plyr: divide and conquer

Hadley Wickham

May 9, 2008

1 Introduction

The plyr package provides tools for solving a common class of problems, where you break apart a big complicated data structure into small simple pieces, operate on each piece independently and then put all the pieces back together (possibly in a different format to the original). This paper introduces the ply family of functions which generalise the apply family, and include all combinations of input from and output to lists, data frames and arrays.

This paper describes version 0.1 of plyr, which requires R 2.7.0 or later and has no run-time dependencies. To install it from within R, run install.packages("plyr"). Information about the latest version of the package, including updated version of this document, can be found online at http://had.co.nz/plyr. If you have any questions or comments about plyr, please feel free to email me directly at h.wickham@gmail.com.

In general, these tools provide a replacement for for loops for a large set of practical problems. 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. It is not true that for loops are slow, but often they do not clearly express the intent of your algorithm, as you need a lot of extra housekeeping code. The tools of plyr aim to eliminate this extra code and illuminate the key components of your computations.

Section 2 introduces the plyr family of tools and how to use them. 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 3. Section 4 discusses the general strategy that these functions support, including cases studies that explore long term professional baseball players and ozone measured over space and time. Finally, Section 5 maps existing R functions to their plyr counterparts and lists related packages, and Section 6 describes future plans.

2 Usage

Table 1 lists the basic set of plyr functions. Each function is named according to the type of input it processes and the type of output it produces. The type of the input determines how it can be broken up, and the various possibilities described in detail in Section 2.1. The output type determines how the pieces are joined back together again, as described in detail in Section 2.2. Note that throughout this paper we will use array to refer to vectors (1d arrays) and matrices (2d arrays) as well.

from to	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`. Arrays include matrices and vectors as special cases.

Arguments to the ply functions are determined by the types of input and output. 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).

Plyr functions have either two or three main arguments, depending on the type of input:

- `a*ply(data., margins., .f, ..., progress.  = "none")`
- `d*ply(data., variables., .f, ..., progress.  = "none")`
- `l*ply(data., f., ..., progress.  = "none")`

The first argument is the `data` to be split up, processed and recombined. For arrays and data frames, the second argument describes how to split up the data by `margins` or by `variables`. The third argument is the function to be applied to each piece. All further arguments are passed on to the processing function. The `.progress` argument controls displaying of a progress bar, and is described at the end of Section 3.

Note that all `plyr` arguments are suffixed with "`.`" - this is to help prevent clashes with arguments in the processing function. Recommended practice is to call `**ply` functions with arguments specified by position, not name.

2.1 Input

- Arrays are sliced
- Data frames are split into groups based on combinations of variables
- Lists are assumed to be broken up already, and so this argument is omitted

2.1.1 In: array (`a*ply`)

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

Can pass data frame if want to treat as 2d structure.

2.1.2 In: data frame (`d*ply`)

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 responsibility to ensure that their length is equal to the number of rows in the data frame).

- The interaction of multiple variables are taken: `.(a, b, c)`
- Functions of variables: `.(round(a))`, `.(a * b)`
- Variables in the global environment `.(anothervar)`

2.2 Output

The output type defines how the pieces will be joined back together again, and how they will be labelled. The labels are particularly important to allow you to match up the input with the output.

The input and output types are the same, except there is an additional output option, which discard the output. This is useful for functions with side effects that make changes outside of R

The output type also places some restrictions on what type of results the processing function should return. Generally, the processing function should return the same type of data as the eventual output, (i.e. vectors, matrices and arrays for `*aply` and data frames for `*dply`) but some other formats are accepted for convenience and are described below.

2.2.1 Out: array (`*aply`)

With array output the dimensionality is determined by the input splits. A list will produce a single dimension, a data frame will have a dimension for each variable split on, and a array will have a dimension for each dimension that it was split on. The processing function should return an atomic (i.e. `is.atomic(x) == TRUE`) of array of fixed size/shape, or a list. If atomic, the extra dimensions will added perpendicular to the original dimensions. If a list, the output will be a list with dimensions.

Some examples should make this easier to understand:

The dimnames of the array will be the same as the input, if an array, or the extracted from the subsets if a data frame.

If there are no results, `adply` will return a logical vector of length 0.

2.2.2 Out: data frames (`*dply`)

The processing functions should either return a data.frame, or a (named) atomic vector of fixed length, which will form the columns of the output.

If there are no results, `*dply` will return an empty data frame.

The output data frame will be supplemented with columns that identify the subset of the original dataset that each piece was computed from. These columns make it easier to merge the old and new data. If the input was a data frame, this will be the values of the splitting variables. If the input was an array, this will be the dimension names.

2.2.3 Out: list (`*lply`)

This is the simplest output format, where each processed piece is joined together in a list. The list also stores the labels associated with each pieces, so that if you use `ldply` or `laply` to further process the list the labels will appear as if you had used `aaply`, `adply`, `daply` or `ddply` directly. `llply` is convenient for calculating complex objects once (e.g. models), from which pieces of interest are later extracted into arrays and data frames.

There are no restrictions on the output of the processing function. If there are no results, `*lply` will return a list of length 0.

2.3 Out: nothing (`*_ply`)

Sometimes you are operating on a list purely for the side effects (e.g. plots, caching, output to screen/file). This is a little more efficient than abandoning the output of `*lply` because it doesn't store the intermediate results.

3 Helpers

The `plyr` package also provides a number of helper function which take a function (or functions) as input and return a modified function as output.

- `splat` converts a function to use. This is useful when you want to pass a function a row of data frame or array, and don't want to manually pull it apart in your function. For example:

  ```
  hp_per_cyl <- function(hp, cyl, ...) hp / cyl
  splat(hp_per_cyl)(mtcars[1,])
  splat(hp_per_cyl)(mtcars)
  ```

  Generally, splatted functions should have `...` as an argument, so you only need to specify the variables that you are interested in. For more information on how splat works, see `do.call`.

  `splat` is applied to functions used in `m*ply` by default.

- `each` takes a list of functions and produces a function that runs each function on the inputs and returns a named vector of outputs. For example, `each(min, max)` is short hand for function(x) c(min = min(x), max = max(x)). Using each with a single function is useful if you want a named vector as output.

- colwise converts a function that works on vectors, to one that operates column-wise of data frame, returning a data fram. For example, colwise(median) is a function that computes the median of each column of a data.frame.

  The optional .if argument specialises the function to only run on certain types of vector, e.g. .if = is.factor or .if = is.numeric. These two restrictions are provided in the premade calcolwise and numcolwise.

- failwith sets a default value to return if the function throws an error. For example, failwith(NA, f) will return an NA whenever f throws an error.

  The optional quiet argument suppresses any notice of the error when it is TRUE.

- e2f converts an expression to a function. This allows you to imitate replicate.

Each plyr function also has a .progress argument which allows you to monitor the progress of long running operations. There are four difference progress bars:

- "none", the default. No progress bar is displayed.
- "text" provides a textual progress bar which.
- "win" and "tk" provide graphical progress bars for Windows and systems with the tcl/tk package loaded.

The progress bars assume that processing each piece takes the same amount of time, so will not be 100% accurate.

4 Strategy

1. Extract a subset of the data for which it is easy to solve the problem
2. Solve the problem by hand, checking as you go
3. Write a function that encapsulates the solution
4. Use the appropriate ply function to split up the original data, apply the function and join the pieces back together.

The following two case studies illustrate these techniques for a range of problems related to a data frame storing the batting records for long-term baseball players, and a 3d array representing space and time values of ozone.

4.1 Case study: baseball data

The baseball data set contains the batting records for all professional US players with 15 or more years of data. The complete list of variables are described fully ?baseball, but for this example we will focus on just four: id, which identifies the player, year the year of the record and rbi the number of runs that the player made in the season, and at bat, the number of times the player had an opportunity to hit the ball.

(This is a rather crude analysis, as it doesn't take into account the people that might already be on the other plates)

What we'll explore is the performance of a batter over his career. To get started, we need to calculate the careeryear, i.e. the number of years since the player started playing. This is easy to do if we have a single player:

```
baberuth <- subset(baseball, id == "ruthba01")
baberuth$cyear <- baberuth$year - min(baberuth$year) + 1
```

To do this for all players, we first make a function:

```
calculate_cyear <- function(df) {
  transform(df,
    cyear = year - min(year),
    cpercent = (year - min(year)) / (max(year) - min(year))
  )
}
```

and then split up the whole data frame into people, run the function on each piece and join them back together into a data frame:

```
baseball <- ddply(baseball, .(id), calculate_cyear)
baseball <- subset(baseball, ab >= 25)
```

To summarise the pattern across all players, we first need to figure out what the common patterns are. A time series plot of rbi/ab, runs per bat, is a good place to start. We do this for Babe Ruth, as shown in Figure 1, then write a function to do it for any player (taking care to ensure common scale limits) and then use d_ply to save a plot for every player to a pdf. We use two tricks here: reorder to sort the players in order of average rbi / ab, and failwith to ensure that even if a single plot doesn't work we will still get output for the others.

```
qplot(cyear, rbi / ab, data=baberuth, geom="line")

xlim <- range(baseball$cyear, na.rm=TRUE)
ylim <- range(baseball$rbi / baseball$ab, na.rm=TRUE)
plotpattern <- function(df) {
  print(qplot(cyear, rbi / ab, data = df, geom="line", xlim = xlim, ylim = ylim ))
}
```

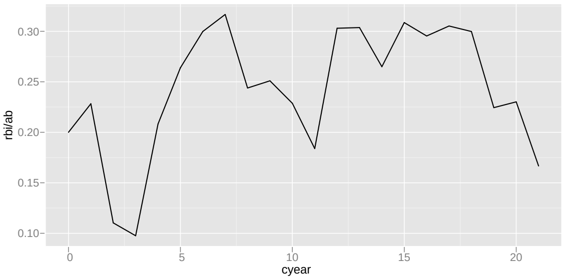

Figure 1: Runs per bat for Babe Ruth.

```
pdf("paths.pdf", width=8, height=4)
d_ply(baseball, .(reorder(id, rbi / ab)), failwith(NA, plotpattern))
dev.off()
```

Flicking through the 1145 plots reveals that there doesn't seem to be much of a common pattern, although many players do seem to have a roughly linear trend with quite a bit of noise. We'll start by fitting a linear model to each player and then exploring the results. This time we'll skip doing it by hand and go directly to the function.

```
model <- function(df) {
  lm(rbi / ab ˜ cyear, data=df)
}
model(baberuth)
models <- dlply(baseball, .(id), model)
```

Now we have a list of 1145 models, one for each player. To do something interesting with these, we need to extract some summary statistics. We'll extract the coefficients of the model (the slope and intercept), and a measure of model fit so we can ensure we're not drawing conclusions based on models that fit the data very poorly, the R-squared. The first few rows of `coef` are shown in Table 2.

```
rsq <- function(x) summary(x)$r.squared
coef <- ldply(models, function(x) c(coef(x), rsq(x)))
names(coef) <- c("id", "intercept", "slope", "rsquare")
```

Figure 2 displays the distribution of r-squared across the models. The models generally do a very bad job of fitting the data! Figure 3 summarises these bad models. These plots show a negative correlation between slope and intercept, and the particularly bad models have estimates for both values close to 0. Reassuringly, there are no players in the bottom left quadrant with both negative slope and intercept.

This concludes the baseball player case study, which used used `ddply`, `d_ply`, `dlply` and `ldply`. Our statistical analysis was not very sophisticated, but the tools of `plyr` made it very easy to work at the player level, and then combine results into a single summary.

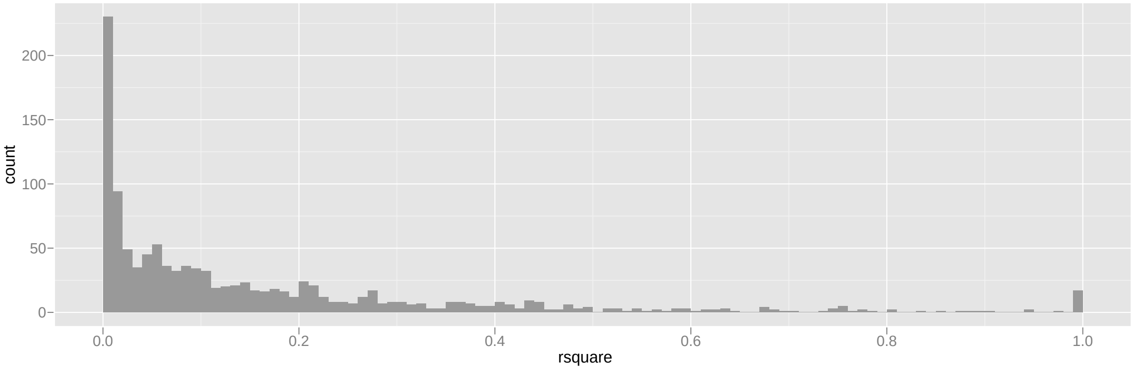

Figure 2: Histogram of model r-squared with bin width of 0.05. Most models fit very poorly! The spike of models with a r-squared of 1 are players with only two data points, found by inspecting `ldply(models[coef$rsquare == 1], "[[", "model")`

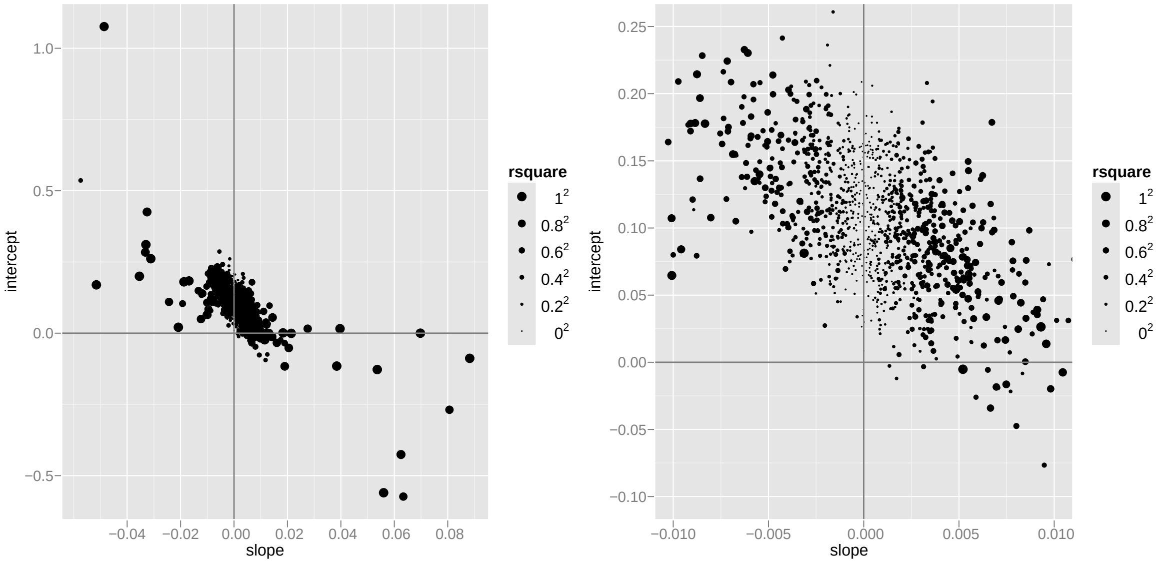

Figure 3: A scatterplot of model intercept and slope, with one point for each model (player). The size of the points is proportion to the R-square of the model. Vertical and horizontal lines emphasis the x and y origins.

id	intercept	slope	rsquare
aaronha01	0.18	0.00	0.00
abernte02	0.00		0.00
adairje01	0.09	−0.00	0.01
adamsba01	0.06	0.00	0.03
adamsbo03	0.09	−0.00	0.11
adcocjo01	0.15	0.00	0.23

Table 2: The first few rows of the coef data frame. Note that the player ids from the original data have been preserved

4.2 Case study: spatio-temporal ozone distribution

Standardisation/smoothing. (Hobbs et al., To appear)

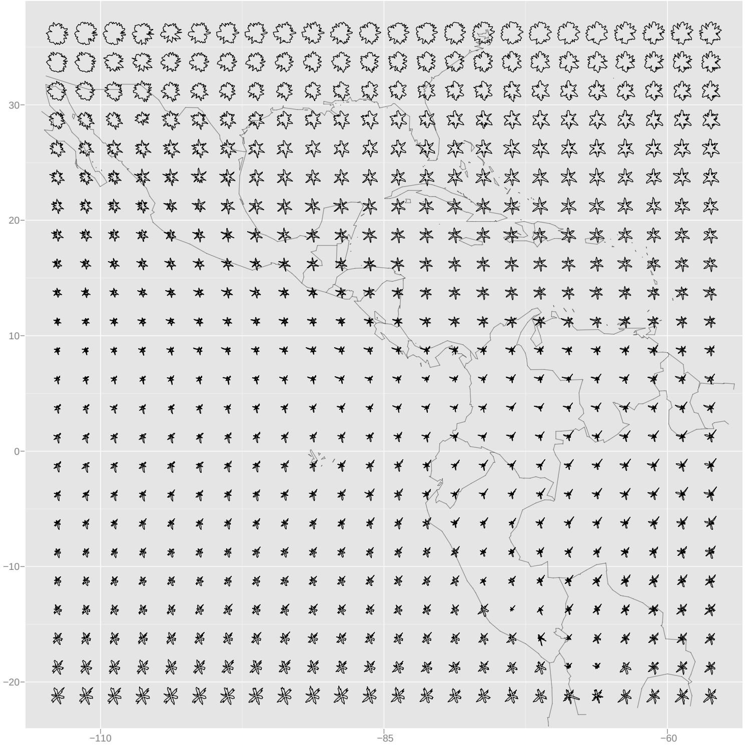

Figure 4: Star glyphs showing variation in ozone over time at each spatial location.

For many other types of operations, it is useful to convert this array structure to a data frame. The melt function in the reshape package is one way to do that which preserves the dimension labels as much as possible.

5 Equivalence to existing R functions

Table 3 describes the equivalent between functions in base R and the functions provided by `plyr`. The built in R functions focus mainly on arrays and lists, not data frames, and most provide an argument to determine whether an array or list should be returned. The syntax is also less consistent than plyr, for example, `mapply` takes a function as the first argument rather than the input data. Compared to `apply`, `aaply` returns the dimensions in a different order so as to be idempotent - i.e. `apply(x, a, function(x) x) == x` for all `a`.

Avoid any ambiguity about what you'll get back from one of these functions. This replaces the `simplify` argument that many of the `apply` functions in base R has, and means that you can depend on the output of each function being a given type (which makes programming with the results easier).

base	from	to	plyr
apply	a	a	aaply
lapply	l	l	llply
sapply	l	a	laply
mapply	a	a/l	maply / mlply
by	d	l	dlply
aggregate	d	d	ddply + colwise

Table 3: Mapping between apply functions and plyr functions.

Related functions `tapply`, `ave` and `sweep` have no corresponding function in `plyr`, and still remain useful. `merge` is also for combining summaries with the original data. The cast function in the reshape package (Wickham, 2005) is closely related to `aaply`.

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

- The `doBy` package
- The `gdata` package
- The `scope` package
- Data manipulation in R, by Phil Spector
- Chapters in MASS, R intro?

6 Future plans

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).

multir

References

J. Hobbs, H. Wickham, H. Hofmann, and D. Cook. Glaciers melt as mountains warm: A graphical case study. *Computational Statistics*, To appear. Special issue for ASA Statistical Computing and Graphics Data Expo 2007.

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