Reshaping Data with the reshape Package

Hadley Wickham

Iowa State University

Abstract

This paper presents the reshape package for R, which provides a common framework for many types of data reshaping and aggregation. It uses a paradigm of 'melting' and 'casting', where the data are 'melted' into a form which distinguishes measured and identifying variables, and then 'cast' into a new shape, whether it be a data frame, list, or high dimensional array. The paper includes an introduction to the conceptual framework, practical advice for melting and casting, and a case study.

Keywords: data reshaping, contingency table, tabulation, aggregation, R.

1. Introduction

Reshaping data is a common task in real-life data analysis, and it is usually tedious and frustrating. You've struggled with this task in **Excel**, in SAS, and in R: how do you get your clients' data into the form that you need for summary and analysis? This paper describes version 0.8.1 of the reshape package for R (R Development Core Team 2007), which presents a new approach that aims to reduce the tedium and complexity of reshaping data.

Data often has multiple levels of grouping (nested treatments, split plot designs, or repeated measurements) and typically requires investigation at multiple levels. For example, from a long term clinical study we may be interested in investigating relationships over time, or between times or patients or treatments. To make your job even more difficult, the data probably has been collected and stored in a way optimised for ease and accuracy of collection, and in no way resembles the form you need for statistical analysis. You need to be able to fluently and fluidly reshape the data to meet your needs, but most software packages make it difficult to generalise these tasks, and new code needs to be written for each new case.

While you are probably familiar with the idea of reshaping, it is useful to be a little more formal. Data reshaping involves a rearrangement of the form, but not the content, of the data. Reshaping is a little like creating a contingency table, as there are many ways to arrange the

same data, but it is different in that there is no aggregation involved. The tools presented in this paper work equally well for reshaping, retaining all existing data, and aggregating, summarising the data, and later we will explore the connection between the two.

In R, there are a number of general functions that can aggregate data, for example `tapply`, `by` and `aggregate`, and a function specifically for reshaping data, `reshape`. Each of these functions tends to deal well with one or two specific scenarios, and each requires slightly different input arguments. In practice, you need careful thought to piece together the correct sequence of operations to get your data into the form that you want. The reshape package grew out of my frustrations with reshaping data for consulting clients, and overcomes these problems with a general conceptual framework that uses just two functions: `melt` and `cast`.

The paper introduces this framework, which will help you think about the fundamental operations that you perform when reshaping and aggregating data, but the main emphasis is on the practical tools, detailing the many forms of data that `melt` can consume and that `cast` can produce. A few other useful functions are introduced, and the paper concludes with a case study, using reshape in a real-life example.

2. Conceptual framework

To help us think about the many ways we might rearrange a data set, it is useful to think about data in a new way. Usually, we think about data in terms of a matrix or data frame, where we have observations in the rows and variables in the columns. For the purposes of reshaping, we can divide the variables into two groups: identifier and measured variables.

1. Identifier (id) variables identify the unit that measurements take place on. Id variables are usually discrete, and are typically fixed by design. In ANOVA notation (Y_{ijk}), id variables are the indices on the variables (i, j, k); in database notation, id variables are a composite primary key.

2. Measured variables represent what is measured on that unit (Y).

It is possible to take this abstraction one step further and say there are only id variables and a value, where the id variables also identify what measured variable the value represents. For example, we could represent this data set, which has two id variables (subject and time):

	subject	time	age	weight	height
1	John Smith	1	33	90	2
2	Mary Smith	1			2

as:

	subject	time	variable	value
1	John Smith	1	age	33
2	John Smith	1	weight	90
3	John Smith	1	height	2
4	Mary Smith	1	height	2

where each row now represents one observation of one variable. This operation is called melting and produces 'molten' data. Compared to the original data set, the molten data has a new id variable 'variable', and a new column 'value', which represents the value of that observation. We now have the data in a form in which there are only id variables and a value.

From this form, we can create new forms by specifying which variables should form the columns and rows. In the original data frame, the 'variable' id variable forms the columns, and all identifiers form the rows. We do not have to specify all the original id variables in the new form. When we do not, the combination of id variables will no longer identify one value, but many, and we will aggregate the data as well as reshaping it. The function that reduces these many numbers to one is called an aggregation function.

The following section describes the melting operation in detail, as implemented in the reshape package.

3. Melting data

Melting a data frame is a little trickier in practice than it is in theory. This section describes the practical use of the `melt` function in R.

In R, melting is a generic operation that can be applied to different data storage objects including data frames, arrays and matrices. This section describes the most common case, melting a data frame. Reshape also provides support for less common data structures, including high-dimensional arrays and lists of data frames or matrices. The built-in documentation for melt, `?melt`, lists all objects that can be melted, and provides links to more details. `?melt.data.frame` documents the most common case of melting a data frame.

The `melt` function needs to know which variables are measured and which are identifiers. This distinction should be obvious from your design: if you fixed the value, it is an id variable. If you do not specify them explicitly, `melt` will assume that any factor or integer column is an id variable. If you specify only one of measured and identifier variables, `melt` assumes that all the other variables are the other sort. For example, with the `smiths` dataset, as shown above, all the following calls have the same effect:

```
R> melt(smiths, id = c("subject", "time"), measured = c("age", "weight",
+     "height"))
R> melt(smiths, id = c("subject", "time"))
R> melt(smiths, id = 1:2)
R> melt(smiths, measured = c("age", "weight", "height"))
R> melt(smiths)

R> melt(smiths)
     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
```

If you want to run these functions yourself, the `smiths` dataset is included in the **reshape** package.

Melt does not make many assumptions about your measured and id variables: there can be any number, in any order, and the values within the columns can be in any order too. In the current implementation, there is only one assumption that `melt` makes: all measured values must be of the same type, e.g., numeric, factor, date. We need this assumption because the molten data is stored in a R data frame, and the value column can be only one type. Most of the time this is not a problem as there are few cases where it makes sense to combine different types of variables in the cast output.

3.1. Melting data with id variables encoded in column names

A more complicated case is where the variable names contain information about more than one variable. For example, here we have an experiment with two treatments (A and B) with data recorded on two time points (1 and 2), and the column names represent both the treatment and the time at which the measurement was taken. It is important to make these implicit variables explicit as part of the data preparation process, as it ensures all id variables are represented in the same way.

```
R> trial <- data.frame(id = factor(1:4), A1 = c(1, 2, 1, 2), A2 = c(2,
+    1, 2, 1), B1 = c(3, 3, 3, 3))
R> (trialm <- melt(trial))
   id variable value
1   1       A1     1
2   2       A1     2
3   3       A1     1
4   4       A1     2
5   1       A2     2
6   2       A2     1
7   3       A2     2
8   4       A2     1
9   1       B1     3
10  2       B1     3
11  3       B1     3
12  4       B1     3
```

To fix this we need to create a time and treatment column after melting:

```
R> (trialm <- cbind(trialm, colsplit(trialm$variable,
+    names = c("treatment", "time"))))
   id variable value treatment time
1   1       A1     1         A    1
2   2       A1     2         A    1
3   3       A1     1         A    1
4   4       A1     2         A    1
5   1       A2     2         A    2
6   2       A2     1         A    2
```

```
7   3        A2    2        A    2
8   4        A2    1        A    2
9   1        B1    3        B    1
10  2        B1    3        B    1
11  3        B1    3        B    1
12  4        B1    3        B    1
```

This uses the `colsplit` function described in `?colsplit`, which deals with the simple case where variable names are concatenated together with some separator. In general variable names can be constructed in many different ways and may need a custom regular expression to tease apart multiple components.

3.2. Already molten data

Sometimes your data may already be in molten form. In this case, all that is necessary is to ensure that the value column is named 'value'. See `?rename` for one way to do this.

3.3. Missing values in molten data

Finally, it is important to discuss what happens to missing values when you melt your data. Explicitly coded missing values usually denote sampling zeros rather than structural missings, which are usually implicit in the data. Clearly a structural missing depends on the structure of the data and as we are changing the structure of the data, we might expect some changes to structural missings. Structural missings change from implicit to explicit when we change from a nested to a crossed structure. For example, imagine a dataset with two id variables, sex (male or female) and pregnant (yes or no). When the variables are nested (ie. both on the same dimension) then the missing value 'pregnant male' is encoded by its absence. However, in a crossed view, we need to add an explicit missing as there will now be a cell which must be filled with something. This is illustrated below:

	sex	pregnant	value
1	male	no	10
2	female	no	14
3	female	yes	4

	sex	no	yes
1	female	14	4
2	male	10	

In this vein, one way to describe the molten form is that it is perfectly nested: there are no crossings. For this reason, it is possible to encode all missing values implicitly, by omitting that combination of id variables, rather than explicitly, with an `NA` value. However, you may expect these to be in the data frame, and it is a bad idea for a function to throw data away by default, so you need to explicitly state that implicit missing values are ok. In most cases it is safe to get rid of them, which you can do by using `na.rm = TRUE` in the call to `melt`. The two different results are illustrated below.

```
R> melt(smiths)
    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

R> melt(smiths, na.rm = TRUE)
     subject time variable value
1 John Smith    1      age 33.00
2 John Smith    1   weight 90.00
3 John Smith    1   height  1.87
4 Mary Smith    1   height  1.54
```

If you do not use `na.rm = TRUE` you will need to make sure to account for possible missing values when aggregating (Section 4.4, page 11), for example, by supplying `na.rm = TRUE` to `mean`, `sum` or `var`.

4. Casting molten data

Once you have your data in the molten form, you can use `cast` to rearrange it into the shape that you want. The `cast` function has two required arguments:

- `data`: the molten data set to reshape

- `formula`: the casting formula which describes the shape of the output format (if you omit this argument, cast will return the data frame in the classic form with measured variables in the columns, and all other id variables in the rows)

Most of this section explains the different casting formulas you can use. It also explains the use of two other optional arguments to cast:

- `fun.aggregate`: aggregation function to use, if necessary

- `margins`: what marginal values should be computed

4.1. Basic use

The casting formula has this basic form `col_var_1 + col_var_2 ~ row_var_1 + row_var_2`. This describes which variables you want to appear in the columns and which in the rows. These variables need to come from the molten data frame or be one of the following special variables:

- `.` corresponds to no variable, useful when creating formulas of the form `. ~ x` or `x ~ .`, that is, a single row or column.

- `...` represents all variables not already included in the casting formula. Including this in your formula will guarantee that no aggregation occurs. There can be only one `...` in a cast formula.

- `result_variable` is used when your aggregation formula returns multiple results. See Section 4.6, page 14, for more details.

The first set of examples illustrate reshaping: all the original variables are used. The first two examples show two ways to put the data back into its original form, with measured variables in the columns and all other id variables in the rows. The second and third examples are variations of this form where we put subject and then time in the columns.

```
R> cast(smithsm, time + subject ~ variable)
  time    subject age weight height
1    1 John Smith  33     90   1.87
2    1 Mary Smith  NA     NA   1.54

R> cast(smithsm, ... ~ variable)
     subject time age weight height
1 John Smith    1  33     90   1.87
2 Mary Smith    1  NA     NA   1.54

R> cast(smithsm, ... ~ subject)
  time variable John Smith Mary Smith
1    1      age      33.00         NA
2    1   weight      90.00         NA
3    1   height       1.87       1.54

R> cast(smithsm, ... ~ time)
     subject variable     1
1 John Smith      age 33.00
2 John Smith   weight 90.00
3 John Smith   height  1.87
4 Mary Smith   height  1.54
```

Because of the limitations of R data frames, it is not possible to label the columns and rows completely unambiguously. For example, note the last three examples where the data frame has no indication of the name of the variable that forms the columns. Additionally, some data values do not make valid column names, e.g., 'John Smith'. To use these within R, you often need to surround them with backticks, e.g., df$`John Smith`.

The following examples demonstrate aggregation. Aggregation occurs when the combination of variables in the cast formula does not identify individual observations. In this case an aggregation function reduces the multiple values to a single one. See Section 4.4, page 11, for more details. These examples use the french fries dataset included in the `reshape` package. It is data from a sensory experiment on french fries, where different types of frier oil, `treatment`, were tested by different people, `subject`, over ten weeks `time`.

The most severe aggregation is reduction to a single number, described by the cast formula `. ~ .`.

```
R> ffm <- melt(french_fries, id = 1:4, na.rm = TRUE)
```

```
R> cast(ffm, . ~ ., length)
  value (all)
1 (all)  3471
```

Alternatively, we can summarise by the values of a single variable, either in the rows or columns.

```
R> cast(ffm, treatment ~ ., length)
  treatment (all)
1         1 1159
2         2 1157
3         3 1155
```

```
R> cast(ffm, . ~ treatment, length)
  value    1    2    3
1 (all) 1159 1157 1155
```

The following casts show the different ways we can combine two variables: one each in row and column, both in row or both in column. When multiple variables appear in the column specification, their values are concatenated to form the column names.

```
R> cast(ffm, rep ~ treatment, length)
  rep   1   2   3
1   1 579 578 575
2   2 580 579 580
```

```
R> cast(ffm, treatment ~ rep, length)
  treatment   1   2
1         1 579 580
2         2 578 579
3         3 575 580
```

```
R> cast(ffm, treatment + rep ~ ., length)
  treatment rep (all)
1         1   1   579
2         1   2   580
3         2   1   578
4         2   2   579
5         3   1   575
6         3   2   580
```

```
R> cast(ffm, rep + treatment ~ ., length)
  rep treatment (all)
1   1         1   579
2   1         2   578
3   1         3   575
```

```
4   2            1   580
5   2            2   579
6   2            3   580

R> cast(ffm, . ~ treatment + rep, length)
    value 1_1 1_2 2_1 2_2 3_1 3_2
1 (all) 579 580 578 579 575 580
```

As illustrated above, the order in which the row and column variables are specified is very important. As with a contingency table, there are many possible ways of displaying the same variables, and the way that they are organised will reveal different patterns in the data. Variables specified first vary slowest, and those specified last vary fastest. Because comparisons are made most easily between adjacent cells, the variable you are most interested in should be specified last, and the early variables should be thought of as conditioning variables. An additional constraint is that displays have limited width but essentially infinite length, so variables with many levels may need to be specified as row variables.

4.2. High-dimensional arrays

You can use more than one ~ to create structures with more than two dimensions. For example, a cast formula of x ~ y ~ z will create a 3D array with x, y, and z dimensions. You can also still use multiple variables in each dimension: x + a ~ y + b ~ z + c. The following example shows the resulting dimensionality of various casting formulas. I do not show the actual output here because it is too large. You may want to verify the results for yourself:

```
R> dim(cast(ffm, time ~ variable ~ treatment, mean))
[1] 10  5  3

R> dim(cast(ffm, time ~ variable ~ treatment + rep, mean))
[1] 10  5  6

R> dim(cast(ffm, time ~ variable ~ treatment ~ rep, mean))
[1] 10  5  3  2

R> dim(cast(ffm, time ~ variable ~ subject ~ treatment ~ rep))
[1] 10  5 12  3  2

R> dim(cast(ffm, time ~ variable ~ subject ~ treatment ~ result_variable,
+    range))
[1] 10  5 12  3  2
```

The high-dimensional array form is useful for sweeping out margins with `sweep`, or modifying with `iapply` (Section 5, page 15).

The ~ operator is a type of crossing operator, as all combinations of the variables will appear in the output table. Compare this to the + operator, where only combinations that appear in the data will appear in the output. For this reason, increasing the dimensionality of the

output, i.e., using more ~s, will generally increase the number of structural missings. This is illustrated below:

```
R> sum(is.na(cast(ffm, ... ~ .)))
[1] 0

R> sum(is.na(cast(ffm, ... ~ rep)))
[1] 9

R> sum(is.na(cast(ffm, ... ~ subject)))
[1] 129

R> sum(is.na(cast(ffm, ... ~ time ~ subject ~ variable ~ rep)))
[1] 129
```

Margins of high-dimensional arrays are currently unsupported.

4.3. Lists

You can also use `cast` to produce lists. This is done with the | operator. Using multiple variables after | will create multiple levels of nesting.

```
R> cast(ffm, treatment ~ rep | variable, mean)
$potato
  treatment        1        2
1         1 6.772414 7.003448
2         2 7.158621 6.844828
3         3 6.937391 6.998276


$buttery
  treatment        1        2
1         1 1.797391 1.762931
2         2 1.989474 1.958621
3         3 1.805217 1.631034


$grassy
  treatment         1         2
1         1 0.4456897 0.8525862
2         2 0.6905172 0.6353448
3         3 0.5895652 0.7706897


$rancid
  treatment        1        2
1         1 4.283621 3.847414
2         2 3.712069 3.537069
3         3 3.752174 3.980172
```

```
$painty
  treatment         1        2
1          1 2.727586 2.439655
2          2 2.315517 2.597391
3          3 2.038261 3.008621
```

Space considerations necessitate only printing summaries of the following lists, but you can see `?cast` for full examples.

```
R> length(cast(ffm, treatment ~ rep | variable, mean))
[1] 5

R> length(cast(ffm, treatment ~ rep | subject, mean))
[1] 12

R> length(cast(ffm, treatment ~ rep | time, mean))
[1] 10

R> sapply(cast(ffm, treatment ~ rep | time + variable, mean), length)
 1  2  3  4  5  6  7  8  9 10
 5  5  5  5  5  5  5  5  5  5
```

This form is useful for input to `lapply` and `sapply`, and completes the discussion of the different types of output you can create with reshape. The remainder of the section discusses aggregation.

4.4. Aggregation

Whenever there are fewer cells in the cast form than there were in the original data format, an aggregation function is necessary. This formula reduces multiple cells into one, and is supplied in the `fun.aggregate` argument, which defaults (with a warning) to `length`. Aggregation is a very common and useful operation and the case studies section (Section 6, page 16) contains further examples of aggregation.

The aggregation function will be passed the vector of a values for one cell. It may take other arguments, passed in through `...` in `cast`. Here are a few examples:

```
R> cast(ffm, . ~ treatment)
  value    1    2    3
1 (all) 1159 1157 1155

R> cast(ffm, . ~ treatment, function(x) length(x))
  value    1    2    3
1 (all) 1159 1157 1155

R> cast(ffm, . ~ treatment, length)
  value    1    2    3
1 (all) 1159 1157 1155
```

```
R> cast(ffm, . ~ treatment, sum)
  value      1      2      3
1 (all) 3702.4 3640.4 3640.2

R> cast(ffm, . ~ treatment, mean)
  value        1        2        3
1 (all) 3.194478 3.146413 3.151688

R> cast(ffm, . ~ treatment, mean, trim = 0.1)
  value        1        2        3
1 (all) 2.595910 2.548112 2.589081
```

You can also display margins and use functions that return multiple results. See the next two sections for details.

4.5. Margins

It is often useful to be able to add statistics to the margins of your tables, for example, as suggested by Chatfield (1995). You can tell `cast` to display all margins with `margins = TRUE`, or list individual variables in a character vector, `margins = c("subject", "day")`. There are two special margins, `"grand_col"` and `"grand_row"`, which display margins for the overall columns and rows respectively. Margins are indicated with '(all)' as the value of the variable that was margined over.

These examples illustrate some of the possible ways to use margins. I've used sum as the aggregation function so that you can check the results yourself. Note that changing the order and position of the variables in the cast formula affects the margins that can be computed.

```
R> cast(ffm, treatment ~ ., sum, margins = TRUE)
  treatment   (all)
1         1  3702.4
2         2  3640.4
3         3  3640.2
4     (all) 10983.0

R> cast(ffm, treatment ~ ., sum, margins = "grand_row")
  treatment   (all)
1         1  3702.4
2         2  3640.4
3         3  3640.2
4     (all) 10983.0

R> cast(ffm, treatment ~ rep, sum, margins = TRUE)
  treatment      1      2   (all)
1         1 1857.3 1845.1  3702.4
2         2 1836.5 1803.9  3640.4
3         3 1739.1 1901.1  3640.2
```

```
4     (all) 5432.9 5550.1 10983.0

R> cast(ffm, treatment + rep ~ ., sum, margins = TRUE)
   treatment   rep   (all)
1          1     1  1857.3
2          1     2  1845.1
3          1 (all)  3702.4
4          2     1  1836.5
5          2     2  1803.9
6          2 (all)  3640.4
7          3     1  1739.1
8          3     2  1901.1
9          3 (all)  3640.2
10     (all) (all) 10983.0

R> cast(ffm, treatment + rep ~ time, sum, margins = TRUE)
   treatment   rep      1      2      3      4      5      6      7      8
1          1     1  156.4  212.8  206.0  181.2  207.5  181.5  156.4  185.1
2          1     2  216.4  195.1  193.6  153.7  204.5  184.8  157.7  216.4
3          1 (all)  372.8  407.9  399.6  334.9  412.0  366.3  314.1  401.5
4          2     1  186.6  212.7  171.8  192.7  157.3  183.4  175.0  173.4
5          2     2  168.3  157.3  185.8  187.1  172.8  214.6  172.0  188.6
6          2 (all)  354.9  370.0  357.6  379.8  330.1  398.0  347.0  362.0
7          3     1  189.2  211.9  172.3  190.2  150.6  160.9  165.2  150.4
8          3     2  216.7  180.5  199.2  191.8  183.0  191.8  218.3  174.6
9          3 (all)  405.9  392.4  371.5  382.0  333.6  352.7  383.5  325.0
10     (all) (all) 1133.6 1170.3 1128.7 1096.7 1075.7 1117.0 1044.6 1088.5
          9     10   (all)
1   176.2  194.2  1857.3
2   121.6  201.3  1845.1
3   297.8  395.5  3702.4
4   184.8  198.8  1836.5
5   145.2  212.2  1803.9
6   330.0  411.0  3640.4
7   173.3  175.1  1739.1
8   163.7  181.5  1901.1
9   337.0  356.6  3640.2
10  964.8 1163.1 10983.0

R> cast(ffm, treatment + rep ~ time, sum, margins = "treatment")
  treatment   rep     1     2     3     4     5     6     7     8     9    10
1         1     1 156.4 212.8 206.0 181.2 207.5 181.5 156.4 185.1 176.2 194.2
2         1     2 216.4 195.1 193.6 153.7 204.5 184.8 157.7 216.4 121.6 201.3
3         1 (all) 372.8 407.9 399.6 334.9 412.0 366.3 314.1 401.5 297.8 395.5
4         2     1 186.6 212.7 171.8 192.7 157.3 183.4 175.0 173.4 184.8 198.8
5         2     2 168.3 157.3 185.8 187.1 172.8 214.6 172.0 188.6 145.2 212.2
6         2 (all) 354.9 370.0 357.6 379.8 330.1 398.0 347.0 362.0 330.0 411.0
```

```
7            3      1 189.2 211.9 172.3 190.2 150.6 160.9 165.2 150.4 173.3 175.1
8            3      2 216.7 180.5 199.2 191.8 183.0 191.8 218.3 174.6 163.7 181.5
9            3  (all) 405.9 392.4 371.5 382.0 333.6 352.7 383.5 325.0 337.0 356.6

R> cast(ffm, rep + treatment ~ time, sum, margins = "rep")
  rep treatment      1      2      3      4      5      6      7      8      9     10
1   1            1 156.4 212.8 206.0 181.2 207.5 181.5 156.4 185.1 176.2 194.2
2   1            2 186.6 212.7 171.8 192.7 157.3 183.4 175.0 173.4 184.8 198.8
3   1            3 189.2 211.9 172.3 190.2 150.6 160.9 165.2 150.4 173.3 175.1
4   1        (all) 532.2 637.4 550.1 564.1 515.4 525.8 496.6 508.9 534.3 568.1
5   2            1 216.4 195.1 193.6 153.7 204.5 184.8 157.7 216.4 121.6 201.3
6   2            2 168.3 157.3 185.8 187.1 172.8 214.6 172.0 188.6 145.2 212.2
7   2            3 216.7 180.5 199.2 191.8 183.0 191.8 218.3 174.6 163.7 181.5
8   2        (all) 601.4 532.9 578.6 532.6 560.3 591.2 548.0 579.6 430.5 595.0
```

4.6. Returning multiple values

Occasionally it is useful to aggregate with a function that returns multiple values, e.g., `range` or `summary`. This can be thought of as combining multiple `cast`s each with an aggregation function that returns one variable. To display this we need to add an extra variable, `result_variable` that differentiates the multiple return values. By default, this new id variable will be shown as the last column variable, but you can specify the position manually by including `result_variable` in the casting formula.

```
R> cast(ffm, treatment ~ ., summary)
  treatment Min. X1st.Qu. Median  Mean X3rd.Qu. Max.
1         1    0        0    1.6 3.194      5.4 14.9
2         2    0        0    1.4 3.146      5.4 14.9
3         3    0        0    1.5 3.152      5.7 14.5

R> cast(ffm, treatment ~ ., quantile, c(0.05, 0.5, 0.95))
  treatment X5. X50. X95.
1         1   0  1.6 11.0
2         2   0  1.4 10.7
3         3   0  1.5 10.6

R> cast(ffm, treatment ~ rep, range)
  treatment 1_X1 1_X2 2_X1 2_X2
1         1    0 14.9    0 14.3
2         2    0 14.9    0 13.7
3         3    0 14.5    0 14.0
```

You can also supply a vector of functions:

```
R> cast(ffm, treatment ~ rep, c(min, max))
  treatment 1_min 1_max 2_min 2_max
```

```
1         1    0  14.9    0  14.3
2         2    0  14.9    0  13.7
3         3    0  14.5    0  14.0

R> cast(ffm, treatment ~ result_variable + rep, c(min, max))
  treatment min_1 min_2 max_1 max_2
1         1     0     0  14.9  14.3
2         2     0     0  14.9  13.7
3         3     0     0  14.5  14.0
```

5. Other convenience functions

There are many other problems encountered in practical analysis that can be painful to overcome without some handy functions. This section describes some of the functions that reshape provides to make dealing with data a little bit easier. More details are provided in the respective documentation.

5.1. Factors

- `combine_factor` combines levels in a factor. For example, if you have many small levels you can combine them together into an 'other' level.

- `reorder_factor` reorders a factor based on another variable. For example, you can order a factor by the average value of a variable for each level.

5.2. Data frames

- `rescaler` performs column-wise rescaling of data frames, with a variety of different scaling options including rank, common range and common variance. It automatically preserves non-numeric variables.

- `merge.all` merges multiple data frames together, an extension of `merge` in base R. It assumes that all columns with the same name should be equated.

- `rbind.fill` rbinds two data frames together, filling in any missing columns in the second data frame with missing values.

5.3. Miscellaneous

- `round_any` allows you to round a number to any degree of accuracy, e.g., to the nearest 1, 10, or any other number.

- `iapply` is an idempotent version of the `apply` function. It is idempotent in the sense that `iapply(x, a, function(x) x)` is guaranteed to return `x` for any value of a. This is useful when dealing with high-dimensional arrays as it will return the array in the same shape that you sent it. It also supports functions that return matrices or arrays in a sensible manner.

6. Case study: French fries

These data are from a sensory experiment investigating the effect of different frying oils on the taste of French fries over time. There are three different types of frying oils (treatment), each in two different fryers (rep), tested by 12 people (subject) on 10 different days (time). The sensory attributes recorded, in order of desirability, are potato, buttery, grassy, rancid, painty flavours. The first few rows of the data are shown in Table 1.

	time	treatment	subject	rep	potato	buttery	grassy	rancid	painty
61	1	1	3	1.00	2.90	0.00	0.00	0.00	5.50
25	1	1	3	2.00	14.00	0.00	0.00	1.10	0.00
62	1	1	10	1.00	11.00	6.40	0.00	0.00	0.00
26	1	1	10	2.00	9.90	5.90	2.90	2.20	0.00
63	1	1	15	1.00	1.20	0.10	0.00	1.10	5.10
27	1	1	15	2.00	8.80	3.00	3.60	1.50	2.30

Table 1: First few rows of the French fries dataset

We first melt the data to use in subsequent analyses.

```
R> ffm <- melt(french_fries, id = 1:4, na.rm = TRUE)
R> head(ffm)
  time treatment subject rep variable value
1    1         1       3   1   potato   2.9
2    1         1       3   2   potato  14.0
3    1         1      10   1   potato  11.0
4    1         1      10   2   potato   9.9
5    1         1      15   1   potato   1.2
6    1         1      15   2   potato   8.8
```

6.1. Investigating balance

One of the first things we might be interested in is how balanced this design is, and if there are many different missing values. We are interested in missingness, so we removed explicit missing values to put structural and non-structural missings on an equal footing.

We can investigate balance using length as our aggregation function:

```
R> cast(ffm, subject ~ time, function(x) 30 - length(x), margins = TRUE)
```

```
   subject   1    2    3    4    5    6    7    8    9   10  (all)
1        3  30   30   30   30   30   30   30   30   30   NA    270
2       10  30   30   30   30   30   30   30   30   30   30    300
3       15  30   30   30   30   25   30   30   30   30   30    295
4       16  30   30   30   30   30   30   30   29   30   30    299
5       19  30   30   30   30   30   30   30   30   30   30    300
6       31  30   30   30   30   30   30   30   30   NA   30    270
7       51  30   30   30   30   30   30   30   30   30   30    300
8       52  30   30   30   30   30   30   30   30   30   30    300
9       63  30   30   30   30   30   30   30   30   30   30    300
10      78  30   30   30   30   30   30   30   30   30   30    300
11      79  30   30   30   30   30   30   29   28   30   NA    267
12      86  30   30   30   30   30   30   30   30   NA   30    270
13   (all) 360  360  360  360  355  360  359  357  300  300   3471
```

Each subject should have had 30 observations at each time, so by displaying the difference we can more easily see where the data are missing.

```
R> cast(ffm, subject ~ time, function(x) 30 - length(x))
   subject 1 2 3 4 5 6 7 8  9 10
1        3 0 0 0 0 0 0 0 0  0 NA
2       10 0 0 0 0 0 0 0 0  0  0
3       15 0 0 0 0 5 0 0 0  0  0
4       16 0 0 0 0 0 0 0 1  0  0
5       19 0 0 0 0 0 0 0 0  0  0
6       31 0 0 0 0 0 0 0 0 NA  0
7       51 0 0 0 0 0 0 0 0  0  0
8       52 0 0 0 0 0 0 0 0  0  0
9       63 0 0 0 0 0 0 0 0  0  0
10      78 0 0 0 0 0 0 0 0  0  0
11      79 0 0 0 0 0 0 1 2  0 NA
12      86 0 0 0 0 0 0 0 0 NA  0
```

There are two types of missing observations here: a non-zero value, or a missing value. A missing value represents a subject with no records at a given time point; they did not turn up on that day. A non-zero value represents a subject who did turn up, but perhaps due to a recording error, missed some observations.

We can also easily see the range of values that each variable takes:

```
R> cast(ffm, variable ~ ., c(min, max))
  variable min  max
1   potato   0 14.9
2  buttery   0 11.2
3    grassy   0 11.1
4    rancid   0 14.9
5    painty   0 13.1
```

Better than just looking at the ranges is to look at the distribution, with a histogram. Here we use the molten data directly, facetting (aka conditioning or trellising) by the measured variable.

```
R> qplot(value, data = ffm, geom = "histogram", facets = . ~ variable,
+    binwidth = 1)
```
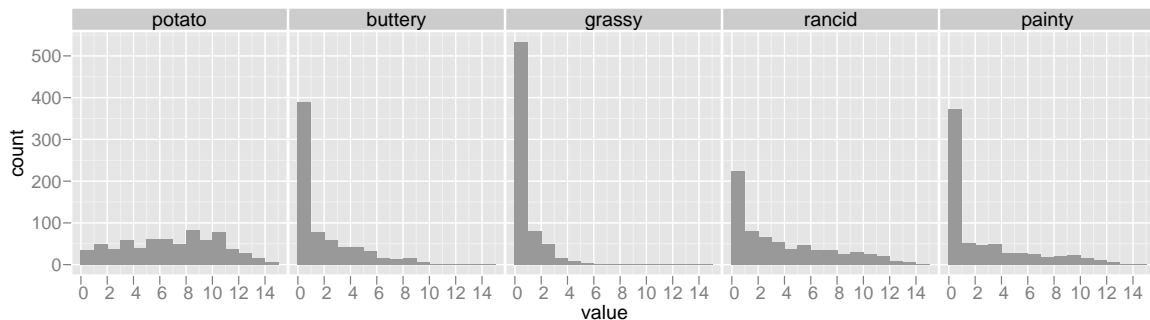

Figure 1: Histogram of French fries data.

6.2. Tables of means

When creating these tables, it is a good idea to restrict the number of digits displayed. You can do this globally, by setting `options(digits = 2)`, or locally, by using `round_any`.

Since the data are fairly well balanced, we can do some (crude) investigation as to the effects of the different treatments. For example, we can calculate the overall means for each sensory attribute for each treatment:

```
R> options(digits = 2)
R> cast(ffm, treatment ~ variable, mean, margins = c("grand_col",
+    "grand_row"))
  treatment potato buttery grassy rancid painty (all)
1         1    6.9     1.8   0.65    4.1    2.6   3.2
2         2    7.0     2.0   0.66    3.6    2.5   3.1
3         3    7.0     1.7   0.68    3.9    2.5   3.2
4     (all)    7.0     1.8   0.66    3.9    2.5   3.2
```

It does not look like there is any effect of treatment. This could be confirmed using a more formal analysis of variance.

6.3. Investigating inter-rep reliability

Since we have a repetition over treatments, we might be interested in how reliable each subject is: are the scores for the two repetitions highly correlated? We can explore this graphically by reshaping the data and plotting the data. Our graphical tools work best when the things
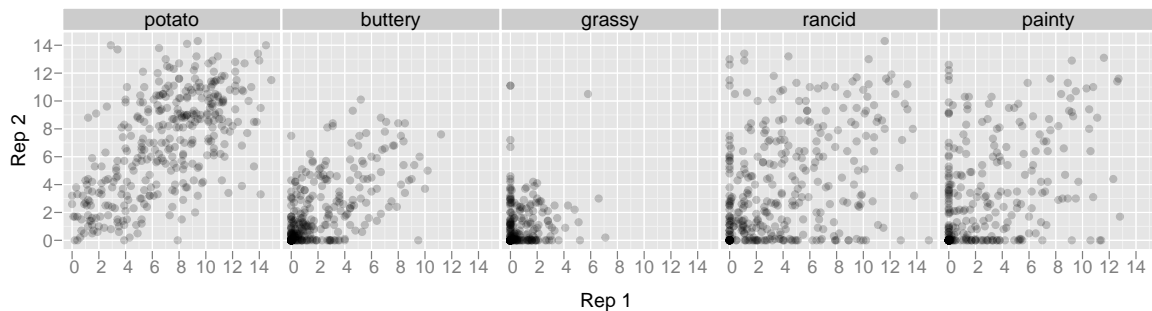
Figure 2: Youden plot of French fries data.

we want to compare are in different columns, so we'll cast the data to have a column for each rep. In quality control literature, this is known as a Youden plot (Youden 1959).

The correlation looks strong for potatoey and buttery, and poor for rancid and painty. Grassy has many low values, particularly zeros, as seen in the histogram. This reflects the training the participants received for the trial: they were trained to taste potato and buttery flavours, their grassy reference flavour was parsley (very grassy compared to French fries), and they were not trained on rancid or painty flavours.

If we wanted to explore the relationships between subjects or times or treatments we could follow similar steps.

7. Where to go next

You can find a quick reference and more examples in `?melt` and `?cast`. You can find some additional information on the reshape website `http://had.co.nz/reshape/`, including copies of presentations and papers related to reshape.

I would like to include more case studies of reshape in use. If you have an interesting example, or there is something you are struggling with please let me know: `h.wickham@gmail.com`.

Acknowledgments

References

Chatfield C (1995). *Problem Solving: A Statistician's Guide.* Chapman & Hall.

R Development Core Team (2007). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Youden W (1959). "Graphical Diagnosis of Interlaboratory Test Results." *Industrial Quality Control*, **15**, 24–28.

Affiliation:

Hadley Wickham
Iowa State University
Ames, Iowa 50011, United States of America
E-mail: h.wickham@gmail.com
URL: http://had.co.nz/