Data Manipulation Using Dplyr

Xuemao Zhang
Department of Mathematics
East Stroudsburg University

June 25, 2019

Outline

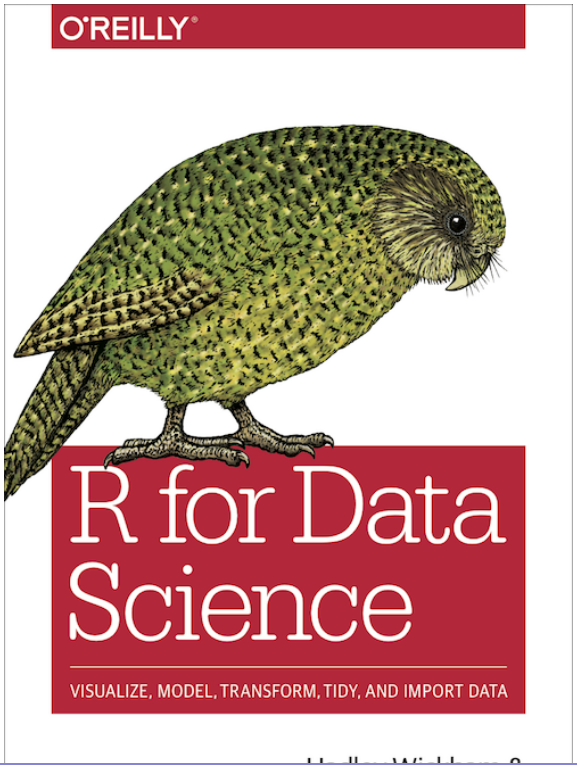
- Overview
- Filter observations
- Select/drop variables
- Add variables
- Rename variables
- Sort data
- Remove duplicate rows
- Summaries
- The %>% operator
- Merge data sets

Install the following packages if you don't have them.

```
install.packages("dplyr");
install.packages("ggplot2");
```

Overview

- `dplyr` is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges.
- The dplyr package is one of the most powerful and popular package in R.
- The author of ggplot2, Hadley Wickham, is also the author of the package.
- R for Data Science (O'Reilly 2017) by Hadley Wickham
- Free Online: https://r4ds.had.co.nz/



Overview

- Data Exploration = Data manipulation + Data visualization.
- **Data wrangling** (a term used in data science) is the process of transforming/mapping data from raw format into ready-to-analyze format.
- Besides `ggplot2()` for data visualization, Hadley Wickham created a series of R packages for data wrangling, including
  - `tidyr` for reshaping your data for plotting and use by different R functions
  - `tibble` for better ways to create, print and subset data frames
  - `dplyr` for data manipulation will be covered in this data camp
- `dplyr` functions process **faster** than base R functions. It is because `dplyr` functions were written in a computationally efficient manner. They are also more stable in the syntax and better supports data frames than vectors.

Overview

- There are 8 fundamental data manipulation verbs in `dplyr` that you will use to do most of your data manipulations.
  - `mutate()` and `transmutate()`: Add/create new variables.
  - `select()`: Select columns (variables) by their names.
  - `filter()`: Pick rows (observations/samples) based on their values.
  - `distinct()`: Remove duplicate rows.
  - `arrange()`: Reorder the rows.
  - `rename()`: Rename columns.
  - `summarise()`: Compute statistical summaries (e.g., computing the mean or the sum) and thus reduce multiple values down to a single summary. It is similar to `R::base::aggregate`
    - ⋆ `group_by()` to group variables for summarise
- R::base:merge() to combine two data.frames (or R::dplyr xxx_joins)

Note. The double colon `::` is used to specify the package where a function is from: `packagename::functionname()`.

Overview

- All these functions work similarly as follow:
  - The first argument is a data frame
  - The subsequent arguments are comma separated list of unquoted variable names and the specification of what you want to do
  - The result is a new data frame
- A special feature in `dplyr` is that you can chain your data manipulation operations using the pipe operator (%>%).

Note. `dplyr package allows to use the forward-pipe chaining operator (%>%) for combining multiple operations. For example, x %>% f is equivalent to f(x). Using the pipe (%>%), the output of each operation is passed to the next operation. This makes R programming easy:`

```
# input      +-------+      +-------+      +-------+      result
# data  %>% |  verb | %>% |  verb | %>% |  verb | ->   data
# frame      +-------+      +-------+      +-------+      frame
```

Filter observations

- The function `filter()` is used to subset data with matching logical conditions.
- It is Similar to base::`which()` or `subset()`

```
library(dplyr);
library(ggplot2);
str(diamonds);
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53940 obs. of  10 va
##  $ carat   : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.2
##  $ cut     : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3
##  $ color   : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2
##  $ depth   : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.
##  $ table   : num  55 61 65 58 58 57 57 55 61 61 ...
##  $ price   : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ x       : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ..
##  $ y       : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.0
##  $ z       : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.3
```

Filter observations

- The data set (prices of round cut diamonds) contains 53,940 observations of 10 variables.
  - price: price in US dollars ($326–$18,823)
  - carat: weight of the diamond (0.2–5.01)
  - cut: quality of the cut (Fair, Good, Very Good, Premium, Ideal)
  - color: diamond colour, from J (worst) to D (best)
  - clarity: a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best))
  - x: length in mm (0–10.74)
  - y: width in mm (0–58.9)
  - z: depth in mm (0–31.8)
  - depth: total depth percentage = z / mean(x, y) = 2 * z / (x + y) (43–79)
  - table: width of top of diamond relative to widest point (43–95)

Filter observations

- base::unique() returns unique values of a variable/data frame.

```
unique(diamonds$cut);
```

```
## [1] Ideal     Premium   Good      Very Good Fair
## Levels: Fair < Good < Very Good < Premium < Ideal
```

```
unique(diamonds$color);
```

```
## [1] E I J H F G D
## Levels: D < E < F < G < H < I < J
```

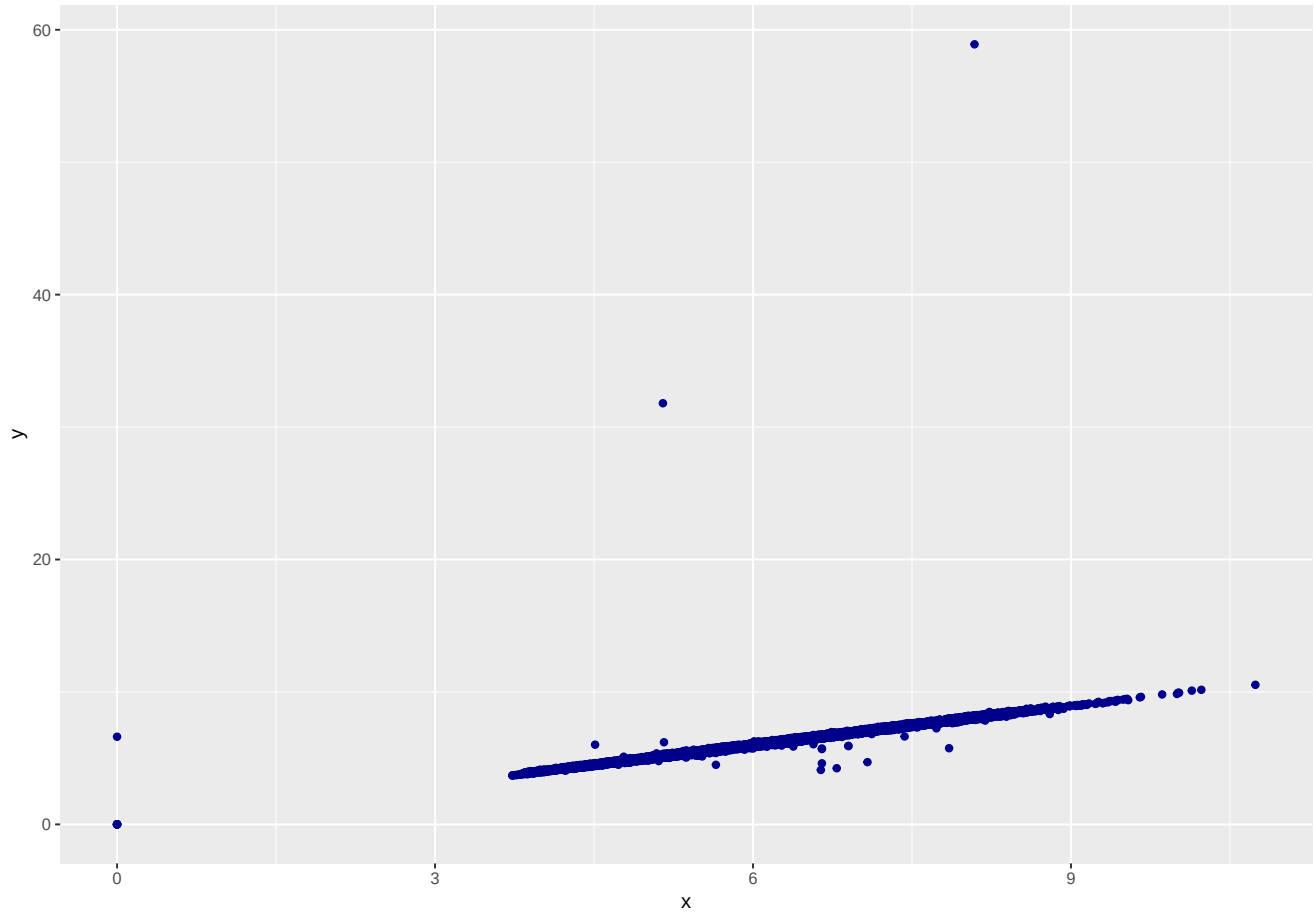
```
unique(diamonds$clarity);
```

```
## [1] SI2  SI1  VS1  VS2  VVS2 VVS1 I1   IF
## Levels: I1 < SI2 < SI1 < VS2 < VS1 < VVS2 < VVS1 < IF
```

Filter observations

- Most data points lie along a line, but thre are outliers.

```
ggplot(diamonds, aes(x,y)) + geom_point(color='darkblue');
```



Filter observations

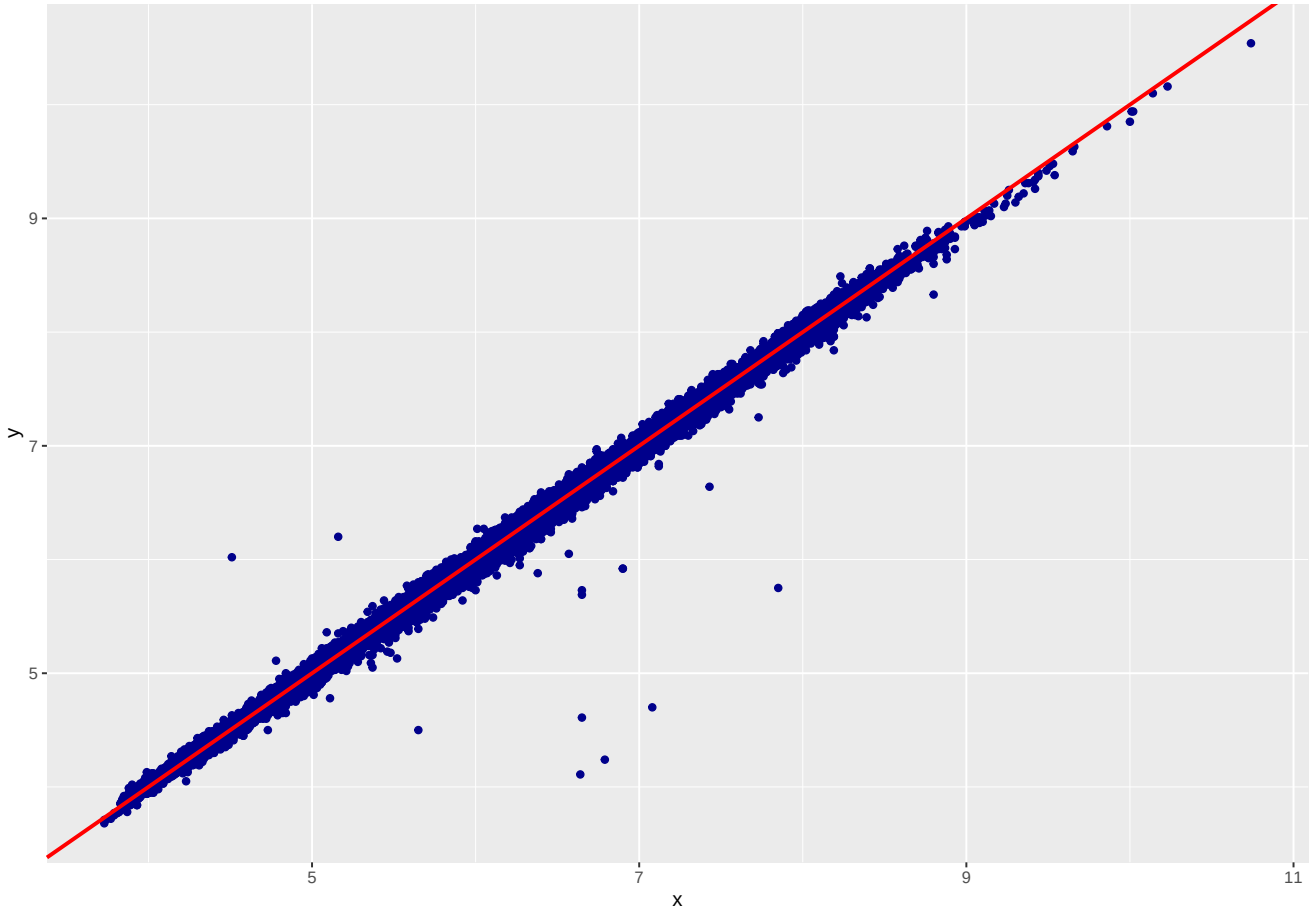
```
filter(diamonds, x==0 | y==0);
```

```
## # A tibble: 8 x 10
##   carat cut       color clarity depth table price     x     y
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <db
## 1  1.07 Ideal     F     SI2      61.6    56  4954     0  6.62
## 2  1    Very Good H     VS2      63.3    53  5139     0  0
## 3  1.14 Fair      G     VS1      57.5    67  6381     0  0
## 4  1.56 Ideal     G     VS2      62.2    54 12800     0  0
## 5  1.2  Premium   D     VVS1     62.1    59 15686     0  0
## 6  2.25 Premium   H     SI2      62.8    59 18034     0  0
## 7  0.71 Good      F     SI2      64.1    60  2130     0  0
## 8  0.71 Good      F     SI2      64.1    60  2130     0  0
```

Filter observations

- Multiple arguments can be provided to `filter()`.

```
diamonds_ok=filter(diamonds, x>0, y>0,y<20);
ggplot(diamonds_ok, aes(x,y)) + geom_point(color='darkblue') +
  geom_abline(color='red',size=1);
```



Filter observations

- `filter()` is similar to `subset()`.
- `subset()` can select both variables (columns) and observations(rows).
- `filter()` works exclusively for observations(rows).
- Two useful comparison operators `%in%` and `is.na()`:
  - `x %in% c("a","b","c")` is used to check if `x` is one of the values in the right hand side.
  - `is.na()` is used to check if a value is missing.

Select/drop variables

- select() allows you to select specific columns from your data.

```
diamonds_ok1=select(diamonds_ok, carat, cut, color, price, clarity);
str(diamonds_ok1);
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53930 obs. of  5 var
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.2
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2
```

Select/drop variables

- select() allows you to drop specific columns from your data.

```
diamonds_ok2=select(diamonds_ok, -c(carat, cut, color,
                                    price, clarity));
str(diamonds_ok2);
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53930 obs. of  5 var
##  $ depth: num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4
##  $ table: num  55 61 65 58 58 57 57 55 61 61 ...
##  $ x    : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y    : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05
##  $ z    : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39
```

Add variables

- `mutate()` allows you to add variables to your dataset which is a common task in any programming environment.

```
diamonds_ok3=mutate(diamonds_ok1, price_per_carat = price/carat);
str(diamonds_ok3);
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53930 obs. of  6 var
##  $ carat          : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26
##  $ cut            : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4
##  $ color          : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2
##  $ price          : int  326 326 327 334 335 336 336 337 337 338
##  $ clarity        : Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2
##  $ price_per_carat: num  1417 1552 1422 1152 1081 ...
```

Rename variables

- rename( ) function is used to change variable name.

```
rename(data , new_name = old_name)
```

```
diamonds_ok4=rename(diamonds_ok3, PPC= price_per_carat );
#diamonds_ok4=mutate(diamonds_ok3, PPC= price_per_carat );
str(diamonds_ok4);
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53930 obs. of  6 var
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.2
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2
##  $ PPC    : num  1417 1552 1422 1152 1081 ...
```

Sort data

- arrange() sorts your data. In base R, this is commonly done with order().

```
data=data.frame(x = c(2,3,5,1,4), y=c(10, 9, 7, 8, 6));
arrange(data,x,y);
```

```
##   x  y
## 1 1  8
## 2 2 10
## 3 3  9
## 4 4  6
## 5 5  7
```

```
arrange(data, desc(x));
```

```
##   x  y
## 1 5  7
## 2 4  6
## 3 3  9
## 4 2 10
## 5 1  8
```

Remove duplicate rows

- The `distinct()` function is used to eliminate duplicates.
- Remove duplicate rows based on all variables

```
dim(diamonds);
```

```
## [1] 53940    10
```

```
diamonds1 = distinct(diamonds);
dim(diamonds1);
```

```
## [1] 53794    10
```

- Remove duplicate rows based on some variables
  - `.keep_all` option is used to retain all other variables in the output data frame.

```
diamonds2 = distinct(diamonds, carat, cut, color, price,
                     clarity, .keep_all= TRUE);
dim(diamonds2);
```

```
## [1] 39756    10
```

Summaries

- summarize() Summarize allows you to compute summary statistics.
- summarize() becomes extremely useful when combined with group_by().

```
by_clarity = group_by(diamonds_ok, clarity)
sum_clarity = summarize(by_clarity, price=mean(price));
sum_clarity;
```

```
## # A tibble: 8 x 2
##   clarity price
##   <ord>   <dbl>
## 1 I1      3924.
## 2 SI2     5061.
## 3 SI1     3996.
## 4 VS2     3924.
## 5 VS1     3839.
## 6 VVS2    3284.
## 7 VVS1    2520.
## 8 IF      2865.
```

Summaries

```
ggplot(sum_clarity, aes(clarity,price)) +
  geom_line(aes(group=1),color="green")+
  geom_point(size=2);
```

Summaries

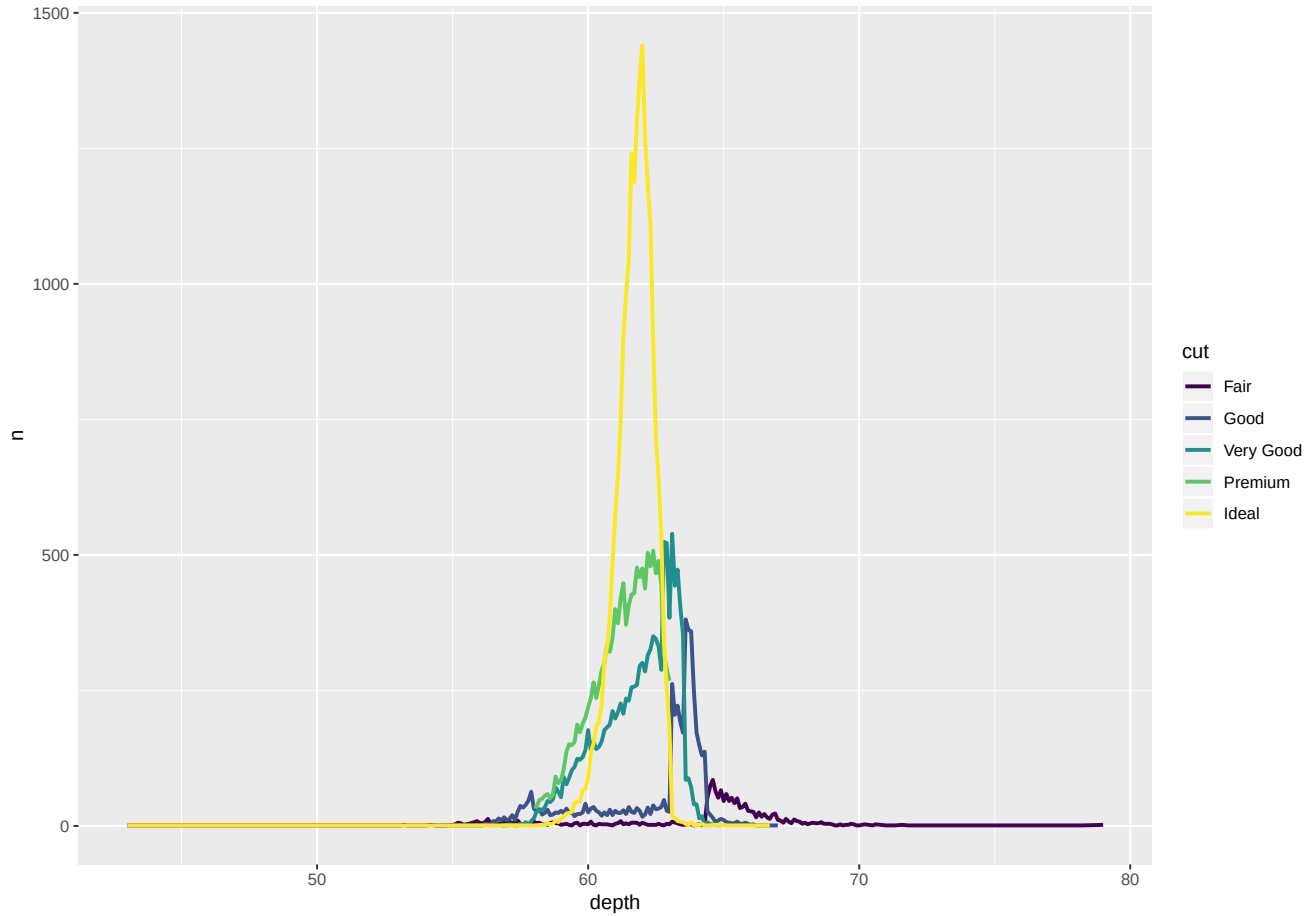
- `group_by()` function allows us to create groups based on more than one variable: `group_by(data, variables)`.
- The special summary function `n()` returns counts.

```
cut_depth =  summarize(group_by(diamonds_ok, cut, depth), n=n());
cut_depth;
```

```
## # A tibble: 492 x 3
## # Groups:   cut [5]
##    cut   depth     n
##    <ord> <dbl> <int>
##  1 Fair   43       1
##  2 Fair   44       1
##  3 Fair   50.8     1
##  4 Fair   51       1
##  5 Fair   52.2     1
##  6 Fair   52.3     1
##  7 Fair   52.7     1
##  8 Fair   53       1
##  9 Fair   53.1     1
```

Summaries

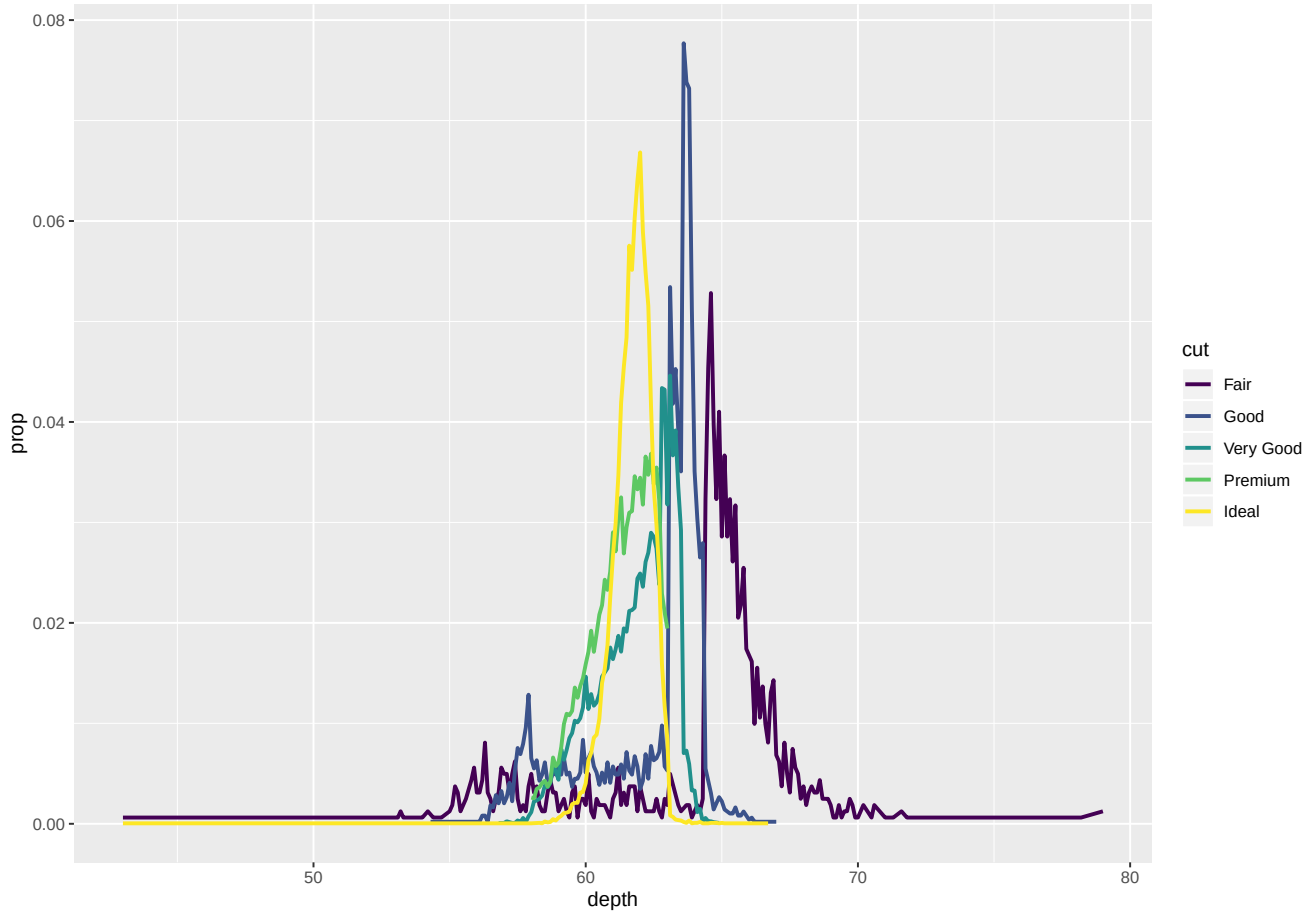
```
ggplot(cut_depth, aes(depth,n, color=cut)) +
  geom_line(size=1);
```



Summaries

- We can use a grouped `mutate()` to convert counts to proportions.

```
cut_depth = mutate(cut_depth, prop=n/sum(n));
ggplot(cut_depth, aes(depth, prop, color=cut)) +
  geom_line(size=1);
```



The %>% operator

- Powerful trick for coding a sequence of operations
- Output of old operation as the first argument of new operation
- Especially useful in combined with ggplot2

```
# input      +-------+      +-------+      +-------+      result
# data  %>% |  verb |  %>% |  verb |  %>% |  verb | ->   data
# frame      +-------+      +-------+      +-------+      frame
```

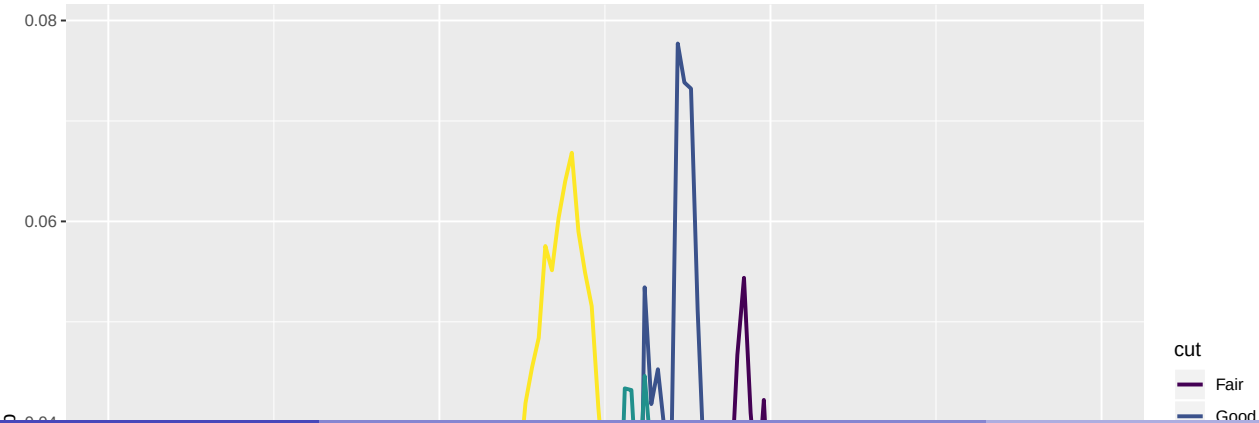
- For example, you use multiple verbs for an analysis.

```
diamonds_ok = filter(diamonds, x>0, y>0,y<20);
cut_depth = group_by(diamonds_ok, cut, depth);
cut_depth = summarize(cut_depth, n=n());
cut_depth = filter(cut_depth, depth>55, depth<70);
cut_depth = mutate(cut_depth, prop=n/sum(n));
```

The %>% operator

- This sequence of operations is a bit painful because we repeated the name of the data frame many times.
- With the **pipe** %>%, the above sequence of operations can be rewritten and used with ggplot.

```
diamonds %>%
filter(x>0, y>0,y<20)%>%
group_by(cut, depth)%>%
summarize( n=n())%>%
filter(depth>55, depth<70)%>%
mutate(prop=n/sum(n))%>%
 ggplot( aes(depth, prop, color=cut)) +
 geom_line(size=1);
```



Merge data sets

- base::`merge()` function.
- dplyr has its own version of this in the form of several **functions**: `left_join`, `right_join`, `inner_join`, `full_join`, `anti_join`.
- The difference between these functions is what happens when there is a row in one data frame without a corresponding row in the other data frame.
  - `inner_join` discards such rows.
  - `full_join` always keeps them, filling in missing data with NA.
  - `left_join` always keeps rows from the first data frame
  - `right_join` always keeps rows from the second data frame
  - `anti_join` is a bit different, it gives you rows from the first data frame that aren't in the second data frame.

Merge data sets

```
#suppose country and year determines the ID
data1= data.frame(country = c("A","A","B","B","C","C"),
         year=c(2017,2018,2017,2017,2017,2018),
         x1=c(1,2,3,4,5,6));
data2= data.frame(country = c("A","A","B","B","C"),
         year=c(2017,2018,2017,2018,2018),
         x2=c(7,8,9,10,11));
```

Merge data sets

```
data1;
```

```
##   country year x1
## 1       A 2017  1
## 2       A 2018  2
## 3       B 2017  3
## 4       B 2017  4
## 5       C 2017  5
## 6       C 2018  6
```

```
data2;
```

```
##   country year x2
## 1       A 2017  7
## 2       A 2018  8
## 3       B 2017  9
## 4       B 2018 10
## 5       C 2018 11
```

Merge data sets

- inner_join returns matching rows only.

```
inner_join(data1,data2, by=c("country","year"));
```

```
##   country year x1 x2
## 1       A 2017  1  7
## 2       A 2018  2  8
## 3       B 2017  3  9
## 4       B 2017  4  9
## 5       C 2018  6 11
```

Merge data sets

- full_join returns all rows.

```
full_join(data1,data2, by=c("country","year"));
```

```
##   country year x1 x2
## 1       A 2017  1  7
## 2       A 2018  2  8
## 3       B 2017  3  9
## 4       B 2017  4  9
## 5       C 2017  5 NA
## 6       C 2018  6 11
## 7       B 2018 NA 10
```

Merge data sets

- `left_join` always keeps rows from the first data frame

```
left_join(data1,data2, by=c("country","year"));
```

```
##   country year x1 x2
## 1       A 2017  1  7
## 2       A 2018  2  8
## 3       B 2017  3  9
## 4       B 2017  4  9
## 5       C 2017  5 NA
## 6       C 2018  6 11
```

Merge data sets

- `right_join` always keeps rows from the second data frame

```
right_join(data1,data2, by=c("country","year"));
```

```
##   country year x1 x2
## 1       A 2017  1  7
## 2       A 2018  2  8
## 3       B 2017  3  9
## 4       B 2017  4  9
## 5       B 2018 NA 10
## 6       C 2018  6 11
```

Merge data sets

- anti_join returns rows from the first data frame that aren't in the second data frame.

```
anti_join(data1,data2, by=c("country","year"));
```

```
##   country year x1
## 1       C 2017  5
```

Questions?

https://dplyr.tidyverse.org/