows), missing data implies that for each $i = 1, \ldots, n$ we have a set $A_i \subset \{1, \ldots, n\}$ of indices for which the measurements are missing (A may potentially be the empty set in which case no data is missing). In other words, $x_j^{(i)}$ (the j variable of the i sample) is missing if $j \in A_i$.

Missing data is a general phenomenon. A few specific examples appear below.

- Recommendation systems recommend to users items from a catalog based on historical user rating. Often, there are a lot of items in the catalog and

each user typically indicates their preference on only a small subset of them (for example by assigning 1-5 stars to previously seen movies).

- In longitudinal studies some of the subjects may not be able to attend each of the surveys throughout the study period. The study organizers may also have lost contact with some of the subjects, in which case all measurements beyond a certain time point are missing.
- In sensor data, some of the measurements may be missing due to sensor failure, battery discharge, or electrical interference.
- In user surveys, users may choose to not respond to some of the questions for privacy reasons.

If the probability of an observation being missing does not depend on observed or unobserved measurements we say that it is missing completely at random (MCAR). For example, in the case of users rating movies using 1-5 stars, we consider ratings of specific movies as dataframe columns and ratings associated with specific users as dataframe rows. Since some movies are more popular than others, the probability of missingness depends on the movie title as well as the movie rating, which violates the MCAR definition.

A more relaxed concept is data missing at random (MAR). This occurs when given the observed data, the probability that data is missing does not depend on the unobserved data. Consider, for example, a survey recording gender, race, and income. Out of the three questions, gender and race are not very objectionable questions, so we assume for now that the survey respondents answer these questions fully. The income question is more sensitive and users may choose to not respond to for privacy reasons. The tendency to report income or to not report income typically varies from person to person. If it only depends on gender and race, then the data is MAR. If the decision whether to report income or not depends also on other variables that are not in the dataframe (such as age or profession), the data is not MAR.

Some data analysis techniques are specifically designed to allow for missing data. In general, however, most methods are designed to work with fully observed data. Below are some general ways to convert missing data to non-missing data.

- Remove all data instances (for example dataframe rows) containing missing values.
- Replace all missing entries with a substitute value, for example the mean of the observed instances of the missing variable.
- Estimate a probability model for the missing variable and replace the missing value with one or more samples from that probability model.

In the case of MCAR, all three techniques above are reasonable in that they may not introduce systematic errors. In the more likely case of MAR or non-MAR data the methods above may introduce systematic bias into the data analysis process.

11.1.1 Missing Data in R

R represents missing data using the `NA` symbol which stands for not available. This is different from impossible values, represented by `NaN` (not a number). The function `is.na` returns a data structure having `TRUE` values where the corresponding data is missing and `FALSE` otherwise. The function `complete.cases` returns a vector whose components are `FALSE` for all samples (dataframe rows) containing missing values and `TRUE` otherwise. The function `na.omit` returns a new dataframe omitting all samples (dataframe rows) containing missing values.

Many data analysis functions fail on data containing missing values and simply return `NA` value. This is intended as a warning to the programmer that the data contains missing values. Some functions have an `na.rm` argument, which if set to `TRUE` changes the function behavior so that it proceeds to operate on the supplied data after removing all dataframe rows with missing values.

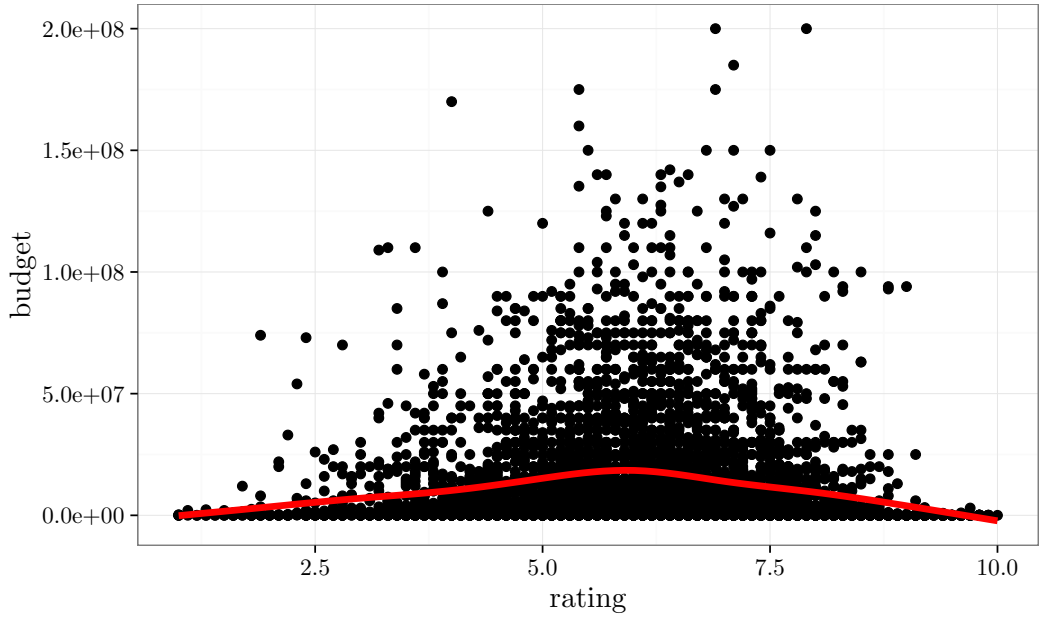
The code below analyzes the dataframe `movies` in the `ggplot2movies` package, which contains 24 attributes (genre, year, budget, user ratings, etc.) for 58788 movies obtained from the website http://www.imdb.com with some missing values. Type `help(movies)` after loading the package `ggplot2movies` for more information.

```
library(ggplot2movies)
names(movies)
##  [1] "title"       "year"
##  [3] "length"      "budget"
##  [5] "rating"      "votes"
##  [7] "r1"          "r2"
##  [9] "r3"          "r4"
## [11] "r5"          "r6"
## [13] "r7"          "r8"
## [15] "r9"          "r10"
## [17] "mpaa"        "Action"
## [19] "Animation"   "Comedy"
## [21] "Drama"       "Documentary"
## [23] "Romance"     "Short"
# display 20 rows, 6 first columns
movies[9000:9020, 1:6]
##                                  title year
## 9000                         Cat People 1982
## 9001 Cat Swallows Parakeet and Speaks! 1996
## 9002        Cat That Hated People, The 1948
## 9003                 Cat and Dupli-cat 1967
## 9004           Cat and the Canary, The 1927
## 9005           Cat and the Canary, The 1939
## 9006           Cat and the Canary, The 1979
## 9007           Cat and the Fiddle, The 1934
## 9008         Cat and the Mermouse, The 1949
## 9009         Cat from Outer Space, The 1978
## 9010                   Cat in the Cage 1978
```

```
## 9011               Cat in the Hat, The 2003
## 9012             Cat on a Hot Tin Roof 1958
## 9013               Cat with Hands, The 2001
## 9014                Cat's Bad Hair Day 2004
## 9015                      Cat's Cradle 1959
## 9016                         Cat's Eye 1985
## 9017                         Cat's Eye 1997
## 9018                Cat's Me-Ouch, The 1965
## 9019                   Cat's Meow, The 2001
## 9020                    Cat's Out, The 1931
##      length    budget rating votes
## 9000    118  18000000    5.8  2862
## 9001     76        NA    5.7     7
## 9002      8        NA    7.9    75
## 9003      7        NA    6.5    24
## 9004     82        NA    7.2   178
## 9005     72        NA    7.6   216
## 9006     89        NA    5.3   156
## 9007     88        NA    6.8    52
## 9008      8        NA    7.0    40
## 9009    104        NA    5.2   583
## 9010     98        NA    2.7    22
## 9011     82 109000000    3.2  4997
## 9012    108   3000000    7.8  4870
## 9013      4        NA    7.9    92
## 9014     15        NA    8.3     7
## 9015      6        NA    6.3    57
## 9016     94   7000000    5.7  2301
## 9017     91        NA    4.8    30
## 9018      6        NA    6.6    26
## 9019    114        NA    6.5  2195
## 9020      7        NA    8.9     7
mean(movies$length)
## [1] 82.33788
mean(movies$budget)
## [1] NA
mean(movies$budget, na.rm = TRUE)
## [1] 13412513
mean(is.na(movies$budget))
## [1] 0.9112914
```

Below, we create a new dataframe that is identical to the `movies` dataframe with the exception that rows with missing values are removed. Then, we graph movie budget vs. average movie rating to examine the relationship between the two variables. The red line below is a smoothing curve showing the average tendency (see Section 10.8 for more details).

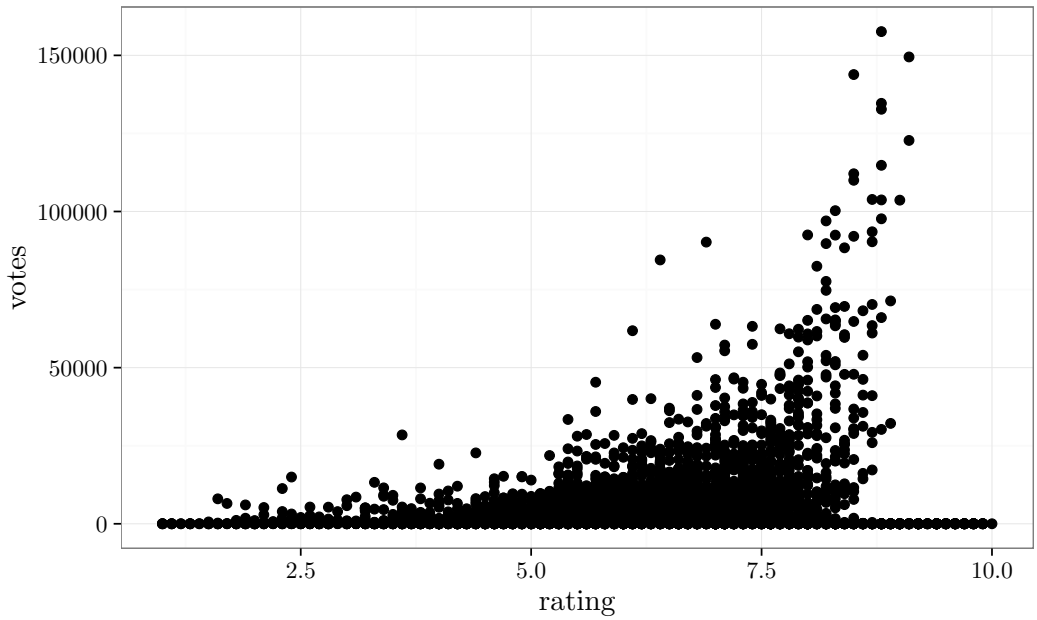
```
moviesNoNA = na.omit(movies)
qplot(rating, budget, data = moviesNoNA, size = I(1.2)) +
  stat_smooth(color = "red", size = I(2), se = F)
```



The graph above shows that there is a high variation in budget, but the general trend is that budget increases with ratings up to a certain point (around 6) and then it decreases. Apparently, high budget movies tend to have decent average IMDB ratings in the range of 5-7.5. The lowest ratings and surprisingly the highest ratings tend to be associated with low-budget movies.

Below we graph total number of IMDB votes vs. average movie rating.

```
moviesNoNA = na.omit(movies)
qplot(rating, votes, data = moviesNoNA, size = I(1.2))
```



The figure above shows that the number of votes (which can be used as a surrogate for popularity) tend to increase as the average rating increase. We also see that the spread in the number of votes increases with the average rating. Finally, it is surprising that the movies featuring the highest average ratings have a very small number of votes. A possible explanation is that when there are only a few votes it is easier to reach very high (or low) average ratings. Consider for example an extreme case where a minor movie is given very high votes by a small circle of fans. In the case of a blockbuster movie with thousands of ratings, opinions vary more and it is much harder to maintain a very high average rating.

When interpreting the graphs above, we should also consider the fact that users tend to see movies that they think they will like, and thus the observed ratings tend to be higher than ratings gathered after showing users random movies.

11.1.2 Missing Data in Python

The `pandas` module in Python provides support for dataframes with missing values. Specifically, `NaN` is used to represent missing values in both floating point and non-floating point arrays. In addition, the value `None` can also be used to represent missing values in arrays containing objects.

Pandas provides the following functions for handling missing values: `isnull` returns a boolean array marking whether entries are missing or not; `notnull` returns the boolean negation of `isnull`; `dropna` drops rows or columns containing missing values; `filna` fills-in missing values.

The code below defines a small dataframe with missing values, and demonstrates the `isnull` method.

```
import numpy as np
import pandas as pd
```

```
data = pd.DataFrame([[1, 2, np.nan],
                     [3, np.nan, 4],
                     [1, 2, 3]])
print("D:\n" + str(data))
print("pd.isnull(D):\n" + str(pd.isnull(data)))
## D:
##    0    1    2
## 0  1  2.0  NaN
## 1  3  NaN  4.0
## 2  1  2.0  3.0
## pd.isnull(D):
##        0      1      2
## 0  False  False   True
## 1  False   True  False
## 2  False  False  False
```

The code below demonstrates the `dropna` method.

```
import numpy as np
import pandas as pd
data = pd.DataFrame([[1, 2, np.nan],
                     [3, np.nan, 4],
                     [1, 2, 3]])
# drop rows
print("data.dropna():\n" + str(data.dropna()))
# drop columns
print("data.dropna(axis = 1):\n" + str(data.dropna(axis = 1)))
## data.dropna():
##    0    1    2
## 2  1  2.0  3.0
## data.dropna(axis = 1):
##    0
## 0  1
## 1  3
## 2  1
```

Above, we used the optional `axis` argument to drop columns instead of rows. The optional `thresh` argument can be used so that only rows (or columns) with more than a certain number of observations will be retained.

Calling the `fillna` function with a numeric argument replaces missing entries with that value. Passing an optional `dict` object that maps column indices to values replaces missing entries with column-specific constants. Alternatively, the `dict` object can be replaced with an array containing column specific fill-in values. By default, `fillna` returns a new dataframe with filled-in values and does not modify the original dataframe. The optional boolean argument `inplace` can be used to modify that behavior such that the original dataframe is modified.

```
import numpy as np
import pandas as pd
data = pd.DataFrame([[1, 2, np.nan],
                     [3, np.nan, 4],
                     [1, 2, 3]])
print("data:\n" + str(data))
# fill-in missing entries with column means
print("data.fillna(data.mean()):\n" + str(data.fillna(data.mean())))
## data:
##    0    1    2
## 0  1  2.0  NaN
## 1  3  NaN  4.0
## 2  1  2.0  3.0
## data.fillna(data.mean()):
##    0    1    2
## 0  1  2.0  3.5
## 1  3  2.0  4.0
## 2  1  2.0  3.0
```

Many dataframe functions contain an optional `fill_value` argument that can be used to fill-in missing values during the operation of the function.

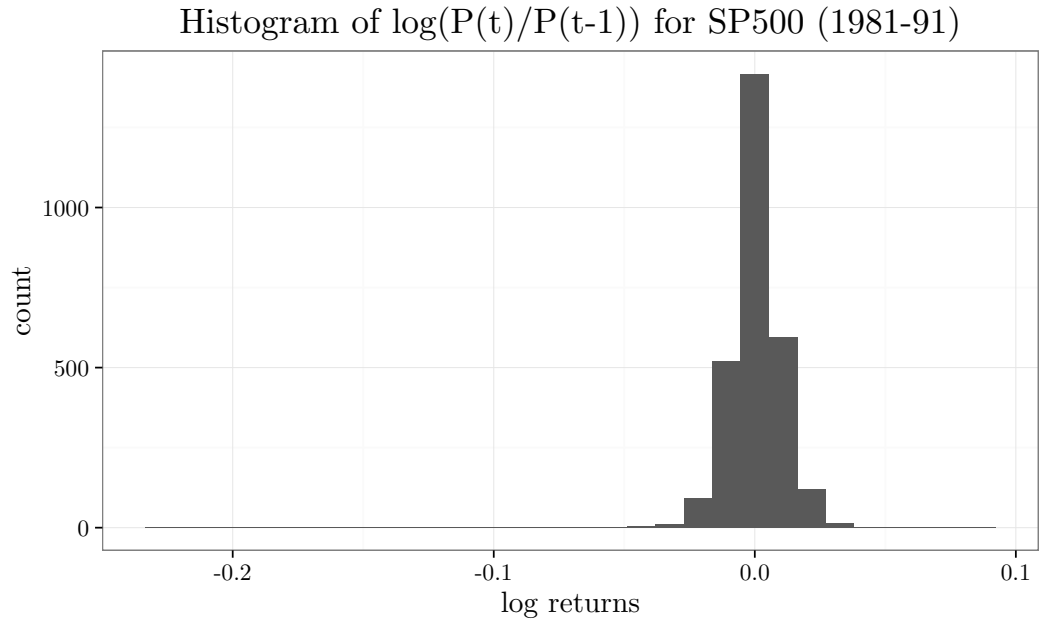
11.2 Outliers

There are two different definitions for outliers. The first considers outliers as corrupted values. That is the case, for example, with human errors during a manual process of entering measurements in a spreadsheet. The second definition considers outliers to be non-corrupt values, but nevertheless are substantially unlikely given our modeling assumptions.

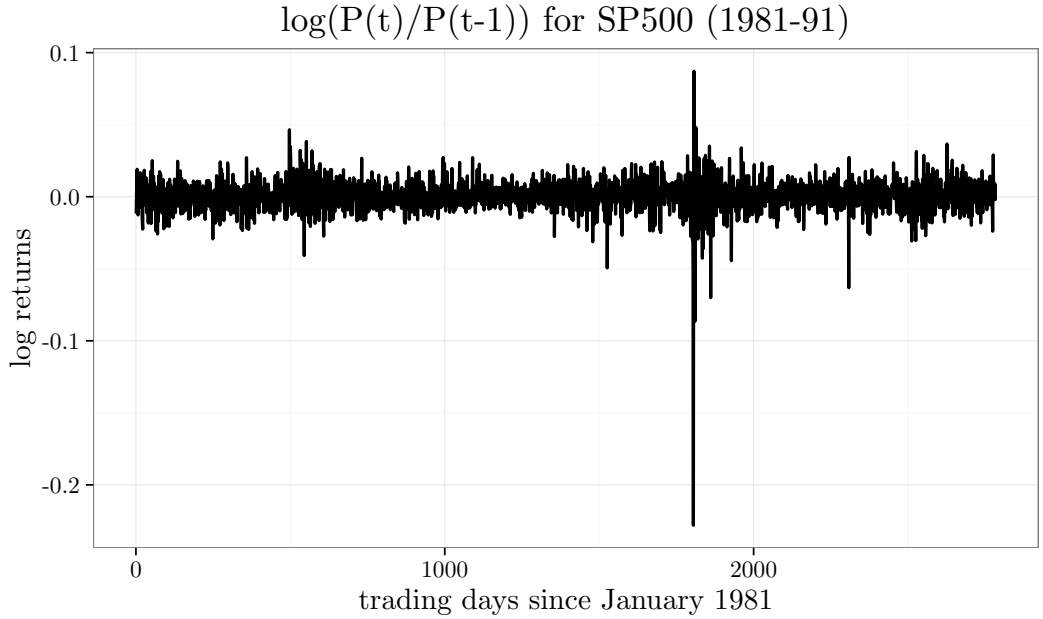
Data analysis based on outliers may result in drastically wrong conclusions. This is pretty clear in the case of corrupted outliers. But it may also be the case with the second definition of outliers, especially when the model used in the data analysis does not account for the extreme observations.

An example for the second outlier definition is the Black Monday stock crash on October 19, 1987, when the Dow Jones Industrial Average lost 22% in one day. The graphs below plot the histogram of log-returns and then the log-returns of the S&P 500 stock index between 1981 and 1991. Log returns is the quantity $\log(P_t/P_{t-1})$, where P_t is the S&P 500 index on day t. A log return close to 0 (return close to 1) indicates no or little daily movement in the index price.

```
library(Ecdat)
data(SP500, package = 'Ecdat')
qplot(r500,
      main = "Histogram of log(P(t)/P(t-1)) for SP500 (1981-91)",
      xlab = "log returns",
      data = SP500)
```




```
qplot(seq(along = r500),
      r500,
      data = SP500,
      geom = "line",
      xlab = "trading days since January 1981",
      ylab = "log returns",
      main = "log(P(t)/P(t-1)) for SP500 (1981-91)")
```



Note how the sharp Black Monday decline barely registers in the histogram of log-returns, but is clearly visible in the log-return graph.

Some models are sensitive to outliers and building such models based on data with outliers can lead to drastically inaccurate predictions. On the other hand, removing outliers is tricky as the resulting model may conclude that future outliers are unlikely to occur.

Robustness describes a lack of sensitivity of data analysis procedures to outliers. An example for a non-robust procedure is computing the mean of n numbers. Assuming a symmetric distribution of samples around 0, we expect the mean to be zero, or at least close to it. But, the presence of a single outlier (very positive value or very negative value) may substantially affect the mean calculation and drive it far away from zero, even for large n.

An example for a robust data analysis procedure is the median, which will not be affected by a single outlier even if it has extreme value. We illustrate this with a hypothetical data of $n+1$ values $\{x^{(1)}, \ldots, x^{(n)}, e\}$, with median b and mean a. The value $e > \max(x^{(1)}, \ldots, x^{(n)})$ is a single high-valued outlier. Fixing the n observations and increasing e to infinity the median will remain constant at b, while the mean would grow to infinity together with the median. This happens regardless of the value of n. In other words, no matter the size of the dataset, a single extreme outlier will affect the mean in a substantial way but will not affect the median.

Three popular techniques for dealing with outliers are listed below.

Truncating. Remove all values deemed as outliers.

Winsorization. Shrink outliers to border of main part of data. One special case of this is to replace outliers with the most extreme of the remaining values.

Robustness. Analyze the data using a robust procedure.

We assume below that a value is considered an outlier if it is below the α percentile or above the $100 - \alpha$ percentile for some small $\alpha > 0$. In many cases, we assume that the data follows a symmetric distribution, in which case the rule above corresponds to being more than c standard deviations away from the mean. The problem is that since the data contains outliers, the estimates of standard deviation or percentiles based on that data is likely to be corrupted as well. This chicken-and-egg problem can be sidelined by computing the standard deviation or percentiles after removing the most extreme values.

The R code below removes outliers. We sample values from a Gaussian distribution and then overwrite the first entry with a single outlier. We then sort the data, exclude the smallest and largest values, compute the standard deviation, and remove the values beyond a certain multiple of the standard deviation values.

```
original_data = rnorm(20)
original_data[1] = 1000
sorted_data = sort(original_data)
filtered_data = original_data[3:18]
```

```
lower_limit = mean(filtered_data) - 5 * sd(filtered_data)
upper_limit = mean(filtered_data) + 5 * sd(filtered_data)
not_outlier_ind = (lower_limit < original_data) &
  (original_data < upper_limit)
print(not_outlier_ind)
##  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [7]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [19]  TRUE  TRUE
data_w_no_outliers = original_data[not_outlier_ind]
```

The R code below winsorizes data containing an outlier. It uses the function `winsorize` from the `robustHD` package.

```
library(robustHD)
original_data = c(1000, rnorm(10))
print(original_data)
##  [1] 1000.0000000   -0.2925366    1.8952183
##  [4]    0.2895131    0.2843515   -0.5394700
##  [7]   -0.2802351   -0.8678683    0.1259644
## [10]    0.6260071   -0.3043207
print(winsorize(original_data))
##  [1]  1.4018458 -0.2925366  1.4018458
##  [4]  0.2895131  0.2843515 -0.5394700
##  [7] -0.2802351 -0.8678683  0.1259644
## [10]  0.6260071 -0.3043207
```

11.3 Data Transformations

11.3.1 Skewness and Power Transformation

In many cases, data is drawn from a highly-skewed distribution that is not well described by one of the common statistical distributions. In some of these cases, a simple transformation may map the data to a form that is well described by common distributions, such as the Gaussian or Gamma distributions (see TAOD volume 1, Chapter 3). A suitable model can then be fitted to the transformed data (if necessary, predictions can be made on the original scale by inverting the transformation).

Power transformations are a family of data transformations for non-negative values (parameterized by $\lambda \in \mathbb{R}$), defined as follows.

$$f_\lambda(x) = \begin{cases} (x^\lambda - 1)/\lambda & \lambda > 0 \\ \log x & \lambda = 0 \\ -(x^\lambda - 1)/\lambda & \lambda < 0 \end{cases} \qquad x > 0, \quad \lambda \in \mathbb{R}. \tag{11.1}$$

The reason for the algebraic form $(x^\lambda - 1)/\lambda$ rather than the simpler x^λ is that the former choice makes $f_\lambda(x)$ continuous in λ as well as in x. The minus sign in the last case ensures that the transformation does not re-order the data points (taking negative powers reverses ordering).

The power transformations can also be used to transform negative data by adding a number large enough so all values are non-negative and then proceeding according to the definition above (11.1).

Intuitively, the power transform maps x to x^λ, up to multiplication by a constant and addition of a constant. This mapping is convex for $\lambda > 1$ and concave for $\lambda < 1$. A choice of $\lambda < 1$ removes right-skewness (data has a heavy tail to the right) with smaller values of λ resulting in a more aggressive removal of skewness. Similarly, a choice of $\lambda > 1$ removes left-skewness.

One way to select λ is to try different values, graph the resulting histograms, and select one of them. There are also more sophisticated methods for selecting λ based on the maximum likelihood method.

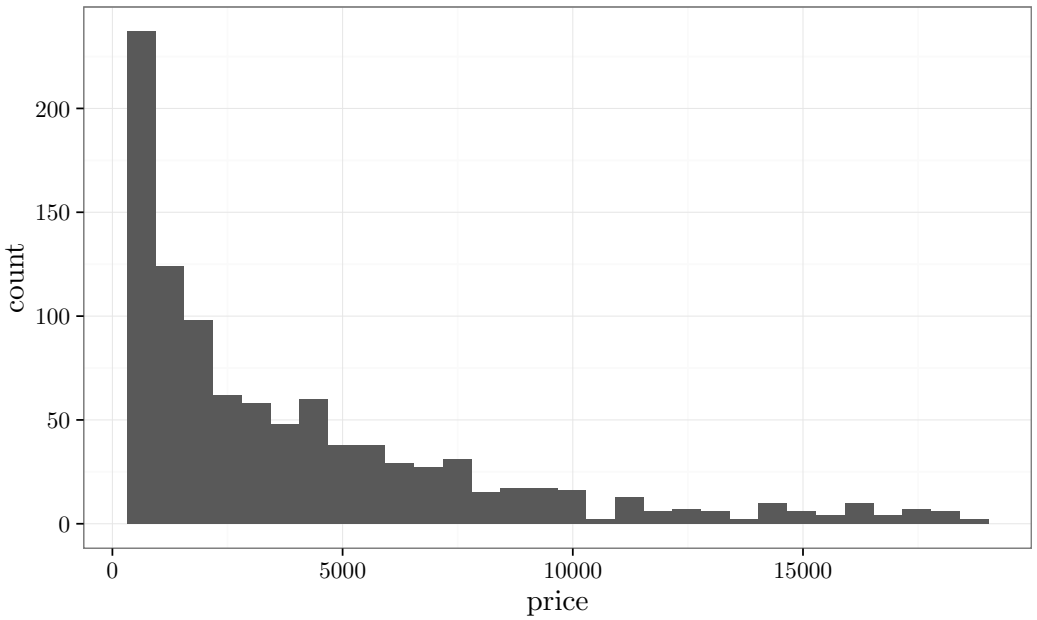
In the example below, we consider the `diamonds` data from the `gpplot2` package, which contains several attributes (cut, carat, price, color, etc.) for 53,940 diamonds.

```
head(diamonds)
##   carat       cut color clarity depth table
## 1  0.23     Ideal     E     SI2  61.5    55
## 2  0.21   Premium     E     SI1  59.8    61
## 3  0.23      Good     E     VS1  56.9    65
## 4  0.29   Premium     I     VS2  62.4    58
## 5  0.31      Good     J     SI2  63.3    58
## 6  0.24 Very Good     J    VVS2  62.8    57
##   price    x    y    z
## 1   326 3.95 3.98 2.43
## 2   326 3.89 3.84 2.31
## 3   327 4.05 4.07 2.31
## 4   334 4.20 4.23 2.63
## 5   335 4.34 4.35 2.75
## 6   336 3.94 3.96 2.48
summary(diamonds)
##      carat               cut
##  Min.   :0.2000   Fair      : 1610
##  1st Qu.:0.4000   Good      : 4906
##  Median :0.7000   Very Good:12082
##  Mean   :0.7979   Premium   :13791
##  3rd Qu.:1.0400   Ideal     :21551
##  Max.   :5.0100
##
##  color        clarity          depth
##  D: 6775   SI1    :13065   Min.   :43.00
##  E: 9797   VS2    :12258   1st Qu.:61.00
##  F: 9542   SI2    : 9194   Median :61.80
##  G:11292   VS1    : 8171   Mean   :61.75
```

```
##  H: 8304   VVS2   : 5066   3rd Qu.:62.50
##  I: 5422   VVS1   : 3655   Max.   :79.00
##  J: 2808   (Other): 2531
##      table           price
##  Min.   :43.00   Min.   :  326
##  1st Qu.:56.00   1st Qu.:  950
##  Median :57.00   Median : 2401
##  Mean   :57.46   Mean   : 3933
##  3rd Qu.:59.00   3rd Qu.: 5324
##  Max.   :95.00   Max.   :18823
##
##        x                y
##  Min.   : 0.000   Min.   : 0.000
##  1st Qu.: 4.710   1st Qu.: 4.720
##  Median : 5.700   Median : 5.710
##  Mean   : 5.731   Mean   : 5.735
##  3rd Qu.: 6.540   3rd Qu.: 6.540
##  Max.   :10.740   Max.   :58.900
##
##        z
##  Min.   : 0.000
##  1st Qu.: 2.910
##  Median : 3.530
##  Mean   : 3.539
##  3rd Qu.: 4.040
##  Max.   :31.800
##
```

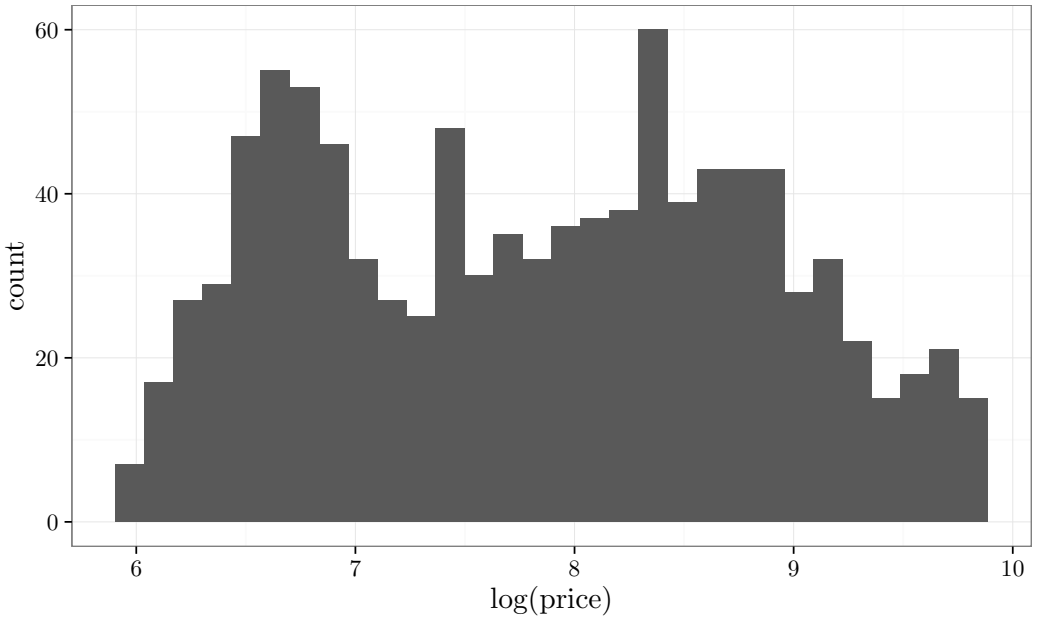
We graph the price histogram of a random subset of 1000 diamonds, obtained by sampling rows from the original dataframe.

```
diamondsSubset = diamonds[sample(dim(diamonds)[1], 1000),]
qplot(price, data = diamondsSubset)
```



The histogram above is skewed to the right and is not very informative. On the other hand, the transformed histogram below reveals an interesting multimodal distribution in the log-scale.

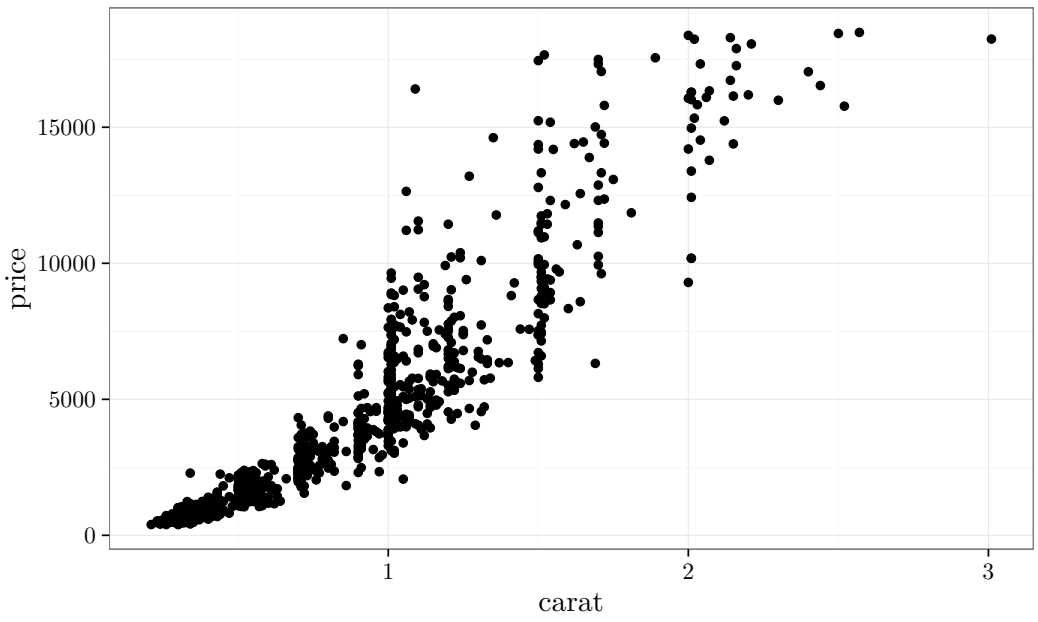
```
qplot(log(price), size = I(1), data = diamondsSubset)
```



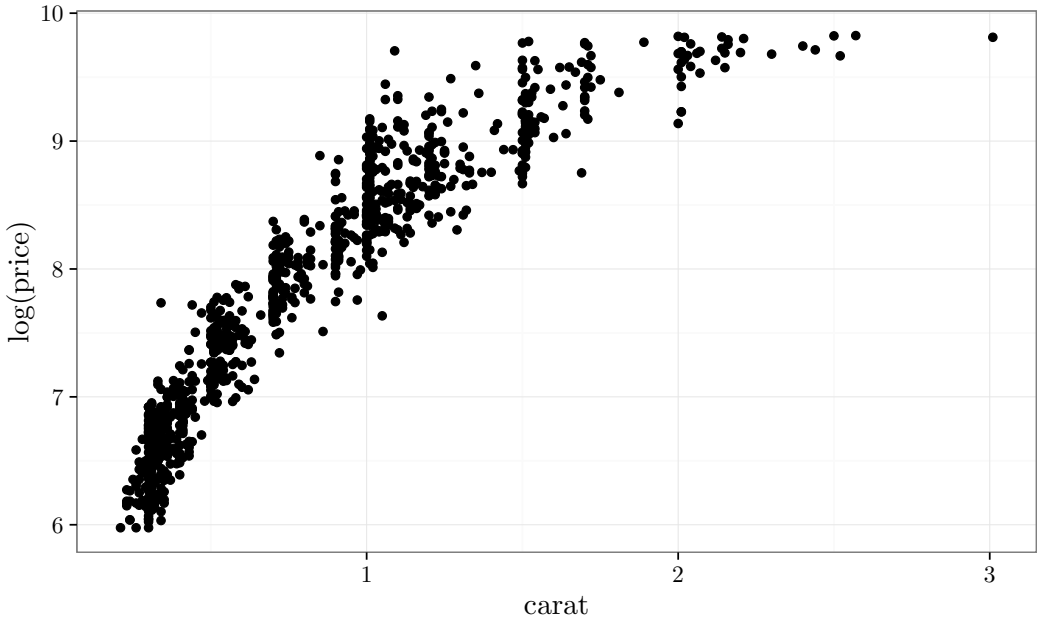
Power transformations are useful also for examining the relationship between two or more data variables. The following plot shows the relationship between diamond price and diamond carat (weight). It is hard to draw much information from that plot beyond the fact that there is a non-linear increasing trend. Trans-

forming both variables using a logarithm shows a striking linear relationship on a log-log scale.

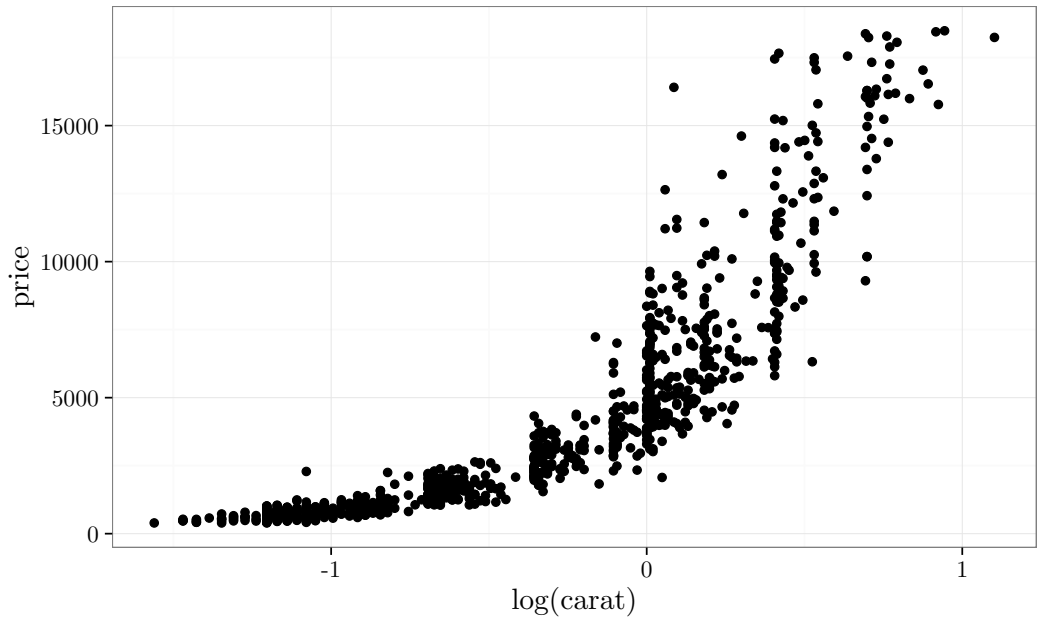
```
qplot(carat,
      price,
      size = I(1),
      data = diamondsSubset)
```



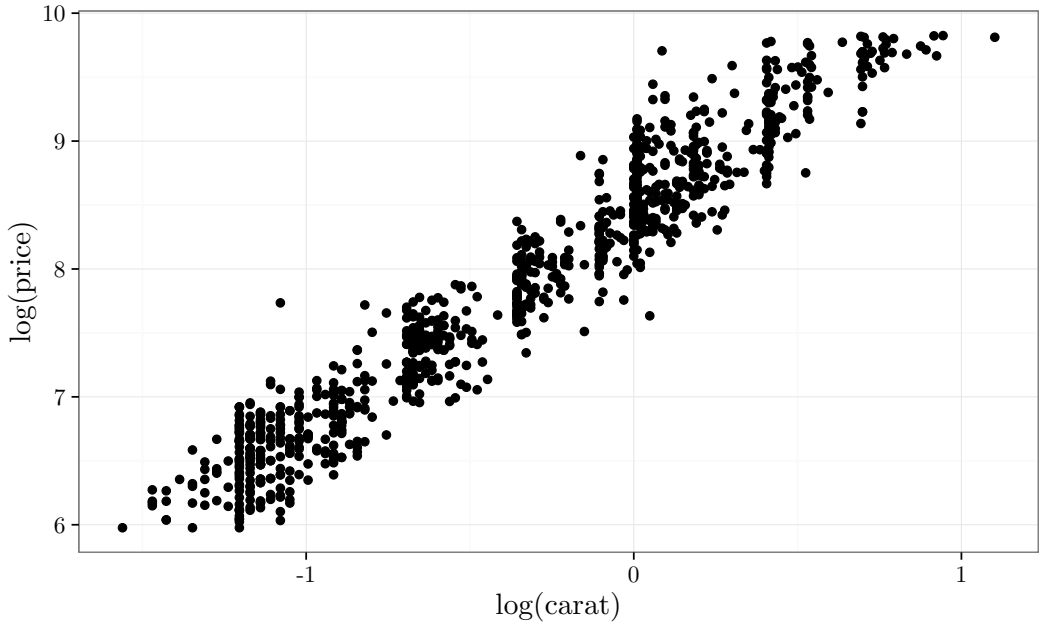

```
qplot(carat,
      log(price),
      size = I(1),
      data = diamondsSubset)
```




```
qplot(log(carat),
      price,
      size = I(1),
      data = diamondsSubset)
```

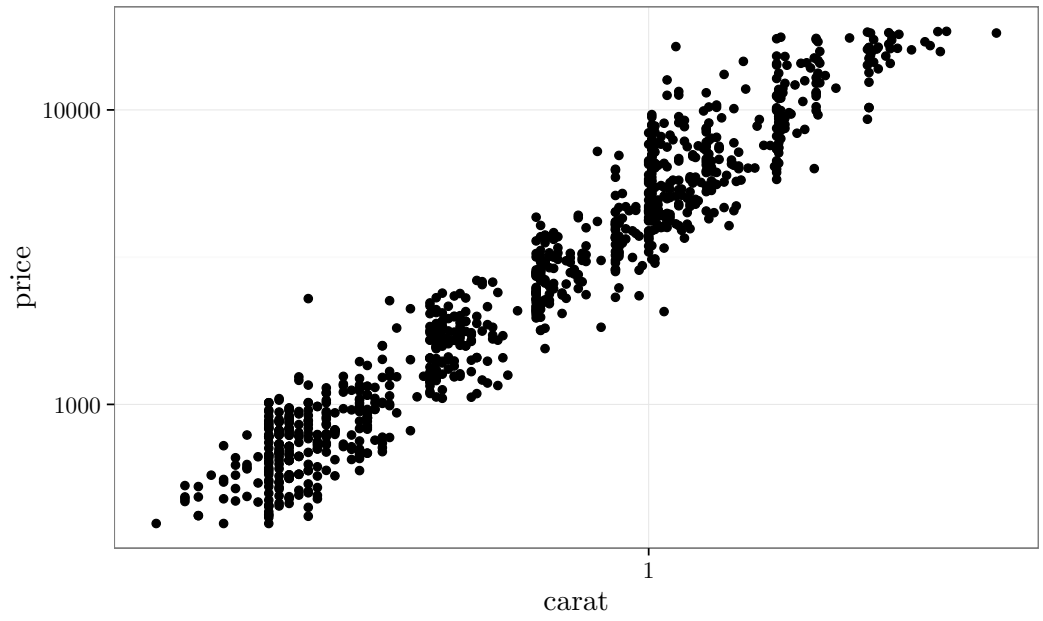



```
qplot(log(carat),
      log(price),
      size = I(1),
      data = diamondsSubset)
```



It is sometimes more readable to display the original un-transformed variables on logarithmic axes.

```
qplot(carat,
      price,
      log = "xy",
      size = I(1),
      data = diamondsSubset)
```

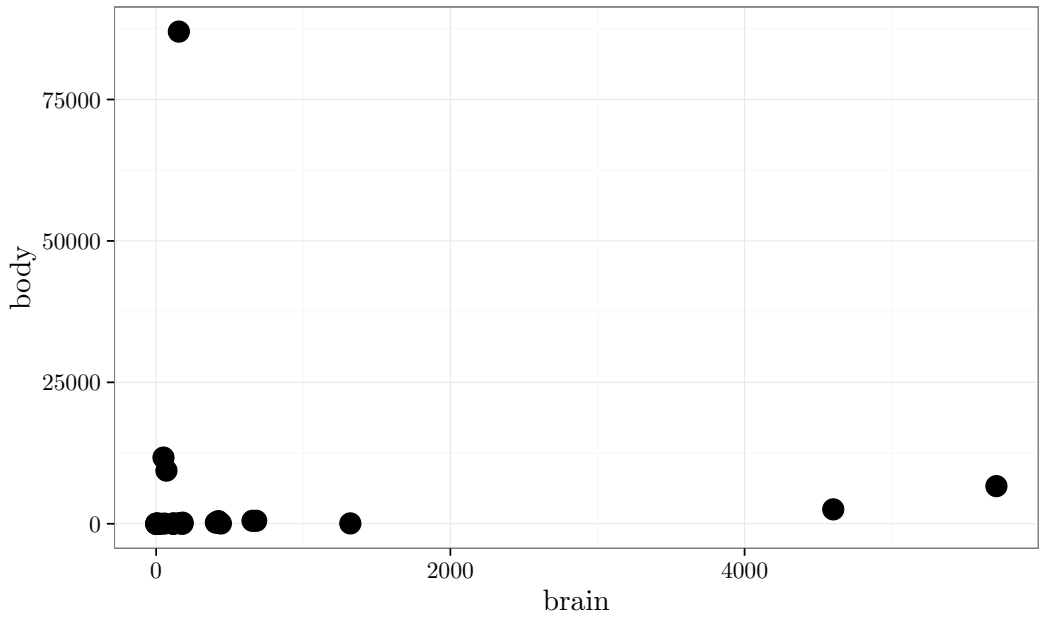


In another example we analyze the `Animals` dataset from the `MASS` package. This dataset contains brain weight (in grams) and body weight (in kilograms)

for 28 different animal species.

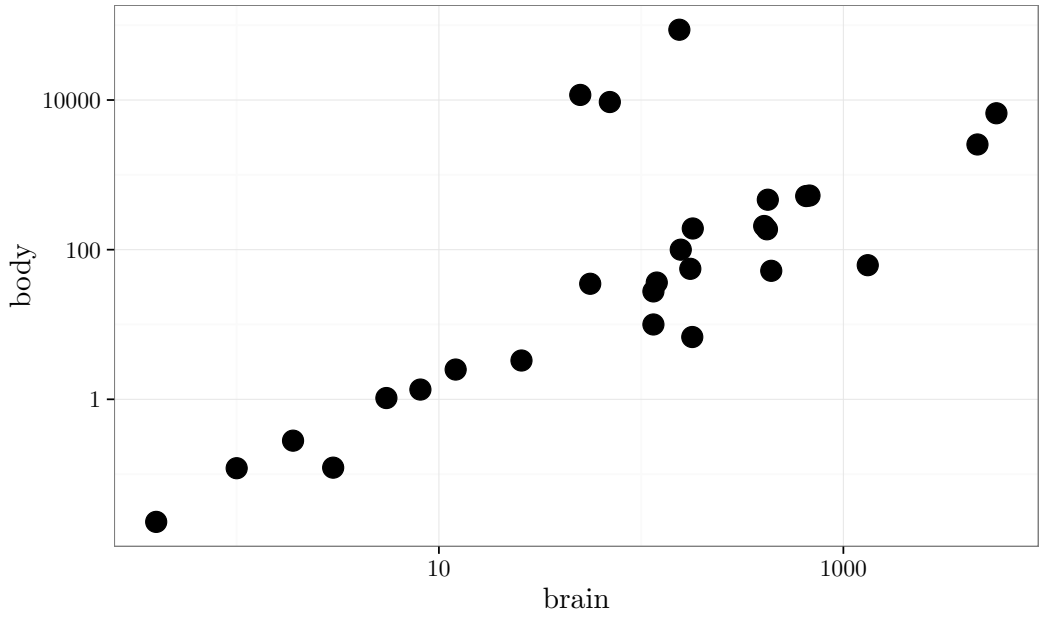
The three largest animals are dinosaurs, whose measurements are obviously the result of scientific modeling rather than precise measurements.

```
library(MASS)
Animals
##                       body  brain
## Mountain beaver      1.350    8.1
## Cow                465.000  423.0
## Grey wolf           36.330  119.5
## Goat                27.660  115.0
## Guinea pig           1.040    5.5
## Dipliodocus      11700.000   50.0
## Asian elephant    2547.000 4603.0
## Donkey             187.100  419.0
## Horse              521.000  655.0
## Potar monkey        10.000  115.0
## Cat                  3.300   25.6
## Giraffe            529.000  680.0
## Gorilla            207.000  406.0
## Human               62.000 1320.0
## African elephant  6654.000 5712.0
## Triceratops       9400.000   70.0
## Rhesus monkey        6.800  179.0
## Kangaroo            35.000   56.0
## Golden hamster       0.120    1.0
## Mouse                0.023    0.4
## Rabbit               2.500   12.1
## Sheep               55.500  175.0
## Jaguar             100.000  157.0
## Chimpanzee          52.160  440.0
## Rat                  0.280    1.9
## Brachiosaurus    87000.000  154.5
## Mole                 0.122    3.0
## Pig                192.000  180.0
qplot(brain, body, data = Animals)
```



The scatter plot above is hard to comprehend. Transforming both quantities by a power transformation reveals interesting linear trend on the log-log scale for most animals, excluding the three dinosaurs.

```
qplot(brain, body, log = "xy", data = Animals)
```



The log-log plot shows a clear cluster of outliers corresponding to the dinosaurs. Apparently, either dinosaurs had a substantially different brain to body weight relationship than contemporary animals, or the estimates are based on flawed models.

11.3.2 Binning

Below, we discuss representing three different types of variables: numeric, ordinal, and categorical, and computing with them.

- A numeric variable represents real valued measurements, and whose values are ordered in a manner consistent with the natural ordering of the real line. We further expect that the dissimilarity between two measurements a, b is described by the Euclidean distance $|b - a|$. For example, height and weight are numeric variables.

- An ordinal variable represents measurements in a certain range R for which we have a well defined order relation. Numeric variables are special cases of ordinal variables. For example, the seasons of the year are ordinal measurements.

- A categorical variable represents measurements that do not satisfy the ordinal or numeric assumption. For example, food items on a restaurant's menu are categorical variables.

The distinction above refers to the essence of the variable rather than how it is stored. For example, we may store categorical measurement corresponding to the fruits {`apple, orange, banana`} using the values $\{0, 1, 2\}$.

The process of binning (also known as discretization) refers to taking a numeric variable $x \in \mathbb{R}$ (typically a real value, though it may be an integer), dividing its range into several bins, and replacing it with a number representing the corresponding bin. Binning is closely related to rounding and in fact all numbers represented digitally are already discretized (see Chapter 1). However, it is often useful to bin values in order to accomplish data reduction, improve scalability for big-data, or capture non-linear effects in linear models. A special case of binning is binarization, which replaces a variable with either 0 or 1 depending on whether the variable is greater or smaller than a certain threshold.

For example, suppose x represent the tenure of an employee (in years) and ranges from 0 to 50. A binning process may divide the range $[0, 50]$ into the following ranges $(0, 10], (10, 20], \ldots, (41, 50]$ and use corresponding replacement values of $5, 15, \ldots, 45$ respectively. The notation $(a, b]$ corresponds to all values larger than a and smaller or equal to b.

The code below shows how to discretize a variable in Python using the `cut` function in the `pandas` module. The output of the `cut` function is an object that has a `categories` field corresponding to the discretized bins, and a `codes` field corresponding to the index of the bin that replaces each value in the data array. Specifically, below we creates an array, define a list of bin ranges and replacement values, call `cut` to replace the original values with the corresponding bins, and then create another array that replaces the values of the original array with the midpoints of the corresponding bins.

```
import numpy as np
import pandas as pd
data = [23, 13, 5, 3, 41, 33]
bin_boundaries = [0, 10, 20, 30, 40 ,50]
bin_values = [5, 15, 25, 35, 45]
cut_data = pd.cut(data, bin_boundaries)
print("labels:\n" + str(cut_data.codes))
binned_data = np.zeros(shape = np.size(data))
for (k,x) in enumerate(cut_data.codes):
    binned_data[k] = bin_values[x]
print("binned_data:\n" + str(binned_data))
## labels:
## [2 1 0 0 4 3]
## binned_data:
## [ 25.  15.   5.   5.  45.  35.]
```

Calling `cut(data,k)` for a positive integer `k`, uses `k` equal-length bins whose coverage equals the range of the data (minimum value, maximum value). In some cases it makes sense to select the bins adaptively, i.e., based on the distribution of the data. The Python function `qcut(data,k)` is similar to `cut(data, k)` except that it uses `k` bins with equal probability mass, where the data distribution is estimated based on the data values. For example `cut(data,2)` performs binning based on two bins whose boundary is the median of the data (see Section 10.10 for a definition of the median). Similarly, `cut(data, 4)` uses bins defined by the sample quartiles. This technique is called quantile binning.

```
import numpy as np
import pandas as pd
original_data = [23, 13, 5, 3, 41, 33]
cut_data = pd.qcut(original_data, 2)
print("C.labels:\n" + str(cut_data.codes))
## C.labels:
## [1 0 0 0 1 1]
```

Discretization in R is similar to Python using the R `cut` function.

11.3.3 Indicator Variables

Indicator vectors refers to a technique that replaces a variable x (numeric, ordinal, or categorical) taking k values with a binary k-dimensional vector v, such that `v[i]` (or v_i in mathematical notation) is one if and only if x takes on the i-value in its range. Thus, the variable is replaced by a vector that is all zeros, except for one component that equals one corresponding to the variable value. Often, indicator variables are used in conjunction with binning as follows: first bin the variable into k bins and then create a k dimensional indicator variable. The resulting vector may have high dimension, but it may be easily handled using computational routines that take advantage of its extreme sparsity.

Indicator vectors are useful in data modeling in two cases.

1. Models for numeric or binary data cannot directly model ordinal or categorical data. For example, replacing {`apple, orange, banana`} with $\{0, 1, 2\}$ and using these values in a model for numeric values would incorrectly assume that banana is greater than orange and that the dissimilarity between orange and apple is identical to the dissimilarity between apple and banana. Using indicator variables can mitigate this problem.

2. In some cases a linear model is used to model numeric variables that exhibit non-linear relationship. As the linearity assumption of the model is violated, the resulting model may be inaccurate. An alternative is to first transform the numeric values using several non-linear transformations (for example multiple power transformations), then bin the transformed data, and finally create indicator vectors to represent the binned values. Training a linear models on the resulting vectors may capture complex non-linear relationships.

3. It is often much easier to compute with indicator functions since they are binary, and thus replacing numeric variables with indicator vectors may improve scalability.

The Python code below defines a numeric array, bins it, and then create indicator vectors to replace the binned values.

```
import numpy as np
import pandas as pd
data = [23, 13, 5, 3, 41, 33]
indicator_values = pd.get_dummies(pd.qcut(data, 2))
print("indicator_values:\n" + str(indicator_values))
## indicator_values:
##    [3, 18]  (18, 41]
## 0      0.0       1.0
## 1      1.0       0.0
## 2      1.0       0.0
## 3      1.0       0.0
## 4      0.0       1.0
## 5      0.0       1.0
```

In many cases, a machine learning routine accepts a single feature vector that needs to be composed from multiple variables. A common strategy is to transform all numeric variable by binning them and then creating indicator vectors and concatenating all of them into a single long binary vector. Additional non-numeric variables can be concatenated as well. Some variations on this strategy is to concatenate the numeric vectors in their original form and possibly concatenate them with transformed versions of the numeric variables (for example power transformations).

11.4 Data Manipulation

We discuss below several common operations that manipulate the structure of dataframes.

11.4.1 Random Sampling, Partitioning, and Shuffling

A common operation in data analysis is to select a random subset of the rows of a dataframe, with or without replacement. For example, a subset of size 2 without replacement corresponds to selecting one row, and then selecting another row from the remaining rows. Sampling with replacement may select the same row multiple times.

We demonstrate these tasks using R code. Python's pandas module features similar functionality. The R `sample` function accepts a vector of values from which to sample (typically a vector of row indices), the number of samples, whether the sampling is done with or without replacement, and the probability of sampling different values. By default, the probability of sampling each value is identical: $1/k$ where k is the number of values from which we sample. For example, the R code below samples 3 times from the values 1,2,3,4, without replacements.

```
sampled_row_indices = sample(1:4, 3, replace=FALSE)
print(sampled_row_indices)
## [1] 1 4 2
```

After obtaining the indices that we wish to sample. we form a new array or dataframe containing the sampled rows of the original dataframe.

```
D = array(data = seq(1, 20, length.out = 20), dim = c(4, 5))
D_sampled = D[sampled_row_indices,]
print(D_sampled)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    4    8   12   16   20
## [3,]    2    6   10   14   18
```

In some cases, we need to partition the dataset's rows into two or more collection of rows. To do so, we proceed with generating a random permutation of k objects (using `sample(k,k)`), where k is the number of rows in the data, and then divide the permutation vector into two or more parts based on the prescribed sizes. We then create new dataframes whose rows correspond to the divided permutation vector. For example, the R code below partitions an array into two new arrays of sizes 75% and 25%.

```
D = array(data = seq(1, 20, length.out = 20), dim = c(4, 5))
print(D)
##      [,1] [,2] [,3] [,4] [,5]
```

```
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
rand_perm = sample(4,4)
first_set_of_indices = rand_perm[1:floor(4*0.75)]
second_set_of_indices = rand_perm[(floor(4*0.75)+1):4]
D1 = D[first_set_of_indices,]
D2 = D[second_set_of_indices,]
print(D1)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    4    8   12   16   20
## [2,]    1    5    9   13   17
## [3,]    2    6   10   14   18
print(D2)
## [1]  3  7 11 15 19
```

A related task is data shuffling, which randomly shuffles the dataframe rows.

```
D = array(data = seq(1, 20, length.out = 20), dim = c(4, 5))
print(D)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
D_shuffled = D[sample(4, 4),]
print(D_shuffled)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    4    8   12   16   20
## [3,]    3    7   11   15   19
## [4,]    2    6   10   14   18
```

11.4.2 Concatenations and Joins

Often, data needs to be aggregated from multiple sources into a single object that will be used for visualization and modeling. We demonstrate below how to do it with Python. R features similar capability.

In some cases the two data sources contain different records (dataframe rows) annotated with the same attribute names (column names). In this case aggregating the two sources is simply concatenating the rows of the two data frames. The Python code below creates two data frames with identical column names and then concatenates them.

```
import numpy as np
import pandas as pd
```

```
data1 = {"ID" : ["2134", "4524"],
         "name" : ["John Smith", "Jane Doe"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["9423", "3483"],
         "name" : ["Bob Jones", "Mary Smith"]}
D2 = pd.DataFrame(data2)
print("D1:\n" + str(D1))
print("D2:\n" + str(D2))
D3 = pd.concat([D1, D2])
print("concatenation of D1, D2:\n" + str(D3))
## D1:
##      ID        name
## 0  2134  John Smith
## 1  4524    Jane Doe
## D2:
##      ID        name
## 0  9423   Bob Jones
## 1  3483  Mary Smith
## concatenation of D1, D2:
##      ID        name
## 0  2134  John Smith
## 1  4524    Jane Doe
## 0  9423   Bob Jones
## 1  3483  Mary Smith
```

When concatenating dataframes with non identical columns, new columns are added to the concatenated dataframe with missing values filled in as needed.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524"],
         "name" : ["John Smith", "Jane Doe"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["9423", "3483"],
         "nick name" : ["Bobby", "Abby"]}
D2 = pd.DataFrame(data2)
print("D1:\n" + str(D1))
print("D2:\n" + str(D2))
D3 = pd.concat([D1, D2])
print("concatenation of D1, D2:\n" + str(D3))
## D1:
##      ID        name
## 0  2134  John Smith
## 1  4524    Jane Doe
## D2:
##      ID nick name
## 0  9423     Bobby
## 1  3483      Abby
## concatenation of D1, D2:
```

```
##      ID         name nick name
## 0  2134  John Smith       NaN
## 1  4524    Jane Doe       NaN
## 0  9423         NaN     Bobby
## 1  3483         NaN      Abby
```

Thus far, we assumed that the records in both data sources correspond to distinct entities. In some cases the same entity is described in two data sources that potentially list different attributes. However, in order to determine how to match records in one data source to records in another source we assume that each data source has one or more columns acting as identifiers. The identifiers can be used to match a row in one dataframe with a row in a different dataframe. The operation of merging such data is called a join and the identifier column is called the key.

We create below two data frames where the identifier attribute is named `ID` in both data sources. An inner join operation retains records that appear in both data sources and all attributes that appear in either the first or the second data source.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524"],
         "name" : ["John Smith", "Jane Doe"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["6325", "2134"],
         "age" : [25, 35],
       "tenure" : [3, 8]}
D2 = pd.DataFrame(data2)
print("D1:\n" + str(D1))
print("D2:\n" + str(D2))
D3 = pd.merge(D1, D2, on = 'ID', how = 'inner')
print("inner join of D1, D2:\n" + str(D3))
## D1:
##      ID        name
## 0  2134  John Smith
## 1  4524    Jane Doe
## D2:
##      ID  age  tenure
## 0  6325   25       3
## 1  2134   35       8
## inner join of D1, D2:
##      ID        name  age  tenure
## 0  2134  John Smith   35       8
```

An outer join operation is similar to inner join, except that all records and all attributes are retained, with missing values (denoted by `NA`) filled in as needed. A left join is similar, except that all common records and attributes are retained plus the records and attributes in the left data source. A right join is similar,

except that all common records and attributes are retained plus the records and attributes in the right data source.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524"],
         "name" : ["John Smith", "Jane Doe"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["6325", "2134"],
         "age" : [25, 35],
         "tenure" : [3, 8]}
D2 = pd.DataFrame(data2)
D3 = pd.merge(D1, D2, on = 'ID', how = 'outer')
print("outer join of D1, D2:\n" + str(D3))
D4 = pd.merge(D1, D2, on = 'ID', how = 'left')
print("left join of D1, D2:\n" + str(D4))
## outer join of D1, D2:
##      ID        name   age  tenure
## 0  2134  John Smith  35.0     8.0
## 1  4524    Jane Doe   NaN     NaN
## 2  6325         NaN  25.0     3.0
## left join of D1, D2:
##      ID        name   age  tenure
## 0  2134  John Smith  35.0     8.0
## 1  4524    Jane Doe   NaN     NaN
```

Above, we assumed that the key attribute acts as a unique identifier in both data sources (any specific key appears at most one time). If we have multiple records with the same key in one or both of the data sources, the join operation forms multiple combinations.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524", "2134"],
         "name" : ["John Smith", "Jane Doe", "JOHN SMITH"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["6325", "2134"],
         "age" : [25, 35],
         "tenure" : [3, 8]}
D2 = pd.DataFrame(data2)
D3 = pd.merge(D1, D2, on = 'ID', how = 'outer')
print("outer join of D1, D2:\n" + str(D3))
## outer join of D1, D2:
##      ID        name   age  tenure
## 0  2134  John Smith  35.0     8.0
## 1  2134  JOHN SMITH  35.0     8.0
## 2  4524    Jane Doe   NaN     NaN
## 3  6325         NaN  25.0     3.0
```

We also assumed above that the non-key attribute names are different in both data sources. If they are not, a suffix is introduced in the merged dataframe.

```
import numpy as np
import pandas as pd
data1 = {"ID" : ["2134", "4524"],
         "f1" : ["John Smith", "Jane Doe"]}
D1 = pd.DataFrame(data1)
data2 = {"ID" : ["6325", "2134"],
         "f1" : [25, 35],
         "f2" : [3, 8]}
D2 = pd.DataFrame(data2)
D3 = pd.merge(D1, D2, on = 'ID', how = 'outer')
print("outer join of D1, D2:\n" + str(D3))
## outer join of D1, D2:
##      ID        f1_x  f1_y   f2
## 0  2134  John Smith  35.0  8.0
## 1  4524    Jane Doe   NaN  NaN
## 2  6325         NaN  25.0  3.0
```

11.4.3 Tall Data and Wide Data

Data in tall format is an array or dataframe containing multiple columns where one or more columns act as a unique identifier and an additional column represents value. For example, consider the tall data below representing item sales in a grocery store.

```
2015/01/01  apples  200
2015/01/01  oranges 150
2015/01/02  apples  220
2015/01/02  oranges 130
```

Such data may represent the entire sales records of the store. This format is convenient for adding new records incrementally representing additional sales as they occur, and for removing old records (possibly due to returns in the above case representing a store). The designation "tall data" reflects the fact that in most cases tall data has many rows but only a few columns.

A disadvantage of tall data format is that it not easy for conducting analysis or summarizing. For example, we may want to compute the total sales for each day or the average daily sales of a specific item such as apples. Computing these quantities from tall data requires writing a program that will collect all relevant rows, aggregate it appropriately, and produce the desired quantity. This is not only time consuming from a programming perspective, but it may also be time consuming in terms of computing time (if there are many rows).

Wide data represents the same information as tall data, but may represent in multiple columns the information that tall data holds in multiple rows. For example, the wide data version of the grocery data above is listed below.

```
Date       apples  oranges
-------------------------
2015/01/01  200     150
2015/01/02  220     130
```

Wide data is usually simpler to analyze. For example, computing the total sales per day requires summing over each row and computing the average daily per-item sales requires computing averages over columns.

Note that when converting the above data from tall to wide, the first column and the second column represent unique keys and the last column represent the corresponding value. In other words, each (date, item) combination can appear only once, while no such restriction applies to the third column. In converting the tall data to wide, the first column in the tall data became the first column in the wide data, the second column in the tall data became the column names in the wide data, and the third column formed the remaining table entries.

11.4.4 Reshaping Data

In this section we describe how to convert data from tall to wide format and vice verse using the reshape2 package in R [41]. Similar functionality is available in Python's pandas module.

The `melt` function accepts a dataframe in a wide format, and the indices of the columns that act as unique identifiers (remaining columns act as measurements or values). It returns a tall version of the dataframe. The R code below demonstrates this with two different selection of identifier columns.

```
library(reshape2)
# toy (wide) dataframe in the reshape2 package
smiths
##      subject time age weight height
## 1 John Smith    1  33     90   1.87
## 2 Mary Smith    1  NA     NA   1.54
# columns 2, 3, 4, 5 are measurements, 1 is key
melt(smiths, id = 1)
##      subject variable value
## 1 John Smith     time  1.00
## 2 Mary Smith     time  1.00
## 3 John Smith      age 33.00
## 4 Mary Smith      age    NA
## 5 John Smith   weight 90.00
## 6 Mary Smith   weight    NA
## 7 John Smith   height  1.87
## 8 Mary Smith   height  1.54
# columns 3, 4, 5 are measurements, 1,2 are key
melt(smiths, id = c(1, 2))
##      subject time variable value
## 1 John Smith    1      age 33.00
```

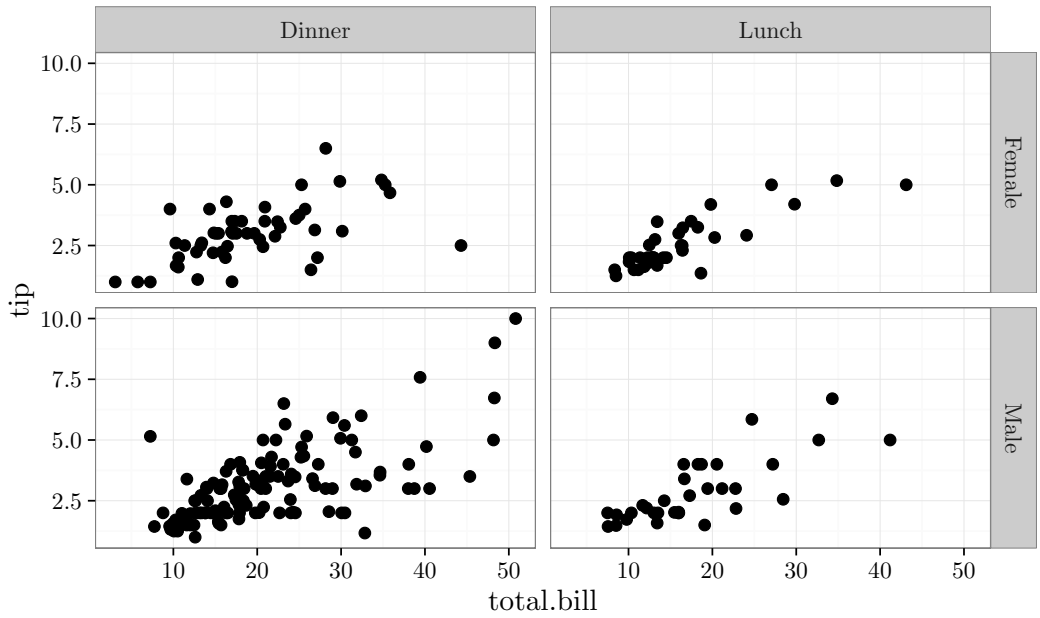
```
## 2 Mary Smith    1      age    NA
## 3 John Smith    1   weight 90.00
## 4 Mary Smith    1   weight    NA
## 5 John Smith    1   height  1.87
## 6 Mary Smith    1   height  1.54
```

Note that the tall data produced by `melt` features appropriate column names.

The functions `acast` or `dcast` (the first returns an array and the second a dataframe) represent the inverse of the melt operation. Their argument is a dataframe in wide form and the second is a formula $a \sim b \sim \cdots \sim$ where each of $a, b, \ldots$ represents a list of variables whose values will be displayed along the dimensions of the returned array or dataframe (a for rows, b for columns, etc.). The cast array or dataframe may have at most one value in each cell. If there are more than a single value setting, a third argument `fun.aggregate` executes the corresponding function in order to aggregate the multiple values into a single value.

The example below uses the `tips` dataset from the `reshape2` package. It contains 244 restaurant tips. Dataframe columns include tip, bill, gender of payer, smoker/non-smoker, day of the week, time of day, and size of party. Type `help(tips)` for more information.

```
tips$total.bill = tips$total_bill
qplot(total.bill,
      tip,
      facets = sex~time,
      size = I(1.5),
      data = tips)
```



We can see from the figure that (1) tip sizes increase with the bill, (2) the variability in tip sizes increase with the bill, (3) bills and tips in dinner are larger than in lunch, (4) there are more dinner tip observations than lunch tip observations, and (5) there are more male payers than female payers.

In the example below we are interested in analyzing tips and bills and the dependence of these variables on the remaining variables. We thus denote the tip and total bill as the measurement variables and the remaining variables as identifiers.

```
library(reshape2)
head(tips)  # first six rows
##   total_bill  tip    sex smoker day   time
## 1      16.99 1.01 Female     No Sun Dinner
## 2      10.34 1.66   Male     No Sun Dinner
## 3      21.01 3.50   Male     No Sun Dinner
## 4      23.68 3.31   Male     No Sun Dinner
## 5      24.59 3.61 Female     No Sun Dinner
## 6      25.29 4.71   Male     No Sun Dinner
##   size total.bill
## 1    2      16.99
## 2    3      10.34
## 3    3      21.01
## 4    2      23.68
## 5    4      24.59
## 6    4      25.29
tipsm = melt(tips,
             id = c("sex","smoker","day","time","size"))
head(tipsm)  # first six rows
##      sex smoker day   time size   variable
## 1 Female     No Sun Dinner    2 total_bill
## 2   Male     No Sun Dinner    3 total_bill
## 3   Male     No Sun Dinner    3 total_bill
## 4   Male     No Sun Dinner    2 total_bill
## 5 Female     No Sun Dinner    4 total_bill
## 6   Male     No Sun Dinner    4 total_bill
##   value
## 1 16.99
## 2 10.34
## 3 21.01
## 4 23.68
## 5 24.59
## 6 25.29
tail(tipsm)  # last six rows
##        sex smoker  day   time size
## 727 Female     No  Sat Dinner    3
## 728   Male     No  Sat Dinner    3
## 729 Female    Yes  Sat Dinner    2
## 730   Male    Yes  Sat Dinner    2
## 731   Male     No  Sat Dinner    2
```

```
## 732 Female     No Thur Dinner    2
##       variable value
## 727 total.bill 35.83
## 728 total.bill 29.03
## 729 total.bill 27.18
## 730 total.bill 22.67
## 731 total.bill 17.82
## 732 total.bill 18.78
# Mean of measurement variables broken by sex.
# Note the role of mean as the aggregating function.
dcast(tipsm,
      sex~variable,
      fun.aggregate = mean)
##      sex total_bill      tip total.bill
## 1 Female   18.05690 2.833448   18.05690
## 2   Male   20.74408 3.089618   20.74408
# Number of occurrences for measurement variables broken by sex.
# Note the role of length as the aggregating function.
dcast(tipsm,
      sex~variable,
      fun.aggregate = length)
##      sex total_bill tip total.bill
## 1 Female         87  87         87
## 2   Male        157 157        157
# Average total bill and tip for different times
dcast(tipsm,
      time~variable,
      fun.aggregate = mean)
##     time total_bill      tip total.bill
## 1 Dinner   20.79716 3.102670   20.79716
## 2  Lunch   17.16868 2.728088   17.16868
# Similar to above with breakdown for sex and time:
dcast(tipsm,
      sex+time~variable,
      fun.aggregate = length)
##      sex   time total_bill tip total.bill
## 1 Female Dinner         52  52         52
## 2 Female  Lunch         35  35         35
## 3   Male Dinner        124 124        124
## 4   Male  Lunch         33  33         33
# Similar to above, but with mean and added margins
dcast(tipsm,
      sex+time~variable,
      fun.aggregate = mean,
      margins = TRUE)
##      sex   time total_bill      tip
## 1 Female Dinner   19.21308 3.002115
## 2 Female  Lunch   16.33914 2.582857
## 3 Female  (all)   18.05690 2.833448
```

```
## 4   Male Dinner   21.46145 3.144839
## 5   Male  Lunch   18.04848 2.882121
## 6   Male  (all)   20.74408 3.089618
## 7  (all)  (all)   19.78594 2.998279
##   total.bill    (all)
## 1   19.21308 13.80942
## 2   16.33914 11.75371
## 3   18.05690 12.98241
## 4   21.46145 15.35591
## 5   18.04848 12.99303
## 6   20.74408 14.85926
## 7   19.78594 14.19005
```

The melt and cast analysis above suggests the following conclusions with respect to the `tips` dataframe.

1. On average, males pay higher total bill and tip than females.

2. Males pay more frequently than females.

3. Dinner bills and tips are generally higher than lunch bills and tips.

4. Males pay disproportionately more times for dinner than they do for lunch (this holds much less for females).

5. Even accounting for (4) by conditioning on paying for lunch or dinner, males still pay higher total bills and tips than females.

A graphical investigation using faceted scatter plots and histograms may reveal similar conclusions. Nevertheless, the above analysis using cast and melt has an advantage over graphical analysis in that a proficient user can very quickly observe properties of the data that are hard to graph (number of measurements for different combination of identifier variables, means of different groups, etc).

The online package documentation or [41] provides additional information on the `reshape2` package.

11.4.5 The Split-Apply-Combine Framework

Many data analysis operations on dataframes can be decomposed to three stages:

1. splitting the dataframe along some dimensions to form smaller arrays or dataframes,

2. applying some operation to each of the smaller arrays or dataframes, and

3. combining the results of the application stage into a single meaningful array or dataframe.

Repeatedly programming all three stages whenever we need to compute a data summary may be tedious and can lead to errors. The plyr package in R [43] automates this process, letting the analyst concentrate on the data analysis task rather than tedious programming.

The plyr package implements the following functions that differ in the type of input arguments they receive and the type of output they provide.

output / input	array	dataframe	list	discarded
array	`aaply`	`adply`	`alply`	`a_ply`
dataframe	`daply`	`ddply`	`dlply`	`d_ply`
list	`laply`	`ldply`	`llply`	`l_ply`

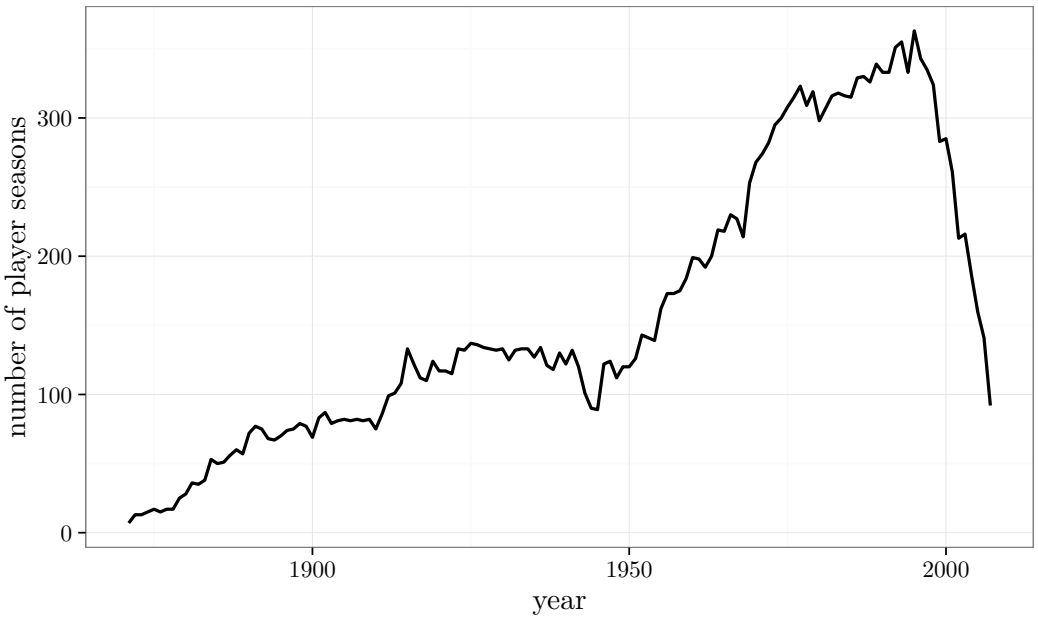
The first argument in each of these functions is the data stored as an array, dataframe, or list depending on the input type in the table above. The second argument of the `a*ply` functions[1], called `.margins`, determines the dimensions that are used to split the data. A value of 1 implies splitting by rows. A value of 2 implies splitting into columns, and so forth. A combination value may also be used, for example `c(1,2)` splits the data into a combination of rows and columns. The second argument of the `d*ply` functions, called `.variables`, determines the dataframe columns (multiple columns are allowed) that are used to split the data. Since there is only one way to split a list there is no corresponding argument for the `l*ply` functions. For all functions the argument `.fun` determines which function to execute in the apply stage.

We illustrate these functions using the `baseball` dataset from the `plyr` package. This dataset contains variables such as year, team, number of runs, number of strikeouts for 1228 baseball players. Each row records the performance of a baseball player during one baseball season. In particular, the performance of specific players is spread across multiple rows, one corresponding to each year played. The following example is inspired by the R help listing of the `ddply` function.

```
library(plyr)
head(baseball)
##             id year stint team lg  g  ab  r
## 4   ansonca01 1871     1  RC1    25 120 29
## 44  forceda01 1871     1  WS3    32 162 45
## 68  mathebo01 1871     1  FW1    19  89 15
## 99  startjo01 1871     1  NY2    33 161 35
## 102 suttoez01 1871     1  CL1    29 128 35
## 106 whitede01 1871     1  CL1    29 146 40
##      h X2b X3b hr rbi sb cs bb so ibb hbp sh
## 4   39  11   3  0  16  6  2  2  1  NA  NA NA
## 44  45   9   4  0  29  8  0  4  0  NA  NA NA
```

[1] We use the asterisk in `a*ply` and elsewhere to indicate a collection of functions obtained by substituting the asterisk with other characters.

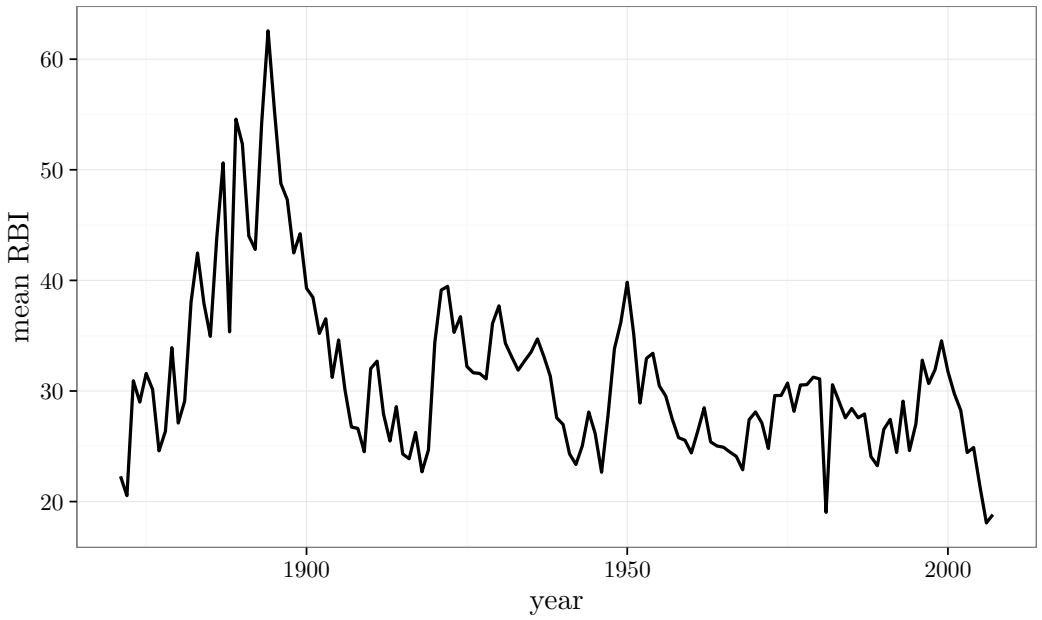
```
## 68  24   3   1  0  10  2  1  2  0  NA  NA NA
## 99  58   5   1  1  34  4  2  3  0  NA  NA NA
## 102 45   3   7  3  23  3  1  1  0  NA  NA NA
## 106 47   6   5  1  21  2  2  4  1  NA  NA NA
##      sf gidp
## 4    NA   NA
## 44   NA   NA
## 68   NA   NA
## 99   NA   NA
## 102 NA   NA
## 106 NA   NA
# count number of players recorded for each year
bbPerYear = ddply(baseball, "year", "nrow")
head(bbPerYear)
##   year nrow
## 1 1871    7
## 2 1872   13
## 3 1873   13
## 4 1874   15
## 5 1875   17
## 6 1876   15
qplot(x = year,
      y = nrow,
      data = bbPerYear,
      geom = "line",
      ylab="number of player seasons")
```



The number of player seasons recorded in the dataset for each year increases dramatically from 1880 to around 1990, but decreases in later years close to the turn of the century. The simple one line command above would be otherwise

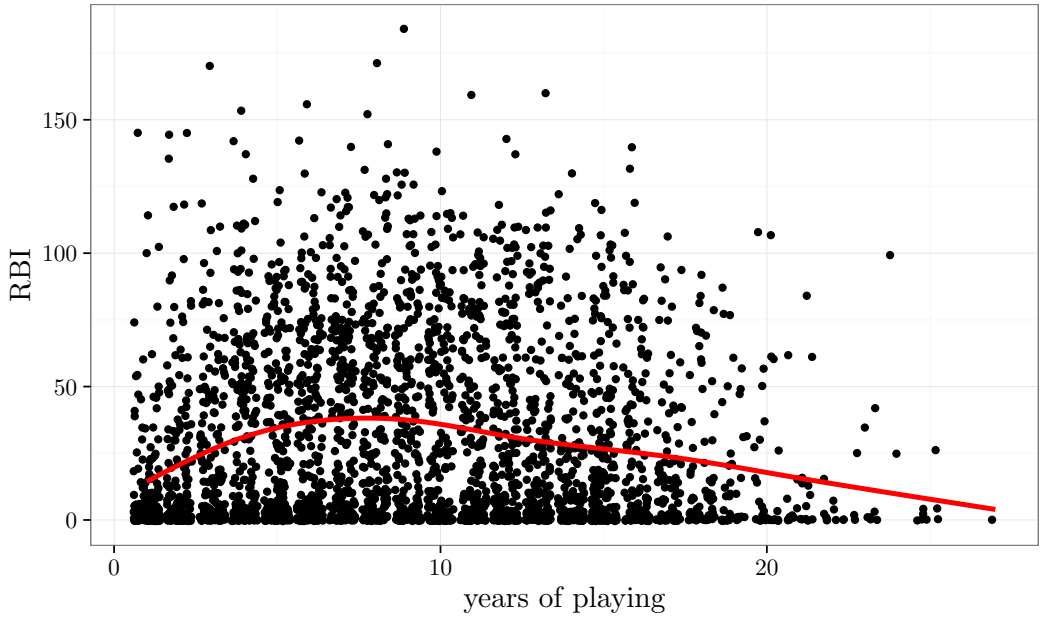
more complicated.

```
# compute mean rbi (batting attempt resulting in runs)
# for all years. Summarize is the apply function, which
# takes as argument a function that computes the rbi mean
bbMod=ddply(baseball,
            "year",
            summarise,
            mean.rbi = mean(rbi, na.rm = TRUE))
head(bbMod)
##   year mean.rbi
## 1 1871 22.28571
## 2 1872 20.53846
## 3 1873 30.92308
## 4 1874 29.00000
## 5 1875 31.58824
## 6 1876 30.13333
qplot(x = year,
      y = mean.rbi,
      data = bbMod,
      geom = "line",
      ylab = "mean RBI")
```



The graph above shows that the mean of the RBI, which is a batting performance measure, was substantially higher in the late nineteenth century than in other times. Below, we add another variable to the dataframe containing the number of years a player has played baseball up to that point. It is computed as the current year minus the year the player started playing plus one.

```
# add a column career.year which measures the number of years passed
# since each player started batting
bbMod2 = ddply(baseball,
               "id",
               transform,
               career.year = year - min(year) + 1)
# sample a random subset 3000 rows to avoid over-plotting
bbSubset = bbMod2[sample(dim(bbMod2)[1], 3000),]
qplot(career.year,
      rbi, data = bbSubset,
      size = I(0.8),
      geom = "jitter",
      ylab = "RBI",
      xlab = "years of playing") +
  geom_smooth(color = "red", se = F, size = 1.5)
```

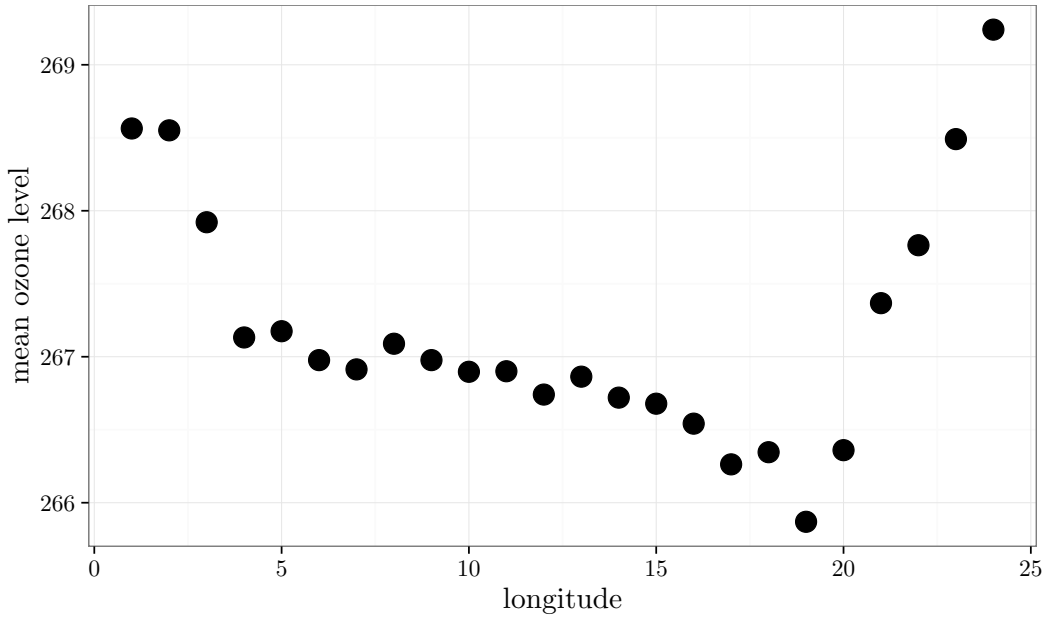


The RBI tends to improve with experience up to 7 or 8 years and then starts to decline (on average). RBI for players with more than 20 years experience tends to be very low, although a few exceptions exist (outliers at the top-right corner of the figure above).

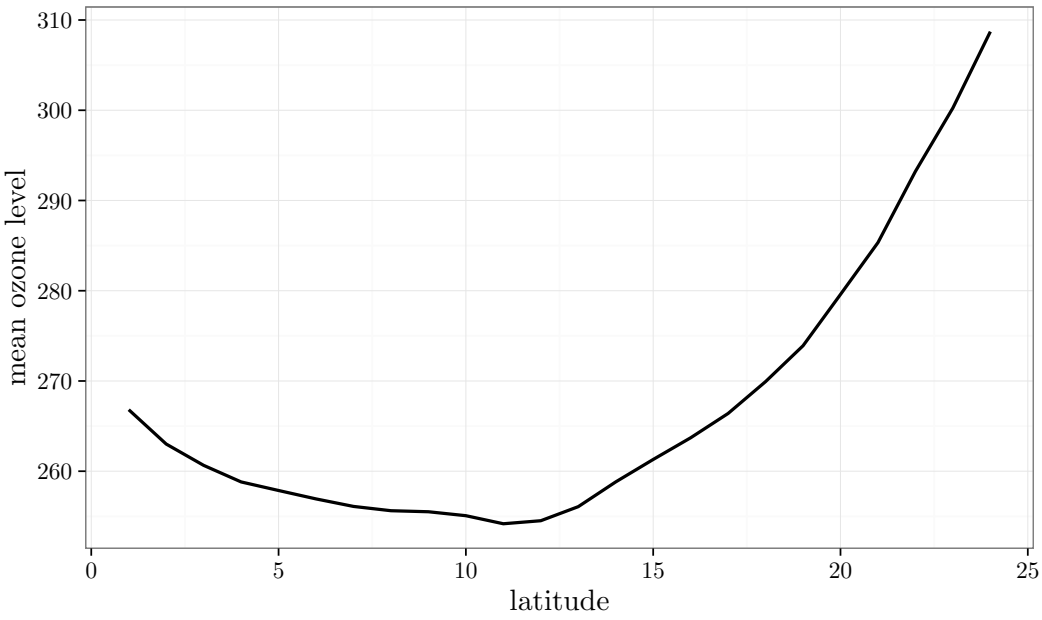
The example below explores the function `aaply` using the `ozone` dataset from the `plyr` package. The ozone dataset contains a 3-dimensional array of ozone measurements varying by latitude, longitude, and time.

```
library(plyr)
dim(ozone)
## [1] 24 24 72
latitude.mean = aaply(ozone, 1, mean)
longitude.mean = aaply(ozone, 2, mean)
```

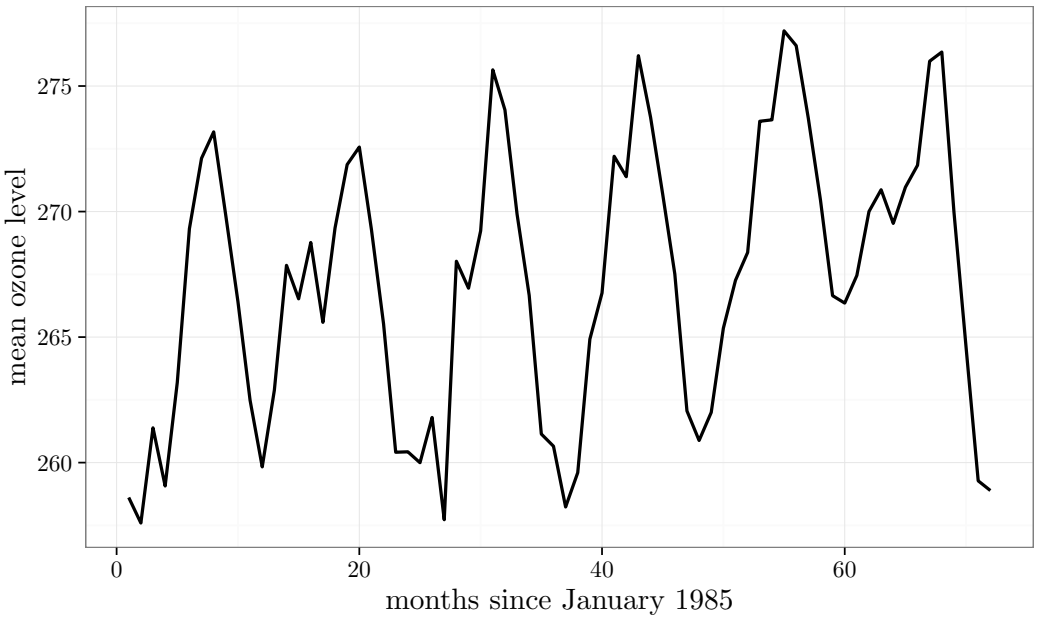
```
time.mean = aaply(ozone, 3, mean)
longitude = seq(along = longitude.mean)
qplot(x = longitude,
      y = longitude.mean,
      ylab = "mean ozone level")
```




```
latitude = seq(along = latitude.mean)
qplot(x = latitude,
      y = latitude.mean,
      ylab = "mean ozone level",
      geom = "line")
```




```
months = seq(along = time.mean)
qplot(x = months,
      y = time.mean,
      geom = "line",
      ylab = "mean ozone level",
      xlab = "months since January 1985")
```



From the three figures above, we conclude that ozone has a clear minimum mean ozone level at longitude 19 and latitude 12, and that the ozone level has an interesting temporal periodicity. Not unexpectedly, the periodicity coincides

with the annual season cycle (each period is 12 months).

The functions in the `plyr` package are very general. See [43] or the online package documentation for more detail.

11.5 Notes

Our discussion in this chapter of handling missing values, outliers, and skewed data is superficial. An in-depth treatment requires substantial technical prerequisites including probability and statistics. There are several sources containing such an in-depth description. For the topic of missing data refer to [19]. For the topic of robustness refer to [13] or the more recent [23]. Power transformations appear in most regression textbooks, for example [17]. More information on manipulating data using R appears in [35]. Specific details on the `reshape2` and `plyr` packages appear in [41] and [43].

11.6 Exercises

1. Follow the example of analyzing the ozone dataset above, but investigate the ozone variance instead of the ozone mean. Are there any meaningful conclusions you can make regarding the change of variance in space and time?

2. The chapter shows how to use power transformations to transform data that is right-skewed or left-skewed into a symmetric shape. In some cases data is skewed towards both sides (there is a heavy tail to the left and a heavy tail to the right). Suggest a generalization of the power transformation that removes such skewness.

3. The chapter presents three alternatives for handling missing data. What could go wrong when each one of these alternatives is used to handle data that is not MCAR?

4. The packages `reshape2` and `plyr` share some functionality. Show two distinct examples and provide code that accomplishes the same results with both packages. For such tasks, which of the two packages would you prefer? What are the pros and cons of the two packages?