Tidying Data with R

Lecture 4

Today's Lecture

The goal of this lecture is to delve deeper into common forms of data work. The reason we work with data is to find answers of interest to us, but the data we find or receive is rarely in the form we need. Oftentimes this impedes our ability to even begin such an analysis. Common issues with messy data and our topics for today:

- Vectors/variables are of the wrong type
  - Dates
- Strings have extra, unneeded information
  - Extracting parts of a character string
  - Fixing typos
- Information is spread across multiple tables
  - Merging data with dplyr
- Information in a table is difficult to extract
  - Wide and long form data
  - Reshaping data

Introduction to data and research goal

The research goal for this lecture is to study the effects of banks' characteristics on lending behavior. Some key measurements relevant to this research might be:

- Number of loans/total principal issued every month
- Type of banks with heavy lending activity vs. little lending activity
- Proportion of loans, between private and government loans, being issued
- etc. Can you name some more?

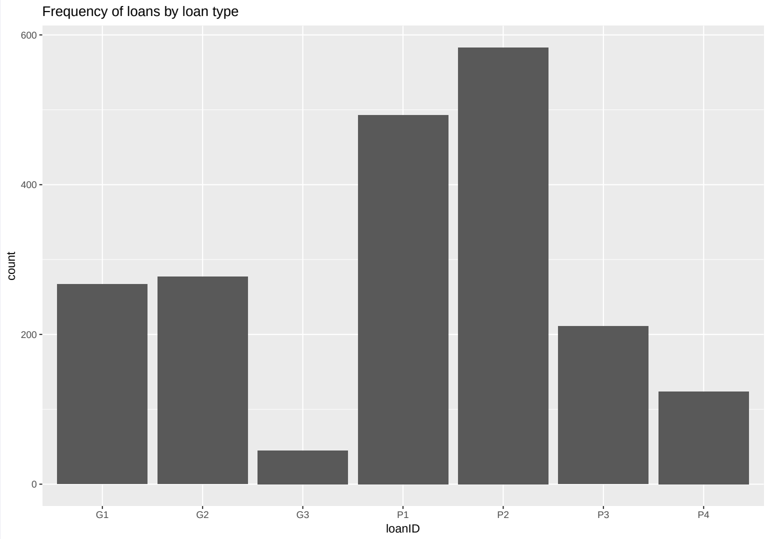
The available data

To address our question(s) we will use what we have available to us, the banks dataset.

```
head(banks)
```

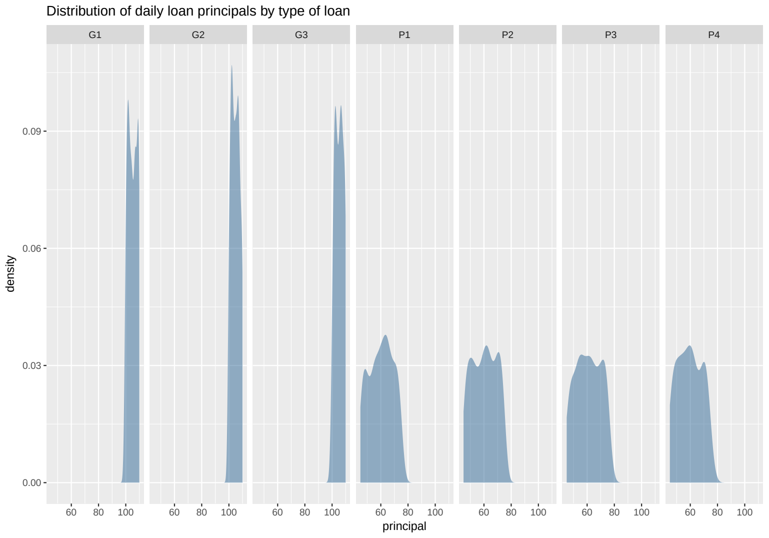
```
## # A tibble: 6 x 6
##         date bankID       type loanID principal
##        <chr>  <dbl>      <chr>  <chr>     <int>
## 1 2016-09-18   5213 Commercial     P3        57
## 2 2016-05-04   4903    Savings     P4        53
## 3 2016-07-09   2682 Commercial     P1        60
## 4 2016-08-10   6396    Savings     P2        68
## 5 2016-09-29   4903    Savings     P2        64
## 6 2015-08-21   6396     Saving     P3        45
## # ... with 1 more variables: loan_num <int>
```

Banks data



Frequency of loans by loan type
600
400
200
0
count
G1
G2
G3
P1
P2
P3
P4
loanID

Banks continued



Important questions to consider:

- What in banks seems useful?
- What seems unneeded?
- Can you think of any additional variables that would be useful for us?

Let's zoom in on the date variable. What type of vector was the date variable read in as?

1. Vectors/variables are of the wrong type

Dates in R

- R treats dates as a uniquely-formatted vector type, represented as: "YYYY-MM-DD".
- The as.Date() function can convert strings into this date vector type.
- Since dates are a vector type, they can be assigned to objects like any other variable, and R can do most operations (that make sense) on dates, simply as part of the basic R package.

```r
dates <- c(as.Date("2010-01-01"),
           as.Date("2010-12-31"))

print(dates)
```

```
## [1] "2010-01-01" "2010-12-31"
```

Dates can behave like numbers

```r
mean(dates)
```

```
## [1] "2010-07-02"
```

```r
min(dates)
```

```
## [1] "2010-01-01"
```

```r
as.Date("2010-02-10") + 10
```

```
## [1] "2010-02-20"
```

Formatting dates

R can convert many, many different looking strings into a date class using the "format" argument in as.Date:

```r
as.Date("19800210", format = "%Y%m%d")
```

```
## [1] "1980-02-10"
```

```r
as.Date("10Feb80", format = "%d%b%y")
```

```
## [1] "1980-02-10"
```

```r
as.Date("9/18/2016", format = "%m/%d/%Y")
```

```
## [1] "2016-09-18"
```

Date symbols for reference

Symbol	Meaning
%d	Day as a number
%a	Abbreviated Weekday
%A	Unabbreviated Weekday
%m	Month as a number
%b	Abbreviated month
%B	Unabbreviated month
%y	Two-digit year
%Y	Four-digit year

In-class exercise:

Convert the following character vector to a date vector.

```
date_vec <- c("Day 5 of March, 2017",
              "Day 23 of May, 2017")
```

Format is both an argument and a function itself:

```
Sys.Date()
```

```
## [1] "2017-09-14"
```

```
format(Sys.Date(), "%d %B %Y")
```

```
## [1] "14 September 2017"
```

```
format(Sys.Date(), "%d-%b")
```

```
## [1] "14-Sep"
```

```
format(Sys.Date(), "%Y")
```

```
## [1] "2017"
```

R can even use the seq command to generate strings of dates, which can be helpful when cleaning data.

```
dates <- seq(as.Date("2015-01-01"),
             to = as.Date("2015-12-31"),
             by = "month")
print(dates)
```

```
##  [1] "2015-01-01" "2015-02-01" "2015-03-01"
##  [4] "2015-04-01" "2015-05-01" "2015-06-01"
##  [7] "2015-07-01" "2015-08-01" "2015-09-01"
## [10] "2015-10-01" "2015-11-01" "2015-12-01"
```

```
mean(dates)
```

```
## [1] "2015-06-16"
```

```
max(dates)
```

In-class exercise:

Use the seq() command to generate a vector of every month-end date in 2010.

Rather technical notes:
In the case of two digit years, R (currently) assumes that years 00-68 are 2000 - 2068, and years 69-99 are 1969 - 1999.

R can even import from Excel, with Excel's wierd 5-digit dates.

```r
# From Windows Excel
as.Date(30829, origin = "1899-12-30")
```

```
## [1] "1984-05-27"
```

```r
# From iOS Excel
as.Date(29367, origin = "1904-01-01")
```

```
## [1] "1984-05-27"
```

Using lubridate to handle dates

```
# Lubridate introduction

# Date creation functions:
# ymd()
# dmy()
# mdy()

# Let's use a lubridate function to change date in
# banks to a Date vector
```

The date creation functions of lubridate do what as.Date() does, but the formatting conditions are not buried within an argument in the function; rather, each format type is denoted by the function name itself.

Accessor functions in lubridate

```
# year()
# month()
# mday()
# yday()
# wday()

# We will use accessor functions to create the
# following variables in banks:
#     monthonly
#     yearonly
```

- Now we know and, perhaps more importantly, can reference the month and the year belonging to each observation.
- Which functions from dplyr could we next use to calculate our desired monthly statistic(s)?
    - Total loans, by number and principal, issued per month

2. Strings have extra, unneeded information

Text patterns, splitting, and extracting

The three major charter types for depository institutions are:

- commercial banks,
- savings/thrift banks, and
- credit unions.

- Banks may behave differently based on their charter type.
- We will want to make sure charter type is reasonably coded in our dataset.

```r
# Let's see what kinds of charters we have in the
# variable 'type'
unique(banks$type)
```

```
## [1] "Commercial"    "Savings"       "Saving"
## [4] "Credit Union" "savins"        "savngi"
## [7] "Savigns"       "savin"
```

Using stringr to handle character strings

- Every stringr function begins with the prefix str_.
- The stringr functions are a toolbox of string manipulators.
- As you will see for yourself, many of these string functions have to do with locating patterns in text.

```r
# str_detect returns a logical vector for where
# the pattern was found
str_detect(string = c("a", "b", "c"), pattern = "b")
```

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

```r
# str_which returns the indices where the pattern
# was found
str_which(string = c("a", "b", "c"), pattern = "b")
```

```
## [1] 2
```

```r
# str_replace replaces patterns within text strings
str_replace("abc", pattern = "a", replacement = "c")
```

```
## [1] "cbc"
```

We can use str_detect() to filter to observations that somewhat look like Savings. Once we have identified these elements, we can then reassign all of those to the same phrase. First, we will need an introduction to str_to_lower().

```r
# str_to_lower
str_to_lower(c("A", "B", "C"))
```

```
## [1] "a" "b" "c"
```

```r
str_to_lower(c("Lower", "Case"))
```

```
## [1] "lower" "case"
```

```r
# Alternatively, there is a str_to_upper
str_to_upper(c("Upper", "Case"))
```

```
## [1] "UPPER" "CASE"
```

This is useful in our case since there are multiple variations of Savings both with and without upper case characters. To make our job of detecting these variations simpler, we will use a version of the variable 'type' where all characters are in the same case.

```
type_lower <- str_to_lower(banks$type)

# Using type_lower and str_detect(), let's fix
# the type variable
# banks <- banks %>%
#   mutate(type = )
```

```r
# Check unique values of 'type' again to see if we
# need to address more cases of misspellings
unique(banks$type)
```

```
## [1] "Commercial"   "Savings"       "Credit Union"
```

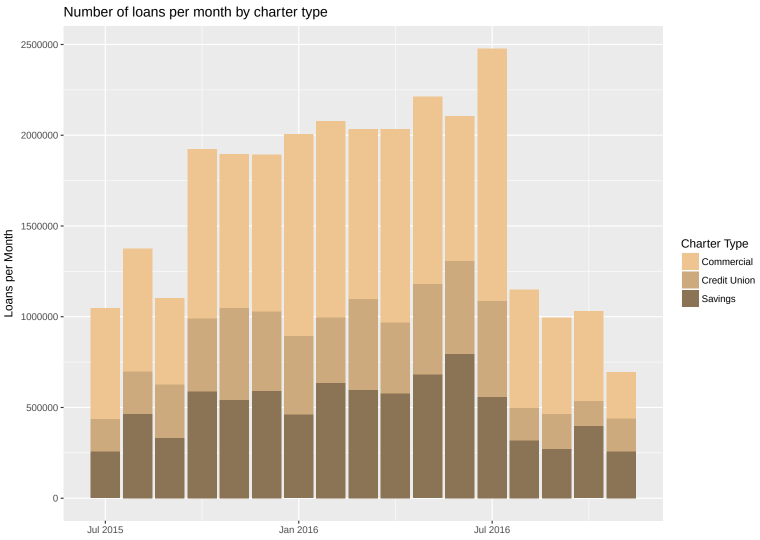
```r
# No longer need type_lower - can remove it to tidy up
rm(type_lower)
```

What are some tasks we can do that we could not before?

```r
TypeSmmry <- banks %>%
              group_by(yearonly, monthonly, type) %>%
              summarize(monthlyCount = sum(loan_num),
                        monthlyValue = sum(principal))
head(TypeSmmry)
```

```
## # A tibble: 6 x 5
## # Groups:   yearonly, monthonly [2]
##   yearonly monthonly          type monthlyCount
##      <dbl>     <dbl>         <chr>        <int>
## 1     2015         7    Commercial       610354
## 2     2015         7 Credit Union       177210
## 3     2015         7       Savings       258921
## 4     2015         8    Commercial       676567
## 5     2015         8 Credit Union       235746
## 6     2015         8       Savings       463309
## # ... with 1 more variables: monthlyValue <int>
```

- So for each true, unique type of bank in our dataset we have a monthly measure of issued loans by number and principal.
- Can you think of reasons why we aggregated our data up to the month level?



Number of loans per month by charter type
Loans per Month
2500000
2000000
1500000
1000000
500000
0
Jul 2015
Jan 2016
Jul 2016
Charter Type
Commercial
Credit Union
Savings

In Class Exercise:

Convert the following vector, numbers, to numeric type.

```r
numbers <- c("2,100", "3,250,000")
```

The next task concerns yet another measurement of interest: proportion of private to government loans.

- Do we have variables that allow us to identify private and government loans yet?

```r
# Let's take a look at loanID
unique(banks$loanID)
```

```
## [1] "P3" "P4" "P1" "P2" "G1" "G2" "G3"
```

- Every value in loanID begins with either a “P” or a “G”.
- “P” is shorthand for private, and so it denotes private loans, and “G” is shorthand for government
- What we really need are just the P’s and G’s
- Introducing str_sub():

```r
# To completely define a substring, you always only
# need three things:
## 1. the full string,
## 2. the starting position of the substring, and
## 3. the ending position of the substring.

# Extract characters from a text string
str_sub("abcd", start = 1, end = 3)
```

```
## [1] "abc"
```

```r
str_sub("abcd", start = 1, end = 2)
```

```
## [1] "ab"
```

To split a string apart into multiple pieces we would use str_split().

```r
# Splits a text string whenever the pattern
# argument is found
str_split("abcd", pattern = "c")
```

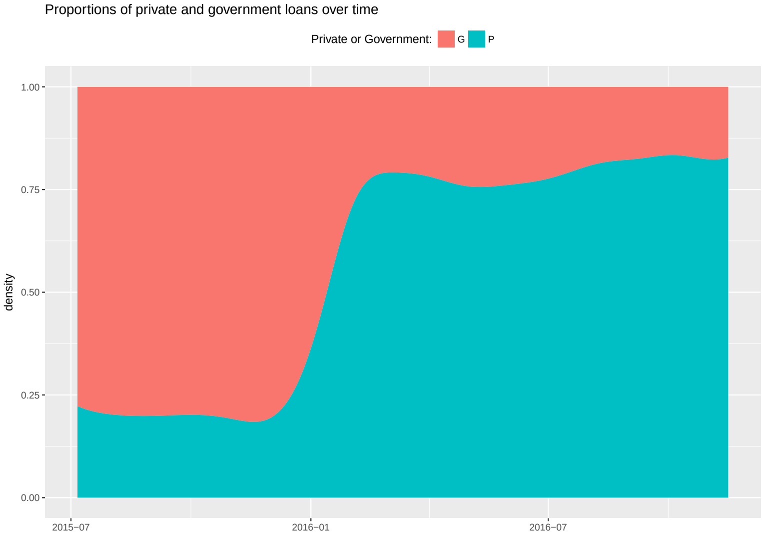
```
## [[1]]
## [1] "ab" "d"
```

```r
str_split("keep_these_parts", pattern = "_")
```

```
## [[1]]
## [1] "keep"  "these" "parts"
```

- Seeing these two stringr functions in action, which one could we use to arrive at our goal of extracting the P's and G's from loanID?

```
# We can use str_sub() to extract the P's and G's
# in loanID
# banks <- banks %>%
#           mutate(loantype = )
```


Proportions of private and government loans over time
Private or Government:
G
P
density
1.00
0.75
0.50
0.25
0.00
2015−07
2016−01
2016−07

In Class Exercise:

Use var_string and str_split to calculate the average value of loans by loan type.

```
var_string <- "principal, loan_num"
```

3. Information is spread across multiple tables

- A common practice in data analysis is to combine multiple sources of information together
- Compared to most other common operations in R, there are few built-in safeguards when merging data.
  - As a result, it's extremely easy to make a mistake without realizing it. **Always review your output!**

Joining related data

- Base R: merge()
  - merge(object 1, object 2, how to join them, what to keep/add)
- Dplyr: the join family - ?join to view all
  - *_join(object 1, object 2, how to join them)

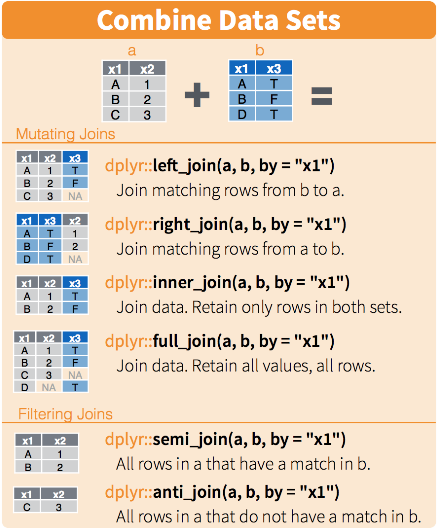
How to know which join statement to use

Mutate joins

- left_join is for augmenting an existing data frame - it always keeps all rows from the first object
  - right_join is similar
- full_join is for combining the entirety of two tables together (all rows and columns from both objects)

Filter joins

- inner_join filters the first object to rows with corresponding matches in the second object (could be considered a mutate join since it adds variables)
  - semi_join is similar except it does not add the variables from the second object
- anti_join only keeps records from the first object that did not get matched with the second object



Joining data by Tim Stuart

loanDescriptions dataset

- Structural table describing the attributes for a multitude of loans
  - If we are able to connect this data to the banks dataset we would then be able to expand our study

```r
head(loanDescriptions)
```

```
## # A tibble: 6 x 4
##   loanID length intAnnual refin
##    <chr>  <dbl>     <dbl> <lgl>
## 1     G1     30    0.1500  TRUE
## 2     G2     40    0.1900  TRUE
## 3     G3     20    0.1200  TRUE
## 4     P1      1    0.0005 FALSE
## 5     P2      3    0.0120 FALSE
## 6     P4     30    0.1750  TRUE
```

```r
left_join(banks, loanDescriptions, by = "loanID") %>%
  dim()
```

```
## [1] 2000   12
```

```r
full_join(banks, loanDescriptions, by = "loanID") %>%
  dim()
```

```
## [1] 2001   12
```

```r
inner_join(banks, loanDescriptions, by = "loanID") %>%
  dim()
```

```
## [1] 1789   12
```

- Explain the differences

In Class Exercise:

Using banks and loanDescriptions complete the following:

- Calculate the aggregate revenue on loans for banks within each of the charter types
- Calculate the proportions of the charter revenue held by each bank of that type
- Return the largest contributor to revenue in each district
  - By bank
  - By loan type

4. Information in a table is difficult to extract

- We will be using tidyr's two main functions: spread() and gather().
  - The former helps you deal with rows that are not complete observations and the latter with columns that are not variables.
- To demonstrate these methods of tidying data, we will be using the two datasets stocks_l and stocks_w.

```r
head(stocks_l)
```

```
##   Stock Year Price
## 1  AAPL 2007   400
## 2  AAPL 2008   450
## 3  AAPL 2009   500
## 4  AMZN 2007   200
## 5  AMZN 2008   150
## 6  AMZN 2009   200
```

```r
head(stocks_w)
```

```
##   Stock 2007 2008 2009
## 1  AAPL  400  450  500
## 2  ADBE   30   10   40
## 3  AMZN  200  150  200
```

The same exact information is stored in these two datasets. The "form" that stocks_l takes is often referred to as long-form data and stocks_w wide-form data. Advantages exist for both forms:

- Long
  - Conceptually clear - you can quickly recognize what constitutes a unique observation in the data.
  - Great for grouping - if you want color, shape, fill, etc. dimensions in your ggplots this is the form you want.
- Wide
  - Extremely common way to store data - the people storing data are typically those that enter data. Wide form usually makes entering data easier.

We can go from one to the other fairly easily in R thanks to packages like tidyr (spread/gather) and reshape2 (dcast/melt).

```r
library(tidyr)

stocks_w %>%
  gather(`2007`, `2008`, `2009`,
         key = "Year", value = "Price") %>%
  head()
```

```
##   Stock Year Price
## 1  AAPL 2007   400
## 2  ADBE 2007    30
## 3  AMZN 2007   200
## 4  AAPL 2008   450
## 5  ADBE 2008    10
## 6  AMZN 2008   150
```

Besides some rearrangement of rows, we put stocks_w through a gather() function and got an output exactly like stocks_l.

Think of spread() and gather() as inverses: one goes wide-to-long and the other long-to-wide (generally). Here we will spread stocks_l, although typically there is little reason to spread a tidy dataset like stocks_l.

```
stocks_l %>%
  spread(key = Year, value = Price)
```

```
##   Stock 2007 2008 2009
## 1  AAPL  400  450  500
## 2  ADBE   30   10   40
## 3  AMZN  200  150  200
```