More data manipulation

December 11, 2017

R packs a lot of functionality to make data manipulation easier, and it is impossible to even give an overview of the array of tools that are available in this little space. However, the majority of the things I end up doing fall in three categories: apply some function to a grouping of the data, merging separate datasets, or reshaping data. Moreover, I think they summarize nicely they way R works.

Split-apply-combine

Consider again our tobacco example and the problem that we had about calculating group means. We used a specific function, `ave` that did precisely that. But the operation is actually a lot more general. We can think about spliting data structures by gropus, performing operations in each group and then grouping things again. Base R provides functions to accomplish this kind of tasks, but I personally prefer the functions in the `plyr` package, because they provide a common, clear, and intuitive interface.

The usage is very similar across functions. The first argument is the input data (such as a `data.frame` or a `list`). Then, a variable that tells how the input should be split into groups, although it may not be needed –think about a `list`. Finally, a description of the operation to be applied to each group. The logic will become clearer with a couple of examples. Let's start by realizing that `plyr` is not shipped with R. After installing it, we can load it and also read in some data.

```
library(plyr)
tobacco <- read.csv("http://koaning.io/theme/data/cigarette.csv")
```

Let's calculate the average income per state (across years). The name of the function gives a lot information: the first letter `d` says that the input will be a `data.frame`; the second letter, that the output will be another data.frame. All functions in `plyr` use the same input-output structure:

```
group_means <- ddply(tobacco, ~ state, mutate, avincome=mean(income))
head(group_means)
```

```
##   state year   cpi     pop   packpc   income  tax   avgprs     taxs
## 1    AL 1985 1.076 3973000 116.4863 46014968 32.5 102.1817 33.34834
## 2    AL 1986 1.096 3992000 117.1593 48703940 32.5 107.9892 33.40584
## 3    AL 1987 1.136 4016000 115.8367 51846312 32.5 113.5273 33.46067
## 4    AL 1988 1.183 4024000 115.2584 55698852 32.5 120.0334 33.52509
## 5    AL 1989 1.240 4030000 109.2060 60044480 32.5 133.2560 33.65600
## 6    AL 1990 1.307 4048508 111.7449 64094948 32.5 143.4486 33.75692
##   avincome
## 1 64137227
## 2 64137227
## 3 64137227
## 4 64137227
## 5 64137227
## 6 64137227
```

See how the grouping is passed as a formula, and now we are giving a name `avincome` to the newly created variable, which is the mean of the `income` variable. The `mutate` argument indicates that we want to perserve the number of rows in the dataset. Compare the output with what happens when we use `summarize` which says that we only one one value per group:

```
group_means <- ddply(tobacco, ~ state, summarize, avincome=mean(income))
head(group_means)
```

```
##   state  avincome
## 1    AL  64137227
## 2    AR  35031507
## 3    AZ  64086970
## 4    CA 623427174
## 5    CO  67137756
## 6    CT  84549187
```

Let's explore a bit more complex example that calculates a separate regression between `packpc` and `log(tax)` for each state and then pulls all the coefficients together.

```
lm_models <- dlply(tobacco, ~ state, function(x) lm(packpc ~ log(tax), data=x))
lm_coefs <- ldply(lm_models, coefficients)
head(lm_coefs)
```

```
##   state (Intercept)  log(tax)
## 1    AL    304.3076 -54.71754
## 2    AR    282.3870 -43.80102
## 3    AZ    242.2289 -43.11060
## 4    CA    226.0603 -40.22890
## 5    CO    397.4823 -84.07108
## 6    CT    302.4185 -52.55956
```

Several things are worth noting. First, that we are first transforming a `data.frame` into a `list` (`dlply`), and then a `list` into a `data.frame` (`ldply`). Can you see why? Second, that the function in our `dlply` called is passed as a anonymous function which takes the data for each country as argument. This is a very common strategy in R: we need a temporary function for a one-off task and we don't even need a name for it. In this case, we just want a function that applies the same model to different datasets. Finally, that `ldply` only needs two arguments: because `lm_models` is already a list, we don't need to do any splitting.

The approach above, using '`**ply` functions is very general but at the cost of being a bit awkward to read and slightly slow. In most cases, we only need to work with `data.frame` and therefore we can make a lot more assumptions on the data. Also, because the structure resembles a table we can then use a similar logic that underneaths SQL: we can develop a few verbs that describe operations and concatenate them one after the other. This approach has become very popular in the last few years and the `dplyr` package has been at the centerfold of the `tidyverse` approach to `R`. The `tidyverse` is not something we can cover, but it is probably useful to see a simple example. For instance, consider the operation above, we took a dataset, we *grouped* it by the `state` variable and we *summarized* each group to create a new variable income. We could express this chain of operations as:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:plyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
tobacco %>%
    group_by(state) %>%
    summarize(avincome=mean(income))
```

```
## # A tibble: 48 x 2
##     state  avincome
##    <fctr>     <dbl>
##  1     AL  64137227
##  2     AR  35031507
##  3     AZ  64086970
##  4     CA 623427174
##  5     CO  67137756
##  6     CT  84549187
##  7     DE  14187419
##  8     FL 249924961
##  9     GA 116220735
## 10     IA  48131054
## # ... with 38 more rows
```

The only strange element is the pipe operator `%>%` which, in essence, passes the output of the LHS as input to the RHS. With that we allow the code to be read from left to right. Compare it to the more standard way of writing the same code:

```r
summarize(group_by(tobacco, state), avincome=mean(income))
```

```
## # A tibble: 48 x 2
##     state  avincome
##    <fctr>     <dbl>
##  1     AL  64137227
##  2     AR  35031507
##  3     AZ  64086970
##  4     CA 623427174
##  5     CO  67137756
##  6     CT  84549187
##  7     DE  14187419
##  8     FL 249924961
##  9     GA 116220735
## 10     IA  48131054
## # ... with 38 more rows
```

`dplyr` is much faster and readable and it has gained a lot of popularity. It is definitely something worth exploring.

Merging two datasets

From the moment a `data.frame` is an object, we can hold as many as our computer allows. And we can put them together and merge them by specifying the keys that they have in common. Consider a trivial example in which we want to take our original dataset and merge into it the vector of means by state that we calculated above.[1]

```r
merged_tobacco <- merge(tobacco, group_means, by="state")
head(merged_tobacco)
```

```
##   state year   cpi     pop   packpc   income  tax   avgprs     taxs
```

[1] Yes, we trying to accomplish the same thing we did in the `mutate` call.

```
## 1    AL 1985 1.076 3973000 116.4863 46014968 32.5 102.1817 33.34834
## 2    AL 1992 1.403 4139269 106.9029 72281824 36.5 176.1137 38.08034
## 3    AL 1991 1.362 4091025 107.0147 67649568 34.5 161.7212 35.93784
## 4    AL 1987 1.136 4016000 115.8367 51846312 32.5 113.5273 33.46067
## 5    AL 1986 1.096 3992000 117.1593 48703940 32.5 107.9892 33.40584
## 6    AL 1989 1.240 4030000 109.2060 60044480 32.5 133.2560 33.65600
##   avincome
## 1 64137227
## 2 64137227
## 3 64137227
## 4 64137227
## 5 64137227
## 6 64137227
```

The `merge` function takes some other arguments to specify different classes of joins, what to do with the units that don't match, or what to do with duplicated variables.

Reshaping a dataset

We can change the structure and reshape our dataset so that rather than having state-year observations, each row represents data across years. Again, there is a very direct way to do it with base R, but I like the way the `reshape2` package works. Operations are split into two separate functions. The first one "melts" the dataset according to some indexing variables. The second one "casts" the data into a particular shape using a formula interface. The functions are very aptly named `melt` and `dcast`:

```r
library(reshape2)
molten_tobacco <- melt(tobacco[, c("state", "year", "tax")], id=c("state", "year"))
head(dcast(molten_tobacco, state ~ variable + year))
```

```
##   state tax_1985 tax_1986 tax_1987 tax_1988 tax_1989 tax_1990 tax_1991
## 1    AL     32.5     32.5     32.5     32.5     32.5     32.5    34.50
## 2    AR     37.0     37.0     37.0     37.0     37.0     37.0    39.00
## 3    AZ     31.0     31.0     31.0     31.0     31.0     31.0    35.25
## 4    CA     26.0     26.0     26.0     26.0     38.5     51.0    53.00
## 5    CO     31.0     31.0     36.0     36.0     36.0     36.0    38.00
## 6    CT     42.0     42.0     42.0     42.0     45.5     56.0    58.00
##   tax_1992 tax_1993 tax_1994 tax_1995
## 1    36.50 38.50000     40.5 40.50000
## 2    42.00 49.20834     55.5 55.50000
## 3    38.00 40.00000     42.0 65.33333
## 4    55.00 57.00000     60.0 61.00000
## 5    40.00 42.00000     44.0 44.00000
## 6    63.75 67.00000     71.0 74.00000
```

We don't always need to perform the two steps and if you look at the output of `molten_tobacco` you will realize that it is not doing much for us other than adding a factor that holds the variable *not* indicated in the `id` argument, and putting its values in a separate column.[2]

[2] We could have omitted the first step and run `dcast(tobacco[, c("state", "year", "tax")], state ~ year)`.