plyr: divide and conquer

Hadley Wickham

May 4, 2008

plyr is a set of tools that solves a common set of problems: you need to break a big problem down into manageable pieces, operate on each pieces and then put all the pieces back together. This paper describes the components that make up plyr. Includes a case study.

1 Introduction

The plyr package provides tool for solving a common class of problems, where you break apart a big data structure, operate on each piece independently and then put all the pieces back together (possible in a different format to the original). This paper introduces the `ply` family of tools which generalise the `apply` family to all combinations of input and output data structures for lists, data frames and arrays (including vectors and matrices).

In general, these tools provide a replacement for `for` loops for a large set of problems that arise in practice. The major assumption that they make is that each piece can be operated on independently, so if there is any dependence (e.g. recursive relationship) between the pieces then these tools are not appropriate. These tools should be use to more clearly express the intent of your algorithm, rather than to speed up computation.

There are a number of other resources that also attempt to simplify this class of problems:

- Base R: apply functions, by, etc. The equivalence between these function and plyr functions are described in Section 5.
- The `doBy` package
- The `gdata` package
- The `scope` package
- Data manipulation in R, by Phil Spector
- Chapters in MASS, R intro?

The basic set of plyr functions are listed in Table 1. They are named according to the type of input they process and the type of output that they produce. The input type determines how input can be broken up, described in detail in Section 2. The output type determines how the pieces

to \ from	array	data.frame	list
array	aaply	daply	laply
data.frame	adply	ddply	ldply
list	alply	dlply	llply
nothing	a_ply	d_ply	l_ply

Table 1: The 12 key functions that make up `plyr`.

are joined back together again, described in detail in Section 3. The `plyr` package also provides a number of helper functions for error recovery, splatting, column-wise processing, and reporting progress. These are described in Section 4.

Note that through this paper we will use array to refer to vectors (1d arrays) and matrices (2d arrays) as well.

Arguments to the ply functions are determined by the types of input and outut. For this reason, it's useful to refer to a complete row or column of Table 1. The notation we use for this is `d*ply` to refer an entire row (fixed input) and `*dply` for an entire column (fixed output).

2 Input

For all `ply` functions the first argument is the data to divide and conquer. The second argument describes how to split up the data, and is different for each input type. Arrays are split up by their dimensions, while data frames are split into groups based on combinations of variables. Lists are assumed to be broken up already. The splitting for data frames and arrays is described in more detail below.

2.1 Data frames (d)

When operating on a data frame, you usually want to split it up into groups based on combinations variables in the data set. For `d*ply` you specify which variables (or functions of variables) to use. These variables are specified in a special way to highlight that they are computed first from the data frame, then the global environment (in which case it's your respsonsibilty to ensure that their length is equal to the number of rows in the data frame).

- Functions of variables: `.(round(a))`, `.(a * b)`
- Variables in the global environment `.(anothervar)`

Split by variables. .() notation

2.2 Arrays (a)

The `margins` argument of `a*ply` describes how to slice up the array in the same way that `apply` does. For example, `margins = 1` specifies that we want to break up the array by rows (the first index when subsetting), and `margins = 2` by columns (the second index when subsetting). You can also use combinations of margins. For example, `margins = 1:2` will split up by the first two dimensions. For a 3d array, this will produce the columns in the z-direction.

A special case of operating on arrays corresponds to the `mapply` function of base R. The plyr equivalents are named `maply`, `mdply`, `mlply` and `m_ply`. These default to working on the first dimension (i.e. row-wise) and automatically splat the function so that function is called not with a single list as input, but each column is passed as a separate argument to the function. Compared to using `mapply`, for the `m*ply` functions you will need to `cbind` the columns together first. This will ensure that each argument has the same length, and allows the `m*ply` functions to have the same argument order as all the other

3 Output

Avoid any ambiguity about what you'll get back from one of these functions.

This strictness should enable certain optimisations that are unavailable to apply and family.

3.1 Data frames (d)

Function needs to return atomic vector of fixed size or data.frame.

Extra variable will be added according to splits.

3.2 Arrays (a)

Function needs to return atomic of vector/matrix/array of fixed size/shape, or a list.

Each splitting criterion creates a new dimension. Dimensions from function are added onto the end. This ensures idempotency

3.3 Lists (l)

Not much to talk about. dl -¿ ld will be labelled the same as dd.

3.4 Ignored (_)

Sometimes you are operating on a list purely for the side effects (e.g. plots, caching, output to screen/file). A little more memory efficient than simply abandoning the output of *lply because it doesn't construct the intermediate storage.

4 Helpers

conditions on function

how extra arguments are passed in

explode/splat

each colwise failwith

progress bars

5 Equivalence to existing R functions

- lapply → llply
- sapply → laply, llply
- apply → aaply, alply
- mapply → maply, mlply
- by → dlply
- aggregate → daply(, colwise(f))
- tapply, ave, sweep → no direct correspondence
- lapply(split(df, ...), f) → dlply
- do.call("rbind", lapply(split(df, ...), f)) → ddply
- HMisc functions

```
* aggregate(mtcars, list(mtcars$cyl), median)
  daply(mtcars, .(cyl), colwise(median))
  daply(mtcars, .(cyl), colwise(median, .if = is.numeric))

* p <- function(df) coef(lm(mpg ~ wt, data = df))
  do.call("rbind", lapply(split(mtcars, mtcars$cyl), p))
  ddply(mtcars, .(cyl), p)
```

The cast function in the reshape package (Wickham, 2005) is a special case of aaply, which provides a number of nice labelling features.

6 Strategy

Take a small dataset, that you can easily solve.

6.1 Case study: baseball data

Calculating stint.

Model for each player

7 Conclusion

If slow, might want to look at the profr package to speed up.

However, it is my aim to eventually implement these functions in C for maximum speed and memory efficiency, so that they are competitive with the built in operations. I also plan to investigate a connection to the `papply` function to allow for easy parallelisation across multiple instances of R (particularly for multi-core machines).

References

H. Wickham. *reshape: Flexibly reshape data.*, 2005. URL `http://had.co.nz/reshape/`. R package version 0.7.1.