REVOLUTION ANALYTICS

Data Manipulation with dplyr

Revolution Analytics

1 The Grammar of Data Manipulation

2 Filtering and Reordering Data

3 Transformations and Summaries

4 Chaining Verbs

5 Advanced Topics

Overview

At the end of this session, you will have learned:

- How to manipulate data quickly with `dplyr` using a very intuitive ‘grammar’
- How to use `dplyr` to perform common exploratory and manipulation procedures
- Apply your own custom functions to group manipulations `dplyr` with `mutate()`, `summarise()` and `do()`
- Connect to remote databases to work with larger than memory datasets

Outline

1 The Grammar of Data Manipulation

2 Filtering and Reordering Data

3 Transformations and Summaries

4 Chaining Verbs

5 Advanced Topics

Why use dplyr?

- R comes with a plethora of base functions for data manipulation
- `dplyr` makes data manipulation easier by providing a few functions for the most common tasks and procedures
- `dplyr` achieves remarkable speed-up gains by using a C++ backend

The dplyr grammar

- `dplyr` was inspired to give data manipulation a simple, cohesive grammar (similar philosophy to `ggplot` - grammar of graphics)

Important verbs

filter	select rows based on matching criteria
select	select columns by column names
arrange	reorder rows by column values
mutate	add new variables based on transformations of existing variables
summarise	reduce variables to smaller values based by groups (group could be entire dataset)

Newer Versions

Starting with v0.3 (October 2014), there are a host of newer verbs.

slice	select rows by number
transmute	transform and drop other variables
distinct	returns unique rows
count	helper verb for tabulating

Data Setup

```
library(dplyr)
dataPath <- "../data"
bankCSV <- file.path(dataPath, "bank-full.csv")
bankData <- read.csv(bankCSV, sep = ";")
```

Viewing Data

- `dplyr` includes a wrapper called `tbl_df` makes df into a 'local df' that improves the printing of dataframes in the console
- if you want to see more of the data you can still coerce to `data.frame`

```
(bankData <- tbl_df(bankData))
```

```
## Source: local data frame [45,211 x 17]
##
##    age           job  marital education default balance housing loan
## 1   58    management  married  tertiary      no    2143     yes   no
## 2   44    technician   single secondary      no      29     yes   no
## 3   33 entrepreneur  married secondary      no       2     yes  yes
## 4   47   blue-collar  married   unknown      no    1506     yes   no
...
```

Outline

1. The Grammar of Data Manipulation
2. Filtering and Reordering Data
3. Transformations and Summaries
4. Chaining Verbs
5. Advanced Topics

Subsetting Data

- `dplyr` makes subsetting by rows very easy
- The `filter` verb takes conditions for filtering rows based on conditions
- This works in a similar fashion to SAS's data step with a `where` clause:

Subsetting Example 1

```
filter(bankData, default == "yes")
```

```
## Source: local data frame [815 x 17]
##
##    age          job  marital education default balance housing loan
## 1   42 entrepreneur divorced  tertiary     yes       2     yes   no
## 2   55     services divorced secondary     yes       1     yes   no
## 3   51       admin.   single secondary     yes      -2      no   no
## 4   33   technician  married secondary     yes      72     yes   no
...
```

Subsetting Example 2

```
filter(bankData, balance < 1000)
```

```
## Source: local data frame [30,569 x 17]
##
##    age           job  marital education default balance housing loan
## 1   44    technician   single secondary      no      29     yes   no
## 2   33 entrepreneur  married secondary      no       2     yes  yes
## 3   33       unknown   single   unknown      no       1      no   no
## 4   35    management  married  tertiary      no     231     yes   no
...
```

Subsetting Example 3

```
filter(bankData, month %in% c("april", "may", "jun"), default == "yes")
```

```
## Source: local data frame [351 x 17]
##
##    age           job  marital education default balance housing loan
## 1   42 entrepreneur divorced  tertiary     yes       2     yes   no
## 2   55      services divorced secondary     yes       1     yes   no
## 3   51        admin.   single secondary     yes      -2      no   no
## 4   33    technician  married secondary     yes      72     yes   no
...
```

Exercise

Your turn:

- How many defaults in the dataset?
- How many of the entrepeneurs that defaulted were also divorced?

Numeric Indices

You can also extract particular rows by number using `slice()`.

```
slice(bankData, 5:10)
```

```
## Source: local data frame [6 x 17]
##
##   age          job  marital education default balance housing loan contact
## 1  33      unknown   single   unknown      no       1      no   no unknown
## 2  35   management  married  tertiary      no     231     yes   no unknown
## 3  28   management   single  tertiary      no     447     yes  yes unknown
## 4  42 entrepreneur divorced  tertiary     yes       2     yes   no unknown
...
```

Select a set of columns

- You can use the `select()` verb to specify which columns of a dataset you want
- This is similar to the `keep` option in SAS's data step.
- Use a colon `:` to select all the columns between two variables (inclusive)
- Use `contains` to take any columns containing a certain word/phrase/character

Select Example 1

```
select(bankData, age, job, default, balance, housing)
```

```
## Source: local data frame [45,211 x 5]
##
##    age           job default balance housing
## 1   58    management      no    2143     yes
## 2   44    technician      no      29     yes
## 3   33 entrepreneur      no       2     yes
## 4   47   blue-collar      no    1506     yes
...
```

Select Example 2

```
select(bankData, default:duration, contains("p"))
```

```
## Source: local data frame [45,211 x 12]
##
##    default balance housing loan contact day month duration campaign pdays
## 1       no    2143     yes   no unknown   5   may      261        1    -1
## 2       no      29     yes   no unknown   5   may      151        1    -1
## 3       no       2     yes  yes unknown   5   may       76        1    -1
## 4       no    1506     yes   no unknown   5   may       92        1    -1
...
```

Select: Other Options

starts_with(x, ignore.case = FALSE) name starts with `x`

ends_with(x, ignore.case = FALSE) name ends with `x`

matches(x, ignore.case = FALSE) selects all variables whose name matches the regular expression `x`

num_range("V", 1:5, width = 1) selects all variables (numerically) from `V1` to `V5`.

You can also use a − to drop variables.

Exercise

Try to use `select()` to

- Drop the first 5 variables in the bank data set.
- Also drop all variables that start with `p`

Hint: Use the $-$ to drop variables.

Renaming Variables

There is also a `rename` verb to easily rename variables. You can use `select` to select and to rename by using named arguments.

```
select(bankData, bought_option = y)
```

```
## Source: local data frame [45,211 x 1]
##
##    bought_option
## 1             no
## 2             no
## 3             no
## 4             no
...
```

Rename Example

```
rename(bankData, bought_option = y)
```

```
## Source: local data frame [45,211 x 17]
##
##    age           job  marital education default balance housing loan
## 1   58    management  married  tertiary      no    2143     yes   no
## 2   44    technician   single secondary      no      29     yes   no
## 3   33 entrepreneur  married secondary      no       2     yes  yes
## 4   47   blue-collar  married   unknown      no    1506     yes   no
...
```

Reordering Data

- You can reorder your dataset based on conditions using the `arrange()` verb

Arrange Example 1

```
arrange(bankData, job, default)
```

```
## Source: local data frame [45,211 x 17]
##
##    age    job  marital education default balance housing loan contact day
## 1   41 admin. divorced secondary      no     270     yes   no unknown   5
## 2   29 admin.   single secondary      no     390     yes   no unknown   5
## 3   45 admin.   single   unknown      no      13     yes   no unknown   5
## 4   44 admin.  married secondary      no    -372     yes   no unknown   5
...
```

Arrange Example 2

```
arrange(bankData, balance, default)
```

```
## Source: local data frame [45,211 x 17]
##
##    age           job  marital education default balance housing loan
## 1   26   blue-collar   single secondary     yes   -8019      no  yes
## 2   49    management  married  tertiary     yes   -6847      no  yes
## 3   60    management divorced  tertiary      no   -4057     yes   no
## 4   43    management  married  tertiary     yes   -3372     yes   no
...
```

Arrange Example 3

You can use `desc()` to sort in descending order.

```
arrange(bankData, desc(balance), default)
```

```
## Source: local data frame [45,211 x 17]
##
##    age          job  marital education default balance housing loan
## 1   51   management   single  tertiary      no  102127      no   no
## 2   59   management  married  tertiary      no   98417      no   no
## 3   84      retired  married secondary      no   81204      no   no
## 4   84      retired  married secondary      no   81204      no   no
...
```

Exercise

Use `arrange()` to sort on the basis of `y`, `marital`, `job` (in descending order), and `balance`

Summary

filter()	Extract subsets of rows. See also `slice()`
select()	Extract subsets of columns. See also `rename()`
arrange()	Sort your data

Questions?

Outline

1 The Grammar of Data Manipulation

2 Filtering and Reordering Data

3 Transformations and Summaries

4 Chaining Verbs

5 Advanced Topics

Transformations

- The `mutate()` verb can be used to make new columns

```
mutate(bankData, DefaultFlag = ifelse(default == "yes", 1, 0))
```

```
## Source: local data frame [45,211 x 18]
##
##    age           job  marital education default balance housing loan
## 1   58    management  married  tertiary      no    2143     yes   no
## 2   44    technician   single secondary      no      29     yes   no
## 3   33 entrepreneur  married secondary      no       2     yes  yes
## 4   47   blue-collar  married   unknown      no    1506     yes   no
...
```

Transformations 2

```
mutate(bankData, BalanceByDuration = balance/duration)
```

```
## Source: local data frame [45,211 x 18]
##
##    age           job  marital education default balance housing loan
## 1   58    management  married  tertiary      no    2143     yes   no
## 2   44    technician   single secondary      no      29     yes   no
## 3   33 entrepreneur  married secondary      no       2     yes  yes
## 4   47   blue-collar  married   unknown      no    1506     yes   no
...
```

Transmute

`mutate()` retains all columns. If you only want to keep the new transforms, you can use `transmute()`

Transmute Example

```
transmute(bankData, BalanceByDuration = balance/duration)
```

```
## Source: local data frame [45,211 x 1]
##
##    BalanceByDuration
## 1         8.210727969
## 2         0.192052980
## 3         0.026315789
## 4        16.369565217
...
```

Summarise Data by Groups

- The `group_by` verb creates a grouping by a categorical variable
- Functions can be placed inside `summarise` to create summary functions

```
args(group_by)
```

```
## function (.data, ..., add = FALSE)
## NULL
```

Example group_by 1

```
summarise(group_by(bankData, default), Num = n())
```

```
## Source: local data frame [2 x 2]
##
##   default   Num
## 1      no 44396
## 2     yes   815
```

Example group_by 2

```
summarise(group_by(bankData, default), Ave.Balance = mean(balance))
```

```
## Source: local data frame [2 x 2]
##
##   default Ave.Balance
## 1      no   1389.8064
## 2     yes   -137.6245
```

Example group_by 3

```
summarise(group_by(bankData, default), Ave.Balance = mean(balance),
  Num = n())
```

```
## Source: local data frame [2 x 3]
##
##   default Ave.Balance   Num
## 1      no   1389.8064 44396
## 2     yes   -137.6245   815
```

Outline

1 The Grammar of Data Manipulation

2 Filtering and Reordering Data

3 Transformations and Summaries

4 Chaining Verbs

5 Advanced Topics

Chaining/Piping

- A `dplyr` installation includes the `magrittr` package as a dependency
- The `magrittr` package includes a pipe operator that allows you to pass the current dataset to another function
- This makes interpreting a nested sequence of operations much easier to understand

Standard Code

Code is executed inside-out.

```
arrange(filter(select(bankData, age, job, education, default), default ==
  "yes"), job, education, age)
```

```
## Source: local data frame [815 x 4]
##
##    age    job education default
## 1   25 admin. secondary     yes
## 2   26 admin. secondary     yes
## 3   26 admin. secondary     yes
## 4   26 admin. secondary     yes
...
```

Reformatted

```
arrange(
  filter(
    select(bankData, age, job, education, default),
    default == 'yes'),
  job, education, age)
```

```
## Source: local data frame [815 x 4]
##
##    age    job education default
## 1   25 admin. secondary     yes
## 2   26 admin. secondary     yes
## 3   26 admin. secondary     yes
## 4   26 admin. secondary     yes
...
```

With Piping

```
bankData %>%
  select(age, job, education, default) %>%
  filter(default == 'yes') %>%
  arrange(job, education, age)
```

```
## Source: local data frame [815 x 4]
##
##    age    job education default
## 1   25 admin. secondary     yes
## 2   26 admin. secondary     yes
## 3   26 admin. secondary     yes
## 4   26 admin. secondary     yes
...
```

More General Piping

- Piping is not restricted to `dplyr` manipulation tasks

```
x1 <- rnorm(10)
x2 <- rnorm(10)
sqrt(sum((x1 - x2)^2))
```

```
## [1] 5.17407
```

```
(x1 - x2)^2 %>% sum() %>% sqrt()
```

```
## [1] 5.17407
```

Pipe + group_by()

The pipe operator is very helpful for group by summaries

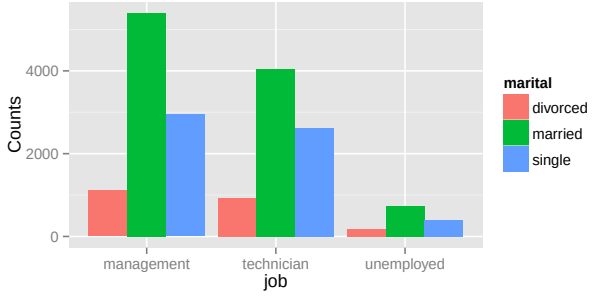
```
bankData %>% group_by(job) %>% summarise(Number = n(), Average.Balance = mean(balance),
  Number.Defaulted = sum(default == "yes"), Default.Rate = Number.Defaulted/Number)
```

```
## Source: local data frame [12 x 5]
##
##              job Number Average.Balance Number.Defaulted Default.Rate
## 1         admin.   5171       1135.8389               74  0.014310578
## 2    blue-collar   9732       1078.8267              201  0.020653514
## 3   entrepreneur   1487       1521.4701               55  0.036987223
## 4      housemaid   1240       1392.3952               22  0.017741935
...
```

Pipe and Plot

As a reminder, piping can also be used for non-dplyr functions.

```
library(ggplot2)
bankData %>% filter(job %in% c("management", "technician", "unemployed")) %>%
  group_by(job, marital) %>% summarise(Counts = n()) %>% ggplot() +
  geom_bar(aes(x = job, y = Counts, fill = marital), stat = "identity",
    position = "dodge")
```



Piping: Unique Values

Piping is also very helpful with identifying unique rows. You can also use `distinct()` to identify unique rows and is typically used with `arrange()`.

```
bankData %>% select(job, marital, education, default, housing, loan,
  contact) %>% arrange(job, marital, education, default, housing,
  loan, contact) %>% distinct()
```

```
## Source: local data frame [1,482 x 7]
##
##        job  marital education default housing loan   contact
## 1  admin. divorced   primary      no      no   no  cellular
## 2  admin. divorced   primary      no      no   no   unknown
## 3  admin. divorced   primary      no      no  yes  cellular
## 4  admin. divorced   primary      no     yes   no  cellular
...
```

Unique Keys

You can specify variables that you only want unique values for.

```
bankData %>% select(job, marital, education, default, housing, loan,
  contact) %>% arrange(job, marital, education, default, housing,
  loan, contact) %>% distinct(job, marital, education)
```

```
## Source: local data frame [144 x 7]
##
##       job  marital education default housing loan  contact
## 1  admin. divorced   primary      no      no   no cellular
## 2  admin. divorced secondary      no      no   no cellular
## 3  admin. divorced  tertiary      no      no   no cellular
## 4  admin. divorced   unknown      no      no   no cellular
...
```

Unique Keys

It will keep the first row with those particular key values.

```
bankData %>% select(job, marital, education, default, housing, loan,
  contact) %>% arrange(job, marital, education, desc(default), desc(housing),
  desc(loan), desc(contact)) %>% distinct(job, marital, education)
```

```
## Source: local data frame [144 x 7]
##
##       job  marital education default housing loan  contact
## 1  admin. divorced   primary      no     yes  yes  unknown
## 2  admin. divorced secondary     yes     yes  yes  unknown
## 3  admin. divorced  tertiary     yes      no  yes cellular
## 4  admin. divorced   unknown      no     yes   no  unknown
...
```

Exercise

Your turn:

- Use the pipe operator to group by job and housing status
- Calculate the counts of observations, and the average and median balance

Summary

`mutate()` Create transformations

`summarise()` Aggregate

`group_by()` Group your dataset by levels

`distinct()` Extract unique values (frequently used with `arrange()`)

Chaining with the %>% operator can result in more readable code.

Questions?

Outline

1 The Grammar of Data Manipulation

2 Filtering and Reordering Data

3 Transformations and Summaries

4 Chaining Verbs

5 Advanced Topics

Multiple Columns

You can `summarise` or `mutate` multiple columns using the same grouping variable.

- `summarise_each` allows you to apply the same summary function to multiple columns
- `mutate_each` also does a similar manipulation for mutate

```
help(summarise_each)
```

Both use the `funs()` function.

Summarise_each Example

```
bankData %>% group_by(education) %>% summarise_each(funs(mean), balance,
  duration)
```

```
## Source: local data frame [4 x 3]
##
##   education  balance duration
## 1   primary 1250.950 255.9330
## 2 secondary 1154.881 258.6858
## 3  tertiary 1758.416 258.5185
## 4   unknown 1526.754 257.3139
```

summarise_each Example 2

You can also use multiple functions.

```
bankData %>% group_by(education) %>% summarise_each(funs(min, mean,
  max), balance, duration)
```

```
## Source: local data frame [4 x 7]
##
##   education balance_min duration_min balance_mean duration_mean
## 1   primary       -2604            0     1250.950      255.9330
## 2 secondary       -8019            0     1154.881      258.6858
## 3  tertiary       -6847            2     1758.416      258.5185
## 4   unknown       -1445            4     1526.754      257.3139
## Variables not shown: balance_max (int), duration_max (int)
```

mutate_each Example

You can use the . to indicate where the variables go in an arbitrary function.

```
bankData %>% group_by(month) %>% select(balance, duration) %>% mutate_each(funs(half = ./2))
```

```
## Source: local data frame [45,211 x 3]
## Groups: month
##
##    month balance duration
## 1    may  1071.5    130.5
## 2    may    14.5     75.5
## 3    may     1.0     38.0
...
```

Additional Helper Functions

- Helper functions `n()` and `count()` count the number of rows in a group
- Helper function `n_distinct(vector)` counts the number of unique items in that vector

```
bankData %>% group_by(job, default) %>% summarise(education_levels = n_distinct(education))
```

```
## Source: local data frame [24 x 3]
## Groups: job
##
##             job default education_levels
## 1        admin.      no                4
## 2        admin.     yes                3
## 3   blue-collar      no                4
...
```

tally

`tally()` is a shortcut for counting

```
bankData %>% group_by(job) %>% tally()
```

```
## Source: local data frame [12 x 2]
##
##              job    n
## 1         admin. 5171
## 2    blue-collar 9732
## 3   entrepreneur 1487
## 4      housemaid 1240
...
```

Without tally()

```
bankData %>% group_by(job) %>% summarise(n = n())
```

```
## Source: local data frame [12 x 2]
##
##              job    n
## 1         admin. 5171
## 2    blue-collar 9732
## 3   entrepreneur 1487
## 4      housemaid 1240
...
```

count

`count()` makes it even easier.

```
bankData %>% count(job)
```

```
## Source: local data frame [12 x 2]
##
##              job    n
## 1          admin. 5171
## 2    blue-collar 9732
## 3   entrepreneur 1487
## 4      housemaid 1240
...
```

Ranking Variables

In base R, you can use `rank`.

```
args(rank)
```

```
## function (x, na.last = TRUE, ties.method = c("average", "first",
##     "random", "max", "min"))
## NULL
```

dplyr Helper Functions

row_number(x) Ties are ordered by row in which the observation is encountered.

ntile(x, n) Rough ranking into *n* bins

min_rank(x) Ties are assigned the min rank

dense_rank(x) Same as min, but no gaps in ranks

percent_rank(x) min_rank(x) scaled to [0, 1]

cume_dist(x) a cumulative distribution function

Rank Examples

```
bankData %>% slice(1:10) %>%
  transmute(Job = job,
            jobRankAvg = rank(job),
            jobRankRow = row_number(job),
            jobRankMin = min_rank(job),
            jobRankDense = dense_rank(job),
            jobRankPerc = percent_rank(job),
            jobRankCume = cume_dist(job))
```

```
## Source: local data frame [10 x 7]
##
##             Job jobRankAvg jobRankRow jobRankMin jobRankDense jobRankPerc
## 1    management        5.0          4          4            3   0.3333333
## 2    technician        8.5          8          8            5   0.7777778
## 3  entrepreneur        2.5          2          2            2   0.1111111
## 4   blue-collar        1.0          1          1            1   0.0000000
...
```

Applying custom functions

- You can also apply your own custom functions using `do()`

```
set.seed(1)
df <- data.frame(houseID = rep(1:10, each = 10), year = 1995:2004,
  price = ifelse(runif(10 * 10) > 0.5, NA, exp(rnorm(10 * 10))))
head(df)
```

```
##   houseID year     price
## 1       1 1995 1.4890017
## 2       1 1996 0.5422509
## 3       1 1997        NA
## 4       1 1998        NA
## 5       1 1999 4.1913535
## 6       1 2000        NA
```

Dragging Down Values

- In our custom dataframe, we have a missing value for some years
- We would like to pull down the previous year's value

```
library(xts)
df %>% group_by(houseID) %>% do(na.locf(.))
```

```
## Source: local data frame [100 x 3]
## Groups: houseID
##
##    houseID year     price
## 1        1 1995 1.4890017
## 2        1 1996 0.5422509
## 3        1 1997 0.5422509
...
```

Grouped Tests

```
bankData %>% filter(marital %in% c("married", "single")) %>% group_by(job) %>%
  do(tTest = t.test(age ~ marital, data = .)) %>% mutate(tTestPVal = get("p.value",
  tTest), tTestStat = get("statistic", tTest))
```

```
## Source: local data frame [12 x 4]
## Groups: <by row>
##
##              job      tTest     tTestPVal tTestStat
## 1         admin. <S3:htest> 7.772104e-164 28.547365
## 2    blue-collar <S3:htest> 8.743035e-318 41.963819
## 3   entrepreneur <S3:htest>  2.302137e-35 13.732785
...
```

Connecting to a database

- `dplyr` provides capabilities to connect to remote databases
- This is useful in case your data is too large to fit in memory
- We will show how to connect to a remote `sqlite` database

Database Setup

```
library(DBI)
library(RSQLite)
outPath <- "../output"
if (!file.exists(outPath)) dir.create(outPath)
bankSQL <- file.path(outPath, "bank.sqlite")
db <- dbConnect(SQLite(), dbname = bankSQL)
dbWriteTable(conn = db, name = "bankfull", value = bankCSV, sep = ";",
  header = T, overwrite = T)
```

```
## [1] TRUE
```

dplyr and SQL

We connect with `src_sqlite()`

```
args(src_sqlite)
```

```
## function (path, create = FALSE)
## NULL
```

Connect with dplyr

```
db.new <- src_sqlite(bankSQL)
mytbl <- tbl(db.new, "bankfull")
mytbl %>% group_by(job) %>% tally()
```

```
## Source: sqlite 3.8.6 [../output/bank.sqlite]
## From: <derived table> [?? x 2]
##
##                job    n
## 1        "admin." 5171
## 2   "blue-collar" 9732
## 3  "entrepreneur" 1487
...
```

Set Operations

`intersect`, `union`, `setdiff`, `setequal`

all have methods for data frames, data tables and SQL db tables

```
args(dplyr::intersect)
```

```
## function (x, y, ...)
## NULL
```

Joins

There are methods for doing joins (merges) in dplyr

Base tool: `merge()`

dplyr tool: `XXXX_join()`

```
help(join)
```

Summary

`dplyr` is a powerful and **fast** package designed to facilitate many common data manipulation problems.

- Subsetting rows (`filter()` and `slice()`) and columns (`select()` and `rename()`)
- Generating transformations (`mutate()` and `transmute()`)
- Aggregation (`summarise()` and `group_by()`)
- Sorting (`arrange()`), unique value identification (`distinct()`), and ranking (`xxx_rank()`)
- Providing SQL connectivity

Questions?

REVOLUTION ANALYTICS

Thank you

Revolution Analytics is the leading commercial provider of software and support for the popular open source R statistics language.

www.revolutionanalytics.com
1.855.GET.REVO
Twitter: @RevolutionR