Chapter 7

Transforming Data

An interesting issue with the delivery of a data mining project is that in reality we spend more of our time working on and with the data than we do building actual models, as we suggested in Chapter 1. In building models, we will often be looking to improve their performance. The answer is often to improve our data. This might entail sourcing some additional data, cleaning up the data, dealing with missing values in the data, transforming the data, and analysing the data to raise its efficiency through a better choice of variables.

In general, we need to transform our data from the raw data originally supplied for a data mining project to the polished and focussed data from which we build our best models. This is often the make-or-break phase of a data mining project.

This chapter introduces these data issues. We then review the various options for dealing with some of these issues, illustrating how to do so in Rattle and R.

7.1 Data Issues

A review of the winning entries in the annual data mining competitions reinforces the notion that building models from the right data is crucial to the success of a data mining project. The ACM KDD Cup, an annual Data Mining and Knowledge Discovery competition, is often won by a team that has placed a lot of effort in preprocessing the data supplied.

The 2009 ACM KDD Cup competition is a prime example. The French telecommunications company Orange supplied data related to customer relationship management. It consisted of 50,000 observations with much missing data. Each observation recorded values for 15,000 (anonymous) variables. There were three target variables to be modelled. One of the common characteristics for many entries was the preprocessing performed on the data. This included dealing with missing values, recoding data in various ways, and selecting variables. Some of the resulting models, for example, used only one or two hundred of the original 15,000 variables.

We review in this section some of the issues that relate to the quality of the data that we might have available for data mining. We then consider how we deal with these issues in the following sections.

An important point to understand is that often in data mining we are making use of, and indeed making do with, the data that is available. Such data might be regularly collected for other purposes. Some variables might be critical to the operation of the business and so special attention is paid to ensuring its accuracy. However, other data might only be informational and so less attention is paid to its quality.

We need to understand many different aspects about how and why the data was collected in order to understand any data issues. It is crucial to spend time understanding such data issues. We should do this before we start building models and then again when we are trying to understand why particular models have emerged. We need to explore the data issues that may have led to specific patterns or anomalies in our models. We may then need to rectify those issues and rebuild our models.

Data Cleaning

When collecting data, it is not possible to ensure it is perfectly collected, except in trivial cases. There will always be errors in the collection, despite how carefully it might have been collected. It cannot be stressed enough that we always need to be questioning the quality of the data we have. Particularly in large data warehouse environments where a lot of effort has already been expended in addressing data quality issues, there will still remain dirty data. It is important to always question the data quality and to be alert to the issue.

There are many reasons for the data to be dirty. Simple data entry errors occur frequently. Decimal points can be incorrectly placed, turning \$150.00 into \$15000. There can be inherent error in any counting or measuring device. There can also be external factors that cause errors to change over time, and so on.

One of the most important ongoing tasks we have in data mining, then, is cleaning our data. We usually start cleaning the data before we build our models. Exploring the data and building descriptive and predictive models will lead us to question the quality of the data at different times, particularly when we identify odd patterns.

A number of simple steps are available in reviewing the quality of our data. In exploring data, we will often explore variables through frequency counts and histograms. Any anomalous patterns there should be explored and explained. For categoric variables, for example, we would be on the lookout for categories with very low frequency counts. These might be mistyped or differently typed (upper/lowercase) categories.

A major task in data cleaning is often focussed around cleaning up names and addresses. This becomes particularly significant when bringing data together from multiple sources. In combining financial and business data from numerous government agencies and public sources, for example, it is not uncommon to see an individual have his or her name recorded in multiple ways. Up to 20 or 30 variations can be possible. Street addresses present the same issues. A significant amount of effort is often expended in dealing with cleaning up such data in many organisations, and a number of tools have been developed to assist in the task.

Missing Data

Missing data is a common feature of any dataset. Sometimes there is no information available to populate some value. Sometimes the data has simply been lost, or the data is purposefully missing because it does not apply to a particular observation. For whatever reason the data is missing, we need to understand and possibly deal with it.

Missing values can be difficult to deal with. Often we will see missing values replaced with sentinels to mark that they are missing. Such sentinels can include things like 9999, or 1 Jan 1900, or even special characters that can interfere with automated processing like “*”, “?”, “#”, or “$”. We consider dealing with missing values through various transformations, as discussed in Section 7.4.

Outliers

An outlier is an observation that has values for the variables that are quite different from most other observations. Typically, an outlier appears at the maximum or minimum end of a variable and is so large or small that it skews or otherwise distorts the distribution. It is not uncommon to have a single instance or a very small number of these outlier values when compared to the frequency of other values of the variable. When summarising our data, performing tests on the data, and in building models, outliers can have an adverse impact on the quality of the results.

Hawkins (1980) captures the concept of an outlier as

> *an observation that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism.*

Outliers can be thought of as exceptional cases. Examples might include extreme weather conditions on a particular day, a very wealthy person who financially is very different from the rest of the population, and so on. Often, an outlier may be interesting but not really a key observation for our analysis. Sometimes outliers are the rare events that we are specifically interested in.

We may be interested in rare, unusual, or just infrequent events in a data mining context when considering fraud in income tax, insurance, and on-line banking, as well as for marketing.

Identifying whether an observation is an outlier is quite difficult, as it depends on the context and the model to be built. Perhaps under one context an observation is an outlier but under another context it might be a typical observation. The decision of what an outlier is will also vary by application and by user.

General outlier detection algorithms include those that are based on distance, density, projections, or distributions. The distance-based approaches are common in data mining, where an outlier is identified based on an observation's distance from nearby observations. The number of nearby observations and the minimum distance are two parameters. Another common approach is to assume a known distribution for the data. We then consider by how much an observation deviates from the distribution.

Many more recent model builders (including random forests and support vector machines) are very robust to outliers in that outliers tend not to adversely affect the algorithm. Linear regression type approaches tend to be affected by outliers.

One approach to dealing with outliers is to remove them from the dataset altogether. However, identifying the outlier remains an issue.

Variable Selection

Variable selection is another approach that can result in improved modelling. By removing irrelevant variables from the modelling process, the resulting models can be made more robust. Of course, it takes a good knowledge of the dataset and an understanding of the relevance of variables to the problem at hand. Some variables will also be found to be quite related to other variables, creating unnecessary noise when building models.

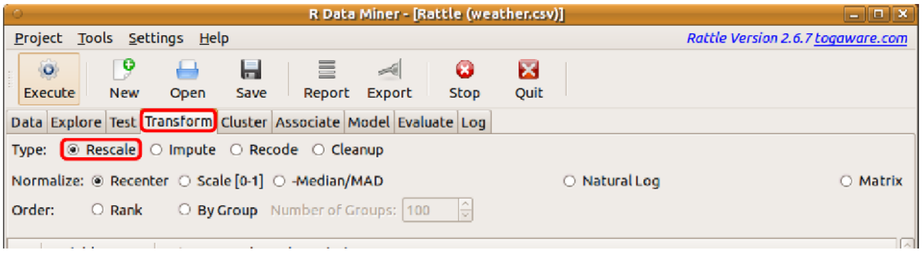
Various techniques can be used for variable selection. Simple techniques include considering different subsets of variables to explore for a subset that provides the best results. Other approaches use modelling measures (such as the information measure of decision tree induction discussed in Chapter 11) to identify the more important collection of variables.

A variety of other techniques are available. Approaches like principal components analysis and the variable importance measures of random forests and boosting can guide the choice of variables for building models.

7.2 Transforming Data

With the plethora of issues that we find in data, there is quite a collection of approaches for transforming data to improve our ability to discover knowledge. Cleaning our dataset and creating new variables from other variables in the dataset occupies much of our time as data miners. A programming language like R provides support for most of the myriad of approaches possible.

Rattle's Transform tab (Figure 7.1) provides many options for transforming datasets using many of the more common transformations. This includes normalising our data, filling in missing values, turning numeric variables into categoric variables and vice versa, dealing with outliers, and removing variables or observations with missing values. For the more complex transformations, we can revert to using R.


R Data Miner - [Rattle (weather.csv)]
Project Tools Settings Help
Rattle Version 2.6.7 togaware.com
Execute New Open Save Report Export Stop Quit
Data Explore Test Transform Cluster Associate Model Evaluate Log

Type: [✓ Rescale] Impute Recode Cleanup

Normalize: [✓ Recenter] Scale [0-1] -Median/MAD Natural Log Matrix

Order: Rank By Group Number of Groups: 100

Figure 7.1: The Transform tab options.

We now introduce the various transformations supported by Rattle. In tuning our dataset, we will often transform it in many different ways. This often represents quite a lot of work, and we need to capture the resulting data in some form.

Once the dataset is transformed, we can save the new version to a CSV file. We do this by clicking on the Export button whilst viewing the Transform (or the Data) tab. This will prompt us for a CSV filename under which the current transformed dataset will be saved. We can also save the whole current state of Rattle as a project, which can easily be reloaded at a later time.

Another option, and one to be encouraged as good practise, is to save to a script file the series of transformations as recorded in the Log tab. Saving these to a script file means we can automate the generation of the transformed dataset from the original dataset. The automatically transformed dataset can then be used for building models or for scoring. For scoring (i.e., applying a model to a new collection of data), we can simply change the name of the original source data file within the script. The data is then processed through the R script and we can then apply our model to this new dataset within R.

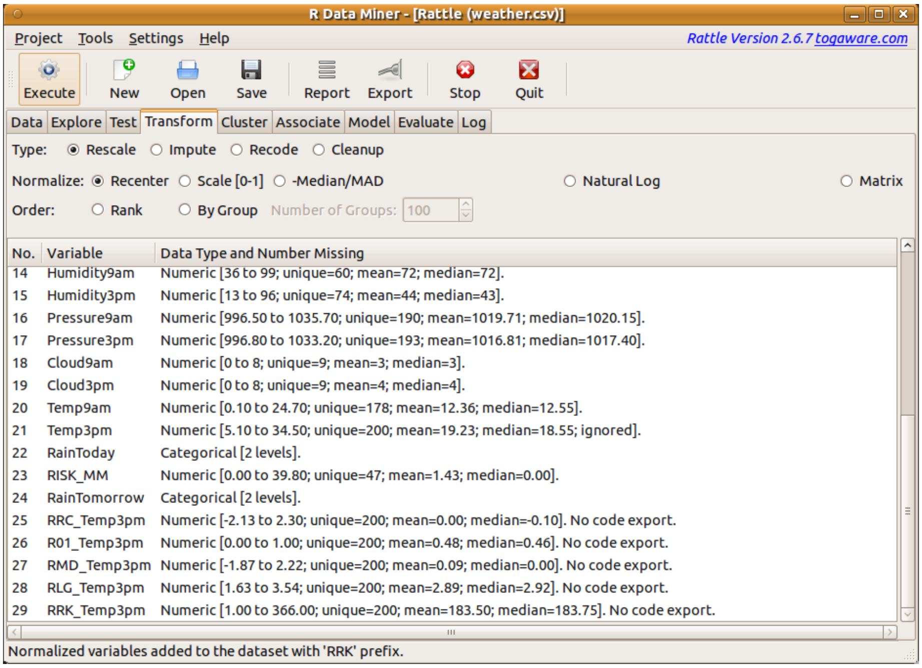
The remainder of this chapter introduces each of the classes of transformations that are typical of a data mining project and supported by Rattle.

7.3 Rescaling Data

Different model builders will have different assumptions on the data from which the models are built. When building a cluster using any kind of distance measure, for example, we may need to ensure all variables have approximately the same scale. Otherwise, a variable like `Income` will overwhelm a variable like `Age` when calculating distances. A distance of 10 "years" may be more significant than a distance of $10,000, yet 10000 swamps 10 when they are added together, as would be the case when calculating distances without rescaling the data.

In these situations, we will want to normalise our data. The types of normalisations (available through the Normalise option of the Transform tab) we can perform include recentering and rescaling our data to be around zero (Recenter uses a so-called Z score, which subtracts the mean and divides by the standard deviation), rescaling our data to be in the range from 0 to 1 (Scale [0–1]), performing a robust rescaling around zero using the median (Median/MAD), applying `log()` to our data, or transforming multiple variables with one divisor (Matrix). The details of these transformations will be presented below.

Other rescaling transformations include converting the numbers into a rank ordering (Rank) and performing a transform to rescale a variable according to some group that the observation belongs to (By Group).


R Data Miner - [Rattle (weather.csv)]
Project Tools Settings Help
Rattle Version 2.6.7 togaware.com
Execute New Open Save Report Export Stop Quit
Data Explore Test Transform Cluster Associate Model Evaluate Log

Type: [✓ Rescale] Impute Recode Cleanup

Normalize: [✓ Recenter] Scale [0-1] -Median/MAD Natural Log Matrix

Order: Rank By Group Number of Groups: 100

No.	Variable	Data Type and Number Missing
14	Humidity9am	Numeric [36 to 99; unique=60; mean=72; median=72].
15	Humidity3pm	Numeric [13 to 96; unique=74; mean=44; median=43].
16	Pressure9am	Numeric [996.50 to 1035.70; unique=190; mean=1019.71; median=1020.15].
17	Pressure3pm	Numeric [996.80 to 1033.20; unique=193; mean=1016.81; median=1017.40].
18	Cloud9am	Numeric [0 to 8; unique=9; mean=3; median=3].
19	Cloud3pm	Numeric [0 to 8; unique=9; mean=4; median=4].
20	Temp9am	Numeric [0.10 to 24.70; unique=178; mean=12.36; median=12.55].
21	Temp3pm	Numeric [5.10 to 34.50; unique=200; mean=19.23; median=18.55; ignored].
22	RainToday	Categorical [2 levels].
23	RISK_MM	Numeric [0.00 to 39.80; unique=47; mean=1.43; median=0.00].
24	RainTomorrow	Categorical [2 levels].
25	RRC_Temp3pm	Numeric [-2.13 to 2.30; unique=200; mean=0.00; median=-0.10]. No code export.
26	R01_Temp3pm	Numeric [0.00 to 1.00; unique=200; mean=0.48; median=0.46]. No code export.
27	RMD_Temp3pm	Numeric [-1.87 to 2.22; unique=200; mean=0.09; median=0.00]. No code export.
28	RLG_Temp3pm	Numeric [1.63 to 3.54; unique=200; mean=2.89; median=2.92]. No code export.
29	RRK_Temp3pm	Numeric [1.00 to 366.00; unique=200; mean=183.50; median=183.75]. No code export.

Normalized variables added to the dataset with 'RRK' prefix.

Figure 7.2: Transforming `Temp3pm` in five different ways.

Figure 7.2 shows the result of transforming the variable `Temp3pm` in five different ways. The simple summary that we can see for each variable in Figure 7.2 provides a quick view of how the data has been transformed. For example, the recenter transform of the variable `Temp3pm` has changed the range of values for the variable from the original 5.10 to 34.50 to end up with −2.13 to 2.30.

Tip: *Notice, as we see in Figure 7.2, that the original data is not modified. Instead, a new variable is created for each transform with a prefix added to the variable's name that indicates the kind of transformation. The prefixes are `RRC_` (for Recenter), `R01_` (for Scale [0–1]), `RMD_` (for Median/MAD), `RLG_` (for Log), and `RRK_` (for Rank).*

Figure 7.3 illustrates the effect of the four transformations on the variable `Temp3pm` compared with the original distribution of the data. The top left plot shows the original distribution. Note that the three normalisations (recenter, rescale 0–1, and recenter using the median/-MAD) all produce new variables with very similar looking distributions. The log transform changes the distribution quite significantly. The rank transform simply delivers a variable with a flat distribution since the new variable simply consists of a sequence of integers and thus each value of the new variable appears just once.

Recenter

This is a common normalisation that re-centres and rescales our data. The usual approach is to subtract the mean value of a variable from each observation's value of the variable (to recentre the variable) and then divide the values by their standard deviation (calculating the square root of the sum of squares), which rescales the variable back to a range within a few integer values around zero.

To demonstrate the transforms on our *weather*, we will load **rattle** and create a copy of the dataset, to be referred to as `ds`:

```
> library(rattle)
> ds <- weather
```

The following R code can then perform the transformation using `scale()`:

```
> ds$RRC_Temp3pm <- scale(ds$Temp3pm)
```

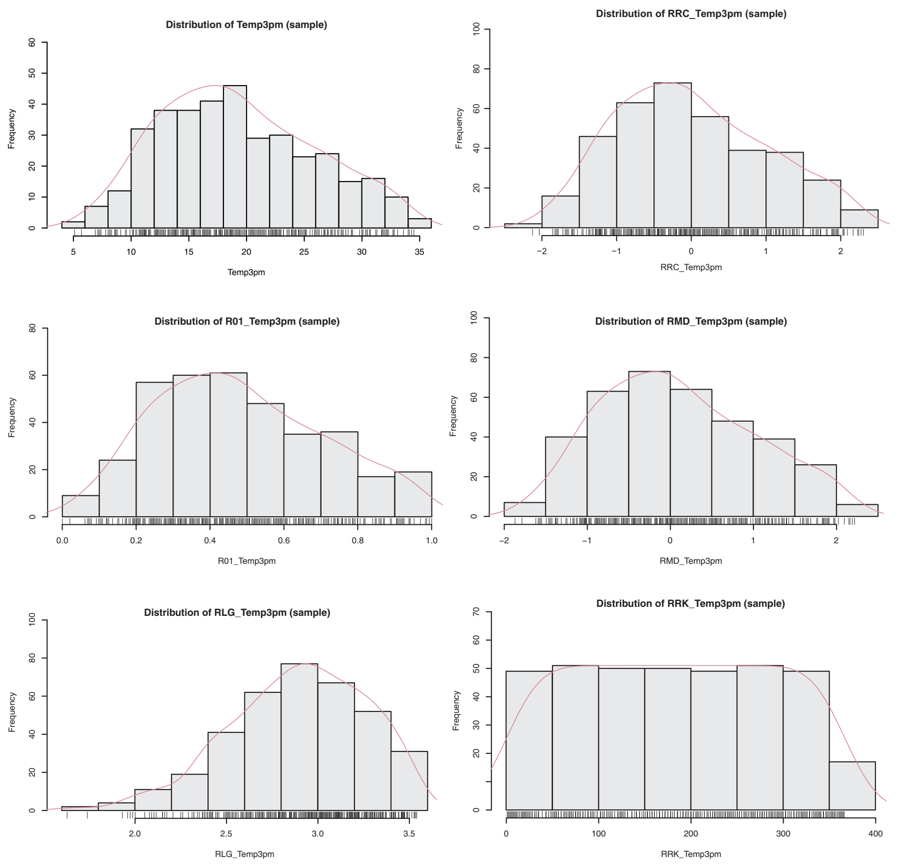


Figure 7.3: Comparing distributions after transforming. From the left to right, top to bottom: original, recenter, rescale to 0–1, rank, log transform, and recenter using median/MAD.

Scale [0–1]

Rescaling so that our data has a mean around zero might not be so intuitive for variables that are never negative. Most numeric variables from the *weather* dataset naturally only take on positive values, including `Rainfall` and `WindSpeed3pm`.

To rescale whilst retaining only positive values, we might choose the Scale [0–1] transform, which simply recodes the data so that the values are all between 0 and 1. This is done by subtracting the minimum value from the variable's value for each observation and then dividing by the difference between the minimum and the maximum values.

The following R code is used to perform the transformation. We use `rescaler()` from **reshape** (Wickham, 2007):

```
> library(reshape)
> ds$R01_Temp3pm <- rescaler(ds$Temp3pm, "range")
```

Median/MAD

This option for recentring and rescaling our data is regarded as a robust (to outliers) version of the standard Recenter option. Instead of using the mean and standard deviation, we subtract the median and divide by the so-called median absolute deviation (MAD).

The following R code is used to perform the transformation. Again we use `rescaler()` from **reshape**:

```
> library(reshape)
> ds$RMD_Temp3pm <- rescaler(ds$Temp3pm, "robust")
```

Natural Log

Often the values of a variable can be quite skewed in one direction or another. A typical example is `Income`. The majority of a population may have incomes below $150,000. But there are a relatively small number of individuals with excessive incomes measured in the millions of dollars. In many approaches to analysis and model building, these extreme values (outliers) can adversely affect any analysis.

Logarithm transforms map a very broad range of (positive) numeric values into a narrower range of (positive) numeric values. The natural log function effectively reduces the spread of the values of the variable. This is particularly useful when we have outliers with extremely large values compared with the rest of the population.

Logarithms can use a so called *base* with respect to which they do the transformation. We can use a base 10 transform to explain what the transform does. With a $\texttt{log}_{10}$ transform, a salary of $10,000 is recoded as 4, $100,000 as 5, $150,000 as 5.17609125905568, and $1,000,000 as 6—that is, a logarithm of base 10 recodes each power of 10 (e.g., 10^5 or 100,000) to the power itself (e.g., 5) and similarly for a logarithm of base 2, which recodes 8 (which is 2^3) to 3.

By default, Rattle simply uses the natural logarithm for its transform. This recodes using a logarithm to base *e*, where *e* is the special number 2.718.... This is the default base that R uses for `log()`. The following R code is used to perform the transformation. We also recode any resulting "infinite" values (e.g., `log`(0)) to be treated as missing values:

```
> ds$RLG_Temp3pm <- log(ds$Temp3pm)
> ds$RLG_Temp3pm[ds$RLG_Temp3pm == -Inf] <- NA
```

Rank

On some occasions, we are not interested in the actual value of the variable but rather in the relative position of the value within the distribution of that variable. For example, in comparing restaurants or universities, the actual score may be less interesting than where each restaurant or university sits compared with the others. A rank is then used to capture the relative position, ignoring the actual scale of any differences.

The Rank option will convert each observation's numeric value for the identified variable into a ranking in relation to all other observations in the dataset. A rank is simply a list of integers, starting from 1, that is mapped from the minimum value of the variable, progressing by integer until we reach the maximum value of the variable. The largest value is thus the sample size, which for the *weather* dataset is 366. A rank has an advantage over a recentring transform, as it removes any skewness from the data (which may or may not be appropriate for the task at hand).

A problem with recoding our data using a rank is that it becomes difficult when using the resulting model to score new observations. How do we rank a single observation? For example, suppose we have a model that tests whether the rank is less than 50 for the variable `Temp3pm`. What does this actually mean when we apply this test to a new observation? We might instead need to revert the rank back to an actual value to be useful in scoring.

The following R code is used to perform the transformation. Once again we use `rescaler()` from **reshape**:

```
> library(reshape)
> ds$RRK_Temp3pm <- rescaler(ds$Temp3pm, "rank")
```

By Group

A By Group transform recodes the values of a variable into a rank order between 0 and 100. A categoric variable can also be identified as part of the transformation. In this case, the observations are grouped by the values of the categoric variable. These groups are then considered as peers. The ranking is then performed with respect to the peers rather than the whole population. An example might be to rank wind speeds within groups defined by the wind direction. A high wind speed relative to one direction may not be a high wind speed relative to another direction. The code to do this gets a little complex.

```
> library(reshape)
> ds$RBG_SpeedByDir <- ds$WindGustSpeed
> bylevels <- levels(ds$WindGustDir)
> for (vl in bylevels)
  {
    grp <- sapply(ds$WindGustDir == vl, isTRUE)
    ds[grp, "RBG_SpeedByDir"] <-
      round(rescaler(ds[grp, "WindGustSpeed"],
                     "range") * 99)
  }
> ds[is.nan(ds$RBG_SpeedByDir), "RBG_SpeedByDir"] <- 50
> v <- c("WindGustSpeed", "WindGustDir", "RBG_SpeedByDir")
```

We can then selectively display some observations:

```
> head(ds[ds$WindGustDir %in% c("NW", "SE"), v], 10)
```

Observation 1, for example, with a `WindGustSpeed` of 30, is at the 18th percentile within all those observations for which `WindGustDir` is `NW`. Overall, we might observe that the `WindGustSpeed` is generally less when the `WindGustDir` is `SE` as compared with `NW`, looking at the rankings within each group. Instead of generating a rank of between 0 and 100, a Z score (i.e., Recenter) could be used to recode within each group. This would require only a minor change to the R code above.

Summary

We summarise this collection of transformations of the first few observations of the variable `Temp3pm`:

Obs.	WindGustSpeed	WindGustDir	RBG_SpeedByDir
1	30	NW	18
3	85	NW	83
4	54	NW	47
6	44	SE	86
7	43	SE	83
14	44	NW	35
15	41	NW	31
29	39	SE	70
33	50	NW	42
34	50	NW	42

Obs.	Temp3pm	RRC_	R01_	RMD_	RLG_	RRK_
1	23.60	0.66	0.63	0.70	3.16	268
2	25.70	0.97	0.70	0.99	3.25	295
3	20.20	0.15	0.51	0.23	3.01	217
4	14.10	−0.77	0.31	−0.62	2.65	92
5	15.40	−0.58	0.35	−0.44	2.73	117
6	14.80	−0.67	0.33	−0.52	2.69	106

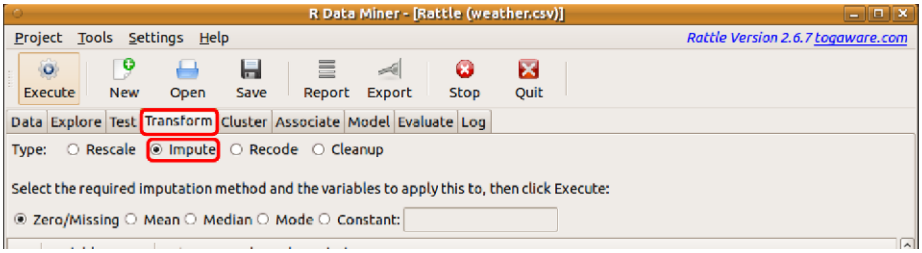
7.4 Imputation

Imputation is the process of filling in the gaps (or missing values) in data. Data is missing for many different reasons, and it is important to understand why. This will guide us in dealing with the missing values. For rainfall variables, for example, a missing value may mean there was no rain recorded on that day, and hence it is really a surrogate for 0 mm of rain. Alternatively, perhaps the measuring equipment was not functioning that day and hence recorded no rain.

Imputation can be questionable because, after all, we are inventing data. We won't discuss here the pros and cons in any detail, but note that, despite such concerns, reasonable results can be obtained from simple imputations.

There are many types of imputations available, only some of which are directly available in Rattle. Imputation might involve simply replacing missing values with a particular value. This then allows, for example, linear regression models to be built using all observations. Or we might add an additional variable to record when values are missing. This then allows the model builder to identify the importance of the missing values, for example. We do note, however, that not all model builders (e.g., decision trees) are troubled by missing values.

Figure 7.4 shows Rattle's Impute option on the Transform tab selected with the choices for imputation, including Zero/Missing, Mean, Median, Mode, and Constant.


R Data Miner - [Rattle (weather.csv)]
Project Tools Settings Help
Rattle Version 2.6.7 togaware.com
Execute New Open Save Report Export Stop Quit
Data Explore Test Transform Cluster Associate Model Evaluate Log

Type: ○ Rescale [✓ Impute] ○ Recode ○ Cleanup

Select the required imputation method and the variables to apply this to, then click Execute:

[✓ Zero/Missing] ○ Mean ○ Median ○ Mode ○ Constant:

Figure 7.4: The Transform tab with the Impute option selected.

When Rattle performs an imputation, it will store the results in a new variable within the same dataset. The new variable will have the same name as the variable that is imputed, but prefixed with either IZR_, IMN_, IMD_, IMO_, or ICN_. Such variables will automatically be identified as having an Input role, whilst the original variable will have a role of Ignore.

Zero/Missing

The simplest imputations involve replacing all missing values for a variable with a single value. This makes the most sense when we know that the missing values actually indicate that the value is 0 rather than unknown. For example, in a taxation context, if a taxpayer does not provide a value for a specific type of deduction, then we might assume that they intend it to be zero. Similarly, if the number of children in a family is not recorded, it could be a reasonable assumption to assume it is zero.

For categoric data, the simplest approach to imputation is to replace missing values with a special value, such as *Missing*. The following R code is used to perform the transformation:

```
> ds$IZR_Sunshine <- ds$Sunshine
> ds$IZR_Sunshine[is.na(ds$IZR_Sunshine)] <- 0
```

Mean/Median/Mode

Often a simple, if not always satisfactory, choice for missing values that are known not to be zero is to use some "central" value of the variable. This is often the mean, median, or mode, and thus usually has limited impact on the distribution. We might choose to use the mean, for example, if the variable is otherwise generally normally distributed (and in particular does not have any skewness). If the data does exhibit some skewness, though (e.g., there are a small number of very large values), then the median might be a better choice.

For categoric variables, there is, of course, no mean or median, and so in such cases we might (but with care) choose to use the mode (the most frequent value) as the default to fill in for the otherwise missing values. The mode can also be used for numeric variables. This could be appropriate for variables that are dominated by a single value. Perhaps we notice that predominately (e.g., for 80% of the observations) the temperature at 9 am is 26 degrees Celsius. That could be a reasonable choice for any missing values.

Whilst this is a simple and computationally quick approach, it is a very blunt approach to imputation and can lead to poor performance from the resulting models. However, it has also been found empirically to be useful. The following R code is used to perform the transformation:

```
> ds$IMN_Sunshine <- ds$Sunshine
> ds$IMN_Sunshine[is.na(ds$IMN_Sunshine)] <-
    mean(ds$Sunshine, na.rm=TRUE)
```

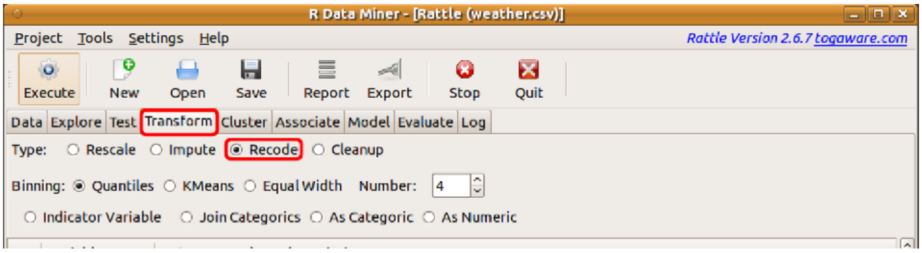
Constant

This choice allows us to provide our own default value to fill in the gaps. This might be an integer or real number for numeric variables, or else a special marker or the choice of something other than the majority category for categoric variables. The following R code is used to perform the transformation:

```
> ds$IZR_Sunshine <- ds$Sunshine
> ds$IZR_Sunshine[is.na(ds$IZR_Sunshine)] <- 0
```

7.5 Recoding

The Recode option on the Transform tab provides numerous remapping operations, including binning and transformations of the type of the data. Figure 7.5 lists the options.


R Data Miner - [Rattle (weather.csv)]
Project Tools Settings Help
Rattle Version 2.6.7 togaware.com
Execute New Open Save Report Export Stop Quit
Data Explore Test Transform Cluster Associate Model Evaluate Log

Type: ○ Rescale ○ Impute [✓ Recode] ○ Cleanup

Binning: [✓ Quantiles] ○ KMeans ○ Equal Width Number: 4

○ Indicator Variable ○ Join Categorics ○ As Categoric ○ As Numeric

Figure 7.5: The Transform tab with the Recode option selected.

Binning

Binning is the operation of transforming a continuous numeric variable into a specific set of categoric values based on the numeric values. Simple examples include converting an age into an age group, and a temperature into `Low`, `Medium`, and `High`. Performing a binning transform may lose valuable information, so do give some thought as to whether binning is appropriate.

Binning can be useful in simplifying models. It is also useful when we visualise data. A mosaic plot (Chapter 5), for example, is only useful for categoric data, and so we could turn `Sunshine` into a categoric variable by binning. Binning can also be useful to set a numeric value as the stratifying variable in various plots in Chapter 5. For example, we could bin `Temp9am` and then choose the new `BE4_Temp9am` (`BE4` for binning into four equal-size bins) as the Target and generate a Box Plot from the Explore tab to see the relationship with the `Evaporation`.

Rattle supports automated binning through the use of `binning()` (provided by Daniele Medri). The Rattle interface provides an option to choose between Quantile (or equal count) binning, KMeans binning, and Equal Width binning. For each option, the default number of bins is four. We can change this to suit our needs. The variables generated are prefixed with either `BQn_`, `BKn_`, or `BEn_`, respectively, with `n` replaced by the number of bins.

Indicator Variables

Some model builders often do not directly handle categoric variables. This is typical of distance-based model builders such as k-means clustering, as well as the traditional numeric regression types of models.

A simple approach to transforming a categoric variable into a numeric one is to construct a collection of so-called *indicator* or *dummy* variables. For each possible value of the categoric variable, we can create a new variable that will have the value 1 for any observation that has this categoric value and 0 otherwise. The result is a collection of new numeric variables, one for each of the possible categoric values. An example might be the categoric variable `Colour`, which might only allow the possible values of `Red`, `Green`, or `Blue`. This can be converted to three variables, `Colour_Red`, `Colour_Green`, and `Colour_Blue`. Only one of these will have the value 1 at any time, whilst the other(s) will have the value 0.

Rattle's Transform tab provides an option to transform a categoric variable into a collection of indicator variables. Each of the new variables has a name that is prefixed by `TIN_`. The remainder of the name is made up of the original name of the categoric variable (e.g., `Colour`) and the particular value (e.g., `Red`). This will give, for example, `TIN_Colour_Red` as one of the new variable names. Table 7.1 illustrates how the recoding works for a collection of observations.

Table 7.1: Examples of recoding a single categoric variable as a number of numeric indicator variables.

Obs.	Colour	Colour_Red	Colour_Green	Colour_Blue
1	Green	0	1	0
2	Blue	0	0	1
3	Blue	0	0	1
4	Red	1	0	0
5	Green	0	1	0
6	Red	1	0	0

In terms of modelling, for a categoric variable with k possible values, we only need to convert it to $k-1$ indicator variables. The kth indicator variable is redundant and in fact is directly determined by the values of the other $k-1$ indicators. If all of the other indicators are 0, then clearly the kth will be 1. Similarly if any of the other $k-1$ indicators is 1, then the kth must be 0. Consequently, we should only include all but one of the new indicator variables as having an Input role. Rattle, by default, will set the role of the first new indicator variable to be Ignore.

There is not always a need to transform a categoric variable. Some model builders, like the Linear model builder in Rattle, will do it automatically.

Join Categorics

The Join Categorics option provides a convenient way to stratify the dataset based on multiple categoric variables. It is a simple mechanism that creates a new variable from the combination of all of the values of the two constituent variables selected in the Rattle interface. The resulting variables are prefixed with `TJN_` and include the names of both the constituent variables.

A simple example might be to join `RainToday` and `RainTomorrow` to give a new variable (`TJN` here and `TJN_RainToday_RainTomorrow` in Rattle):

```
> ds$TJN <- interaction(paste(ds$RainToday, "_",
                              ds$RainTomorrow, sep=""))
> ds$TJN[grepl("^NA_|_NA$", ds$TJN)] <- NA
> ds$TJN <- as.factor(as.character(ds$TJN))
> head(ds[c("RainToday", "RainTomorrow", "TJN")])

  RainToday RainTomorrow     TJN
1        No          Yes  No_Yes
2       Yes          Yes Yes_Yes
3       Yes          Yes Yes_Yes
4       Yes          Yes Yes_Yes
5       Yes           No  Yes_No
6        No           No   No_No
```

We might also want to join a numeric variable and a categoric variable, like the common `Age` and `Gender` stratification. To do this, we first use the Binning option within Recode to categorise the `Age` variable and then use Join Categorics.

Type Conversion

The As Categoric and As Numeric options will, respectively, convert a numeric variable to categoric (with the new categoric variable name prefixed with TFC_) and vice versa (with the new numeric variable name prefixed with TNM_). The R code for these transforms uses `as.factor()` and `as.numeric()`:

```
> ds$TFC_Cloud3pm <- as.factor(ds$Cloud3pm)
> ds$TNM_RainToday <- as.numeric(ds$RainToday)
```

7.6 Cleanup

It is quite easy to get our dataset variable count up to significant numbers. The Cleanup option allows us to tell Rattle to actually delete columns from the dataset. Thus, we can perform numerous transformations and then save the dataset back into a CSV file (using the Export option).

Various Cleanup options are available. These allow us to remove any variable that is ignored (Delete Ignored), remove any variables we select (Delete Selected), or remove any variables that have missing values (Delete Missing). The Delete Obs with Missing option will remove observations (rather than variables—i.e., remove rows rather than columns) that have missing values.

7.7 Command Summary

This chapter has referenced the following R packages, commands, functions, and datasets:

`as.factor()`	function	Convert variable to be categoric.
`as.numeric()`	function	Convert variable to be numeric.
`is.na()`	function	Identify which values are missing.
`levels()`	function	List the values of a categoric variable.
`log()`	function	Logarithm of a numeric variable.
`mean()`	function	Mean value of a numeric variable.
`rescaler()`	function	Remap numeric variables.

reshape	package	Transform variables in various ways.
`scale()`	function	Remap numeric variables.