Chapter 4: Readying the Data

1 Concepts

1.1 Text is data

Speeches contains a wealth of information. As humans, we are taught to understand verbal and written communication – pick out the nouns, verbs, and adjectives, then combine the information to decipher meaing. Take the following excerpt from the 2010 State of the Union:

> Now, one place to start is serious financial reform. Look, I am not interested in punishing banks. I'm interested in protecting our economy. A strong, healthy financial market makes it possible for businesses to access credit and create new jobs. It channels the savings of families into investments that raise incomes. But that can only happen if we guard against the same recklessness that nearly brought down our entire economy. We need to make sure consumers and middle-class families have the information they need to make financial decisions. We can't allow financial institutions, including those that take your deposits, to take risks that threaten the whole economy.

To many, text might not be considered data despite the fact that any analytical mind with a command of the English language can identify key terms:

> ~~Now, one place to start is serious~~ financial reform. ~~Look, I am not interested in~~ punishing banks. ~~I'm interested in~~ protecting our economy. ~~A~~ strong, healthy financial market ~~makes it possible for~~ businesses ~~to access~~ credit ~~and~~ create new jobs. ~~It channels the~~ savings of families ~~into~~ investments ~~that~~ raise incomes. ~~But that can only happen if we guard against the same~~ recklessness ~~that nearly brought down our entire~~ economy. ~~We need to make sure~~ consumers ~~and~~ middle-class families ~~have the information they need to make~~financial decisions. ~~We can't allow~~ financial institutions, ~~including those that take your~~ deposits,~~to take risks that threaten the whole~~ economy.

Much like the logic that guides keyword identification, text can be shaped from an unstructured dataset into a well-defined, structured dataset:

Table 1: Most frequent terms found in excerpt.

Terms	Frequency of Term	Number of Characters
financial	4	9
economy	3	7
families	2	8
interested	2	10

Of course, this process could be done manually, but imagine sorting through all 7,304 words in the 2010 address or scaling the process to the roughly *1.9 million words* in addresses State of the Union addresses between 1790 and 2016. All the steps required to convert unstructured text into usable data can be done with a little bit of planning, technical imagination and data manipulation. Every little detail about the data needs to be considered and meticulously converted into a usable form. From a data format perspective, capitalized characters are not the same as lower case. Contractions are not the same as terms that are spelled out. Punctuation affect spacing. Carriage returns and new line markers, while not visible in reading mode, are recorded.

Let's take one line from above and dissect the changes that need to be made:

> "We need to make sure consumers and middle-class families have the information they need to make financial decisions. We can't allow financial institutions, including those that take your

deposits, to take risks that threaten the whole economy."

We then turn everything into lower case so all letters of the alphabet are read the same.

> "we need to make sure consumers and middle-class families have the information they need to make financial decisions. we can't allow financial institutions, including those that take your deposits, to take risks that threaten the whole economy."

Then, we get rid of punctuation by substituting values with empty quotations (`""`).

> "we need to make sure consumers and middleclass families have the information they need to make financial decisions we cant allow financial institutions including those that take your deposits to take risks that threaten the whole economy"

Each space between each word can be used as a *delimiter* that can be used as a symbol for a program to break apart words into elements in a list.

Table 2: Terms

we	families	financial	those	that
need	have	decisions	that	threaten
to	the	we	take	the
make	information	cant	your	whole
sure	they	allow	deposits	economy
consumers	need	financial	to	
and	to	institutions	take	
middleclass	make	including	risks	

There are words in there that don't add much value as they are commonplace and filler. In text processing, these words are known as *stop words*. In each domain, the list of stop words likely differs, thus data scientists may need to build a customized list. For simplicity, we've used a stop words list that is used in the mySQL – an open source relational database management system. The result is the list of remaining words.

Table 3: Terms after removing stop words

make	information	financial	risks
consumers	make	institutions	threaten
middleclass	financial	including	economy
families	decisions	deposits	

From that data, we can aggregate the data into a form that is meaningful to answer a research question. For example, the frequency of words may provide a clue as to what the text is about. In this case, each "financial" and "make" appear twice in the text, perhaps indicating that there is an orientation towards action (make) for financial considerations.

Table 4: Term Frequencies

Term	Freq	Term	Freq
financial	2	including	1
make	2	information	1
consumers	1	institutions	1
decisions	1	middleclass	1
deposits	1	risks	1
economy	1	threaten	1
families	1		

This is just the tip of the iceberg. Text processing is just one aspect of readying data for use

Tidy Data. The ultimate goal of data retrieval and processing is to construct a data set that is ready for analysis, modeling and visualization.Tthis process, however, can occupy between 50% to 80% of a data scientist's time [(@lohr2014)]. Data needs to be processed into different forms depending on the use case. The number of tools available to *wrangle* data are many. But what if a standard framework could be applied to reduce the time burden and make data more actionable sooner. For one thing, we could follow the principles of *tidy data* laid out in @wickham2014:

- Each variable forms a column.
- Each observation forms a row.
- Each type of observational unit forms a table.

A *messy* data set is one that does not follow the previously described structure. For example, a hypothetical company's financial data may contain each month's data as columns with each row representing a different financial concept. Working with this data in the current form to produce a time series forecast would prove to be a challenge. The data should be a multivariate time series, thus the ideal is to store months as rows and each financial concept as a column.

Table 5: Example of a messy data set.

value	Dec.1.2017	Jan.1.2018	Feb.1.2018
revenue	46188	13009	8223
cost	32000	41277	1900
heads	73	36	37

Fortunately, this is a relatively simple task in this case – we simply need to transpose the data into tidy form. Once the data are in tidy form, the data can be more easily manipulated and used for data science purposes. The table below, for example, can not only be used for time series and cross sectional regression analysis, but visualized in a variety of ways. For the remainder of this book, we will tend towards tidy data and use them as the fundamental data form that will drive analyses.

Table 6: Example of a tidy data set.

date	revenue	cost	heads
12/1/17	46188	32000	73
1/1/18	13009	41277	36
2/1/18	8223	1900	37

We should point out that a suite of packages such as `tidyr` that make tidying data simpler and there are entire texts dedicated to data engineering and manipulating data. This chapter is intended to be a primer, providing a brief review of programming paradigms that are necessary to shape data into usable form so it can be the engine that drives impact in the public and social sectors. The chapter begins with illustrating a workflow for retrieving and assembling data, then explores common methods of manipulating data values and formats, then closes with manipulating data structures.

2 Retrieval and Assembly

While we briefly covered loading of data in the previous chapter, the retrieval and assembly of a data set can be one of the largest barriers to getting a project off the ground. Here, we lay out a few practices that make the process far simpler.

There are a multitude of data storage formats in use. Fortunately, R is equipped to load virtually all data formats. Below is a recommended set of functions that are easy to use and flexible.

Table 7: Recommended functions to load an assortment of files.

Function	Package	Description
read_excel	readxl	Load Excel files into a data frame. Note that with excel files, generally need to specify which sheet to load using the *sheet* argument and may need to skip a number of rows at the beginning of worksheet to indicate the header row using *skip*.
read_csv	readr	Load any comma separated file into data frame format.
read_delim	readr	Load delimited file using any delimiter such as tab delimited ("\") using the *sep* argument.
readLines	base R	Read a text file, line by line. This is helpful for working with free form text.
xmlToDataFrame	XML	Read XML into a data frame if the structure is simple and flat. Note that XML files tend to have a more complex hierarchical structure.
xmlToList	XML	If the XML is complicated, read each element as a list.
fromJSON	rjson	Read JSON into list. If JSON is a flat, non-hierarchical file, then the resulting object can be simply converted into a data frame.
read_dta	haven	Load Stata data file.
read_sas	haven	Load SAS data files.
read_spss	haven	Load SPSS data files.
load	base R	Load saved R data set that can contain multiple objects. Note that this does not need to be assigned to another object.
readRDS	base R	Read individually saved R object.

In the course of writing this text, the authors have come across a number of misconceptions about loading data that introduce manual editing to data files. In particular, we have found that:

- Analysts will often manually open files and delete unnecessary rows at the top of a file as functions may not be able to read in files.
- The number of columns will differ, requiring columns to be deleted or empty columns added before loading.
- Column names differ, requiring manual editing of headers.

These manual steps become tedious step when presented with hundreds of files. In this section, we illustrate a simple efficient workflow that reduces the amount of effort required to load data. Through this workflow, we read in gasoline spot price data from the US Energy Information Administration (EIA): two worksheets in an Excel file (gas prices from NY Harbor) and a CSV (Gas Prices from the US Gulf Coast). The goal is to combine the data into one data frame with daily spot prices, then calculate the correlation between the two sets of spot prices as shown below.

Table 8: Target format

date	price1	price2
2016-12-28	1.739	1.702
2016-12-29	1.729	1.703
2016-12-30	1.722	1.699

Loading CSVs. Before loading any file, it is generally a good idea to open the file in a code editor to take a peek at the structure. Otherwise, blindly loading can also give a clue as well. Let's start with the CSV `doe_usgulf.csv` using the `readr` package, then inspect the first few rows.

```
library(readr)
gulf <- read_csv("data/doe_usgulf.csv")
head(gulf, 3)
```

```
## # A tibble: 3 x 2
##   Date   `U.S. Gulf Coast Conventional Gasoline Regular Spot Price FOB (D~
##   <chr>                                                              <dbl>
## 1 1/2/14                                                              2.52
## 2 1/3/14                                                              2.49
## 3 1/6/14                                                              2.52
```

It appears that the data was successfully loaded, but requires cosmetic changes to the column headers. To practice good data hygiene, we would keep column names lower case and replace all spaces and punctuation with periods.

```
colnames(gulf) <- tolower(gsub("[[:space:][:punct:]]", ".", colnames(gulf)))
```

However, as these column names are quite long, we can otherwise opt to overwrite them with something more concise. Notice that we did not need to open the CSV and manually edit the column names.

```
colnames(gulf) <- c("date", "gulf.price")
```

Loading an Excel file. Excel files tend to have a few more curve balls than CSVs as there are often rows included to enhance readability, but making parsing more challenging. To take a first look at the first sheet of our Excel file `doe_ny.xlsx`, we need to specify `sheet = 1`, then we use `head` to examine what was loaded. We can see that the wrong row was assumed to be the header.

```
library(readxl)
attempt1 <- read_excel("data/doe_ny.xlsx", sheet = 1)
head(attempt1, 3)
```

```
## # A tibble: 3 x 4
##   Sourcekey EER_EPMRU_PF4_Y35NY_DPG                          X__1  X__2
##   <chr>     <chr>                                            <lgl> <lgl>
## 1 Date      New York Harbor Conventional Gasoline Regular Spo~ NA    NA
## 2 41641     2.718                                            NA    NA
## 3 41642     2.6709999999999998                               NA    NA
```

Fortunately, we can skip rows when using most loading functions. In this case, we only need to skip one row using the `skip` argument. Below, we read in the first and second sheets of `doe_ny.xlsx`, then rename the headers

```
#Load first two sheets
sheet1 <- read_excel("data/doe_ny.xlsx", sheet = 1, skip = 1)
sheet2 <- read_excel("data/doe_ny.xlsx", sheet = 2, skip = 1)
```

```
#Rename columns
colnames(sheet1) <- c("date", "ny.price")
colnames(sheet2) <- c("date", "ny.price")
```

Appending data. It turns out that *sheet1* and *sheet2* are from the same time series of spot prices. We can construct a new, more complete data frame by appending one data frame to the other using the `rbind` function.

```
  ny <- rbind(sheet1, sheet2)
```

The above situation is the ideal: two data frames with the same number of columns and the same column names. But what if one data frame has at least one more column than the other? The `rbind.fill` function in the `plyr` package appends the two data frames together, filling any additional missing columns with `NA` values. Below, rather than loading the entire `plyr` library, we can selectively use the `rbind.fill` using the double colon operator `::`.

```
  ny <- plyr::rbind.fill(sheet1, sheet2)
```

Combining data by rows. The two data frames `gulf` and `ny` contain the same number of rows. If they are in the same exact desired order and have the same number of rows, we can use the `cbind` function to join the datasets by rows. The `cbind` function will join all columns in both data frames together, including the dates. To ensure the data is neatly organized, we keep all columns from the `gulf` dataset and only the second column from the `ny` data set. The resulting data frame contains three columns containing daily spot prices from two sources.

To calculate the correlation between the two sets of prices, we use the `cor` function finding that $\rho = 0.989617$ – the two sets of prices are highly correlated. Note that `cor` only accepts numeric values, thus the date column is temporarily dropped.

```
#Combine data
  prices <- cbind(gulf, ny[,2])

#Calculate correlation
  cor(prices[,-1])
```

```
##            gulf.price  ny[, 2]
## gulf.price   1.000000 0.989617
## ny[, 2]      0.989617 1.000000
```

3 Manipulating values

3.1 Strings

More often than not, data cleansing involves finding, extracting, and replacing the contents of string values. For example, below is a vector of four string values:

```
budget <- c("Captain's Log, Stardate 1511.8. I have $10.20 for a big galactic mac.",
            "The ensign has $1,20 in her pocket.",
            "The ExO spent has $0.25 left after paying for overpriced warp core fuel.",
            "Chief medical officer is the high roller with $53,13.")
```

What if we need to extract the total available funds available to buy galactic big macs? All four elements contain dollar values, which can benefit from feature engineering. To do so, we use a combination of text manipulation functions and *regular expressions* or *regex* – a series of characters that describe a regularly occurring text pattern.

First, commas should be replaced with a period using `gsub()`, assigning the result to a new object `new`. Note that in some regions, such as Europe, commas are used as decimals rather than periods.

```
new <- gsub(",", "\\.", budget)
```

Second, find the elements that contain the following pattern: a dollar sign followed by one to two digits, followed by a period, then another two digits (`\\$\\d{1,2}\\.\\d{2}`). The pattern can be used with the functions `regexpr()` to find the positions of the matching patterns in the text, then `regmatches()` is used to extract.

```
indices <- regexpr("\\$\\d{1,2}\\.\\d{2}", new)
numbers <- regmatches(new, indices)
print(numbers)
```

```
## [1] "$10.20" "$1.20"  "$0.25"  "$53.13"
```

Third, we should replace dollar sign with blank and strip out any leading white space using `trimws()`.

```
numbers <- trimws(gsub("\\$","", numbers))
print(numbers)
```

```
## [1] "10.20" "1.20"  "0.25"  "53.13"
```

Lastly, convert the character vector to numeric, then sum the vector.

```
money <- as.numeric(numbers)
print(paste0("Total galactic big mac funds = $", sum(money)))
```

```
## [1] "Total galactic big mac funds = $64.78"
```

A number of observations. In steps one through three, you will have noticed that the characters `"$"`, `"."`, and `"d"` were preceded by double backslash. These are known as *escaped characters* as the double backslash preceding the characters changes their meanings. In step two, a sequence of unusual characters (`\\$\\d{1,2}\\.\\d{2}`) was used to find the `$x.xx` pattern, which can be broken into specific commands:

- `\\$` is a dollar sign.
- `\\d{1,2}` is a series of numerical characters that is between one to two digits long.
- `\\.` is a period.
- `\\d{2}` is a series of numerical characters that is exactly two digits long.

Mastering *regex* is a productivity multiplier, opening the possibility of ultra-precise text replacement, extraction, and other manipulation. Imagine scenarios where raw data is not quality controlled and mass errors plague the usefulness of the data. An analyst may spend days if not weeks or months cleaning data by hand (or rather through find and replace). With regex, haphazard cleaning is no longer an issue. To make the most of regex requires a command of both *text manipulation functions* that are designed to interpret regex as well as *regex* itself.

Text manipulation functions

Find and replace are useful functions in most word processing and spreadsheet softwares. But what does it take to do find and replace at scale. The following seven text manipulation functions are commonly implemented in programming languages. Each searches for a user-defined pattern and returns a result in a well-defined format.

Table 9: Recommended text manipulation functions

Description	Base R	stringr
Returns either the index position of a matched string or the string containing the matched portion.	grep	
Returns a logical vector indicating if a matched string was found.	grepl	str_detect
Searches and replaces patterns in strings.	gsub	
Returns the character position of a pattern in a string.	regexpr	
Extract substring using positions	regmatches	
Splits strings into a list of values based on a delimiter.	strsplit	
Extract substring from a string based on string positions.	substr	str_extract
Trim whitespace (excess spaces)	trimws	str_trim

Traditionally, functions like `grep()` are available through command line interfaces and are a core offering of the R programming language. On their own, some basic tasks can be accomplished such as exact matches of specific text. As will be seen later, these functions combined with regex are quite powerful. To illustrate the basic functionality, let's assume we have four sentences that indicate when four US laws were signed.

```
laws <- c(". Dodd-Frank Act was signed into federal law on July 21, 2010.",
    "Glass-Steagall Act was signed into federal law by FDR on June 16, 1933",
    "Hatch Act went into effect on August 2, 1939",
    "Sarbanes-Oxley Act was signed into law on July 30, 2002")
```

Suppose we need to find acts that are named for two congressmen. The `grep()` function can be used to find the index positions of elements in a vector that contain `"-"`. Otherwise stated, return the row number for each sentence that contains a hyphen. In this case, the 1st, 2nd, and 4th elements in the `laws` vector contain hyphens.

```
grep("-", laws)
```

```
## [1] 1 2 4
```

`grep()` can also return the matched value when the option `value` is set to `TRUE`. This is handy for inspecting the accuracy of matches. In practice, with large data sets that contain variable names that follow a common convention, column names can be efficiently searched.

```
  grep("-", laws, value = TRUE)
```

```
## [1] ". Dodd-Frank Act was signed into federal law on July 21, 2010."
## [2] "Glass-Steagall Act was signed into federal law by FDR on June 16, 1933"
## [3] "Sarbanes-Oxley Act was signed into law on July 30, 2002"
```

This can also be expressed in a different way. The `grepl()` function can be used to obtain a vector of logical values (`TRUE`/`FALSE`) that is the same length as the input vector `laws`.

```
  grepl("-", laws)
```

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

Regular Expressions

Regular Expressions (regex) are powerful commands that give coders the flexibility to search data and surface results following a pattern. Before proceeding into more complex string combinations, knowledge of a few cleverly designed capabilities may go a long way:

(1) Alternatives (e.g. "OR" searches) can be surfaced by using a pipe "`|`". For example, a string search for "Bob or Moe" would be represented as "Bob|Moe".

(2) The extent of a search should be denoted by parentheses `()`. For example, a string search for "Jenny" or an alternative spelling like Jenny would be represented as "Jenn(y|i)"."

(3) A search for one specific character should be placed between square brackets `[]`.

(4) The number of characters is placed between curly brackets `{}`.

In New York City, the famed avenue *Broadway* is may be written and abbreciated in a number of ways. The vector `streets` contains a few instances of spellings of Broadway mixed in with other streets that start with the letter `B`.

```
#A sampling of street names
  streets <- c("Bruckner Blvd", "Bowery", "Broadway", "Bway", "Bdway",
        "Broad Street", "Bridge Street", "B'way")

#Search for two specific options
  grep("Broadway|Bdway", streets, value = TRUE)
```

```
## [1] "Broadway" "Bdway"
```

```
#Search for two variations of Broadway
  grep("B(road|')way", streets, value = TRUE)
```

```
## [1] "Broadway" "B'way"
```

```
#Search for cases where either d or apostrophe are between B and way
  grep("B[d']way", streets, value = TRUE)
```

```
## [1] "Bdway" "B'way"
```

3.1.0.1 Escaped characters

Quite a few single characters hold a special meaning in addition to the literal meaning. To disambiguate their meaning, a backslash precedes these characters to denote the alternative meaning. A few include:

- `\n`: new line
- `\r`: carriage return
- `\t`: tab
- `\'`: single quote when in a string enclosed in single quotes (`'Nay, I can\'t'`)
- `\"`: double quote when in a string enclosed in double quotes (`"I have a \"guy\"."`)

In other cases, double backslashes should be used:

- `\\.`: period. Otherwise, un-escaped periods indicate searches for *any* single character.
- `\\$`: dollar sign. A dollar sign without backslashes indicates to find patterns at the end of a string.

3.1.0.2 Character Classes

A *character class* or *character set* is used to identify specific characters within a string. How would one represent "12.301.1034" or "?!?!?!"? One or more of the following character classes can do the job:

- `[:punct:]`: Any and all punctuation such as periods, commas, semicolons, etc. For specific specific punctuation, simply enclose the characters between two brackets. For example, to find only commas and carrots, use `[<>,]`.
- `[:alpha:]`: Alphabetic characters such as a, b, c, etc. With other languages including R, it is commonly written as `[a-z]` for lower case and `[A-Z]` for upper case.
- `[:digit:]`: Numerical values. With other languages including R, it is commonly written as `\\d` or `[0-9]`. For any non-digit, write `\\D`.
- `[:alnum:]`: Alphanumeric characters (mix of letters and numbers). With other languages including R, it is indicated using to as `[0-9A-Za-z]` or `\\w`. For any non-alphanumeric character, use `\\W`.
- `[:space:]`: Spaces such as tabs, carriage returns, etc. For any white space, use `\\s`. For any non-whitespace character, use `\\S`.
- `[:graph:]`: Human readable characters including `[:alnum:]` and `[:punct:]`.
- `\\b`: Used to denote "whole words". `\\b` should be placed before and after a regex pattern. For example, `\\b\\w{10}\\b` indicates a 10 letter word.

There are quite a few character classes not listed above, but for these constitute the lion's share. It is worth keeping in mind that the implementation of character classes may differ between programming languages. A number of the above are extensions that have been implemented in R in a specific manner.

3.1.0.3 Quantifiers

Each character class on its own indicates a search for *one and only one character*. In practice, most character searches will involve a search for more than just one character. To indicate such a search, regex relies on *quantifiers* to indicate the length of patterns. For example, a search for a year between the year 1980 and 2000 will require exactly four digits, but a search for the speed of a gust of wind will likely vary between 1 and 3 digits. The following six quantifiers provide a degree of both flexibility and specificity to accomplish search tasks:

- `{n}`: match pattern n times for a preceding character class. For example `"\\d{4}"` looks for a four digit number.
- `{n, m}`: match pattern at least n-times and not more than m times for a preceding character class. For example `"\\d{1,4}"` looks for one to four digit number.
- `{n, }`: match at least n times for a preceding character class. For example `"\\d{4,}"` looks for a number that has at least four digits.

- `*`: Wildcard, or match at least 0 times.
- `+`: Match at least once.
- `?`: Match at most once.

In the example below, quantifiers are used to extract specific number patterns with a high degree of accuracy.

```
  dates <- c("Octavian became Augustus on 16 Jan 27 BCE",
             "In the year 2000, a computer bug was expected to topple society.",
             "In the 5400000000 years, our sun will become a red dwarf.")

#Match an element with a 9 digit number
  grep("\\d{9}", dates, value = TRUE)
```

```
## [1] "In the 5400000000 years, our sun will become a red dwarf."
```

```
#Match an element with a 9 digit number
  grep("\\b\\d{4}\\b", dates, value = TRUE)
```

```
## [1] "In the year 2000, a computer bug was expected to topple society."
```

```
#Match a date that follows 16 January 27 BCE
  grep("\\d{2}\\s\\w{3}\\s\\d{2}\\s\\w{3}", dates, value = TRUE)
```

```
## [1] "Octavian became Augustus on 16 Jan 27 BCE"
```

3.1.0.4 Position matching

Regex builds in functionality to search for patterns based on position of a substring in a string, such as at the start or end of a string. There are quite a few other position matching patterns, but the following two are the workhorses.

- `$`: Search at the end of a string.
- `^`: Start of string when placed at the beginning of a regex pattern.

To demonstrate these patterns, we'll apply `grep()` to three headlines from the BBC.

```
  headlines <- c("May to deliver speech on Brexit",
           "Pound falls with May's comments",
           "May: Brexit plans to be laid out in new year")
  print(headlines)
```

```
## [1] "May to deliver speech on Brexit"
## [2] "Pound falls with May's comments"
## [3] "May: Brexit plans to be laid out in new year"
```

```
#Find elements that contain May at the beginning of the string
  grep("^May", headlines, value = TRUE)
```

```
## [1] "May to deliver speech on Brexit"
## [2] "May: Brexit plans to be laid out in new year"
```

```
#Find elements that contain Brexit at the beginning of the string
  grep("Brexit$", headlines, value = TRUE)
```

```
## [1] "May to deliver speech on Brexit"
```

3.1.1 DIY: Redact

In an increasingly digital world, data privacy is a sensitive issue that has taken center stage. At the center of it is the safeguarding of Personally identifiable information (PII). Legislation in the European Union, namely the General Data Protection Regulation or GDPR, requires companies to protect the personal data of European Union (EU) citizens associated with transactions conducted in the EU [(@gdpr)]. The US Census Bureau, which administers the decennial census, must apply disclosure avoidance practices in order so that individuals cannot be identified [(@censusavoidance)]. Anonymization has become a common task when working with sensitive PII data, spanning complex probabilistic methods to simple redaction. We focus here on the latter.

The following sentence, for example, contains hypothetical PII and sensiive information – John's social security number and balance in his savings account are shown. When presented with many lines of sensitive information, one could review each sentence and manually redact sensitive information, but given thousands if not millions of sensitive information, this is simply not feasible.

```
  statement <- "John Doe (SSN: 012-34-5678) has $2303 in savings in his bank account."
```

Using a combination of regex and `stringr` we can redact sensitive information with placeholders. To remove the SSN, we need a regex pattern that captures a pattern with three digits (`\\d{3}`), a hyphen, two digits (`\\d{2}`), a hyphen, then four digits (`\\d{4}`), or when combined: `\\d{3}-\\d{2}-\\d{4}`. The matched pattern is then replaced with `XXXXX`.

```
  library(stringr)
```

```
## Warning: package 'stringr' was built under R version 3.4.4
```

```
  new_statement <- str_replace(statement,"\\d{3}-\\d{2}-\\d{4}", "XXXXX")
  print(new_statement)
```

```
## [1] "John Doe (SSN: XXXXX) has $2303 in savings in his bank account."
```

Next, we replace the dollar value by matching a string that starts with the dollar sign (`\\$`) followed by at least one digit (`\\d{1,}`). And finally, the John Doe's first and last name are replaced by looking for two substrings that each have at least one uppercase letter with an unspecified length (`[A-z]{1,} [A-z]{1,}`) and are found at the beginning of the string (`^`). The resulting sentence has little to no information about the individual in question.

```
#Find a replace dollar amount
  new_statement <- str_replace(new_statement,"\\$(\\d{1,})", "XXXXX")

#Find and replace first and last name
  new_statement <- str_replace(new_statement,"^[A-z]{1,} [A-z]{1,}", "XXXXX")

  print(new_statement)
```

```
## [1] "XXXXX (SSN: XXXXX) has XXXXX in savings in his bank account."
```

3.2 Stringr

Trim whitespace str_subset: Replace strings str_detect: Create binary variable from text: str_count: Number of matches str_replace: Replace matching substrings

3.3 Working with Dates

Working dates can be challenging. Most programming languages do not automatically recognize a date variable, requiring coders to specify the format using a set of date symbols. The `lubridate` package significantly lowers the bar for R users, making the process of working and manipulating dates more seamless and less prone to error.

```
require(lubridate)
```

Upon loading lubridate, we can convert string and numeric objects with values that represent dates into date objects. The `as_date` function is most intuitive, automatically detecting the date format and assumes that the date is recorded in UTC.

```
d0 <- as_date("2010-08-20")
```

In case `as_date` is unable to detect the date, the user can more formally define the date format using functions such as `mdy` and `ymd`. Both are able to accommodate both string and numeric formats.

```
d1 <- mdy("01/20/2010")
d2 <- mdy_hm("01/20/2010 00:00 AM")
d3 <- ymd(20100101)
```

Once data are converted into date objects, it is easy to process derivative information. To calculate duration between two dates is as simple as subtraction.

```
d1 - d3
```

```
## Time difference of 19 days
```

Often times will need to extract parts of the date. With lubridate, each `year`, `month`, `quarter`, `day`, `hour`, `minute`, and `second` can be extracted using a dedicated function. For example, we can extract the `year` and `quarter` and concatenate into a string using `paste0`.

```
  y1 <- year(d0)
  q1 <- quarter(d0)
  paste0(y1, "Q", q1)
```

```
## [1] "2010Q3"
```

In base R, we can alternatively use the `format` function to specify an output format using date symbols. The example below outputs a concatenation of year and month.

```
  format(d0, "%Y-%m")
```

```
## [1] "2010-08"
```

4 Manipulating structures

4.1 Missing values

Missing values: is.na Complete observations only: complete.cases

`dplyr` Extracting rows, columns, and specific elements from a data frame Basic R: Index notation, etc.

4.2 Basic math

Arithmetic operations: Sum, mean, etc.

Operation	General	Row-Wise	Column-wise
Sum	`sum`	`rowSums`	`colSums`
Mean	`mean`	`rowMeans`	`colMeans`
Minimum	`min`		
Maximum	`max`		

Distance measures: dist() - L1/L2 String distance: adist() - Levenshtein, stringdist() - soundex Similarity measures: cor(), `coop` library

4.3 Control Structures

Much of data science requires developing specialized code to handle the eccentricities of a dataset. Re-running blocks of code is required, often times on multiple data samples and subpopulations. It's simply not scalable to manually change variables and assumptions of the code everytime.

Variables are typically treated differently based on their quality and characteristics. In order to accomplish analytical and programming tasks, control structures are used to determine how a program will treat a given variable given conditions and parameters. In this section, we will cover two commonly used control structures: if...else statements and for loops.

4.3.1 If and If...Else Statement

If statements evaluate a logical statement, then execute a script based on whether the evaluated statement is true or false. If the statement is `TRUE`, then the code block is executed.

```
budget <- 450
if(budget > 400){
  #If statement true, run script goes here
  print("You're over budget. Cut back.")
}
```

```
## [1] "You're over budget. Cut back."
```

In cases where there are two or more choices, if...else statements would be appropriate. In addition to the `if()` statement, an `else` statement is included to handle cases where the logical statement is `FALSE`.

```
budget <- 399
if(budget >= 400){
  #If statement true, run script goes here
  print("You're over budget. Cut back.")
} else {
  #else, run script goes here
  print("You're under budget, but watch it.")
}
```

The complexity of these statements can be as simple as `if(x > 10){ print("Hello")}` more complex trees:

```
age <- 23

if(age <= 12){
```

```
    print("kid")
  } else if(age >12 && age <20) {
    print("teenager")
  } else if(age >=20 && age <65) {
    print("adult")
  } else{
    print("senior")
  }
```

```
## [1] "adult"
```

4.3.2 For-loops

Loops can be used to run the a given statement of code multiple times for a specified number of times or a list of index value. This is a functionality that is available in most programming languages, but the programming syntax will be different. Conceptually, for loops can be likened to an assembly line in a car factory. In order to build a car, a series of well-defined, well-timed processes need to coordinated in a serial fashion. To build 500 cars, the process needs to be executed 500 times. For-loops are essentially the same: Given a well-defined, self-contained process, a process can be be iterativelyapplied to address repetive tasks.

Let's take the following example. The code block essentially says "print values for the range of 1 through 5", where `i` is an *index value*. When executing the statement, R will push the first value in the sequence of 1:5 into the index (in this case, it's the number 1), then the code block in between the `{}` (curly brackets) will be executed, treating `i` as if it's the number 1. Upon executing the code without error, R will advance to the next value in the sequence and repeat the process until all values in the sequence have been completed.

```
for(i in 1:5){
  print(paste0("Car #", i))
}
```

```
## [1] "Car #1"
## [1] "Car #2"
## [1] "Car #3"
## [1] "Car #4"
## [1] "Car #5"
```

We can do the same for a vector or list of values. In the example below, the vector `news` contains six terms. Using a for-loop, we can print out each word in the vector.

```
news <- c("The","Dow","Is","Up","By","400pts")
for(i in news){
  print(i)
}
```

```
## [1] "The"
## [1] "Dow"
## [1] "Is"
## [1] "Up"
## [1] "By"
## [1] "400pts"
```

For-loops has a few qualities that users should be aware. First, what happens within the for-loop is written to the R environment as *global variables*. That means that any object (e.g. calculations, models) that is created in the loop will be accessible in the programming enviromment even after the loop ends. This may be a good or bad, depending on the use case: Good if one wants to keep copies of the intermediate results of a loop iteration, but bad if the user is not careful to take note of the potential floor of extraneous objects

that may effect downstream calculations. Second, one of the most common mistakes when using loops is failing to record the result of the loop. There are functions in R that are designed to log and package results from loops, but in plain vanilla loops, this is not the case.

A common paradigm with for-loops is to iteratively execute repetitive tasks. For example, if a calculation needed to be applied to each of one million files and the results need to be logged, then for-loops are a good option. Typically, the paradigm proceeds as follows:

1. Create placeholder object (e.g. a vector, matrix, or data frame);
2. Initialize loop; and
3. Add outputs to placeholder at the end of each loop iteration.

This may be applied in a broad variety of cases such as processes each data set in a repository of many large data sets, calculating complex statistics for various strata and subsets within the data, among others. Best practices with loops start with initializing new placeholder objects to full length before the loop rather than increasing the object size within the loop[1]. In R, this is particularly important issue for efficient data processing.

In the example below, we would like to calculate the minimum and maximum of each of 1000 randomly generated normal distributions with $\mu = 1000$ and $\sigma = 10$. To do this, a placeholder data frame `x` with three columns (iteration, min and max) is created with $n = 1000$ rows for each of the random distributions to be generated. Then, we use `Sys.time()` to capture when the loop starts and end – a common practice for optimizing code. The loop is initiated for 1 to 1000 iterations to calculate the mininum and maximum. At the end of each iteration, the min and max results are overwritten to the row that corresponds to the iteration in the placeholder `x`.

```
#Set placeholder data frame with n rows
  n <- 1000
  x <- data.frame(iteration = 1:n,
                  min = numeric(n),
                  max = numeric(n))

#Loop
  start <- Sys.time()
  for(i in 1:n){
    y <- rnorm(10000, 1000, 10)
    x$min[i] <- min(y)
    x$max[i] <- max(y)
  }
  Sys.time() - start
```

```
## Time difference of 0.836514 secs
```

The above process required roughly 0.8 seconds to process. *What happens if the placeholder length were not pre-specified?* For the given parameters, the task normally may last between 1.2 and 1.5 seconds. This may not seem to be much time, but at scale with millions if not billions of records and iterations, the time does tend to add up.

```
#Set placeholder data frame without dimensions
  n <- 1000
  x <- data.frame()

#Loop
  start <- Sys.time()
  for(i in 1:n){
    set.seed(i)
```

[1] https://www.r-project.org/doc/Rnews/Rnews_2008-1.pdf

```
    y <- rnorm(10000, 1000, 10)
    x <- rbind(x, cbind(iteration = i,
                  min = min(y),
                  max = max(y)))
  }
  Sys.time() - start
```

```
## Time difference of 0.8838899 secs
```

4.3.2.1 R-specific: `apply`

For-loops are common across all languages, but the efficiency of their implementation will vary. As was described in the previous chapter, R is an interpretted language optimized for mathematical and statistical calculation – quite different than other languages. This means that programming in R is most optimal when vectorizing calculation – linear algebra calculations of vectors and matrices using operations such as `+`, `-`, `*`, `%*%`, among others.

In R, the speed of for-loops may be improved using `lapply()` under certain circumstances. `lapply()`, or *list apply* Whereas the intermediate objects in for-loops are global variables, `lapply()` creates temporary *local variables*.

```
#Set n
  n <- 1000

#Loop
  start <- Sys.time()
  x <- lapply(1:n, function(i){
     y <- rnorm(10000, 1000, 10)
     return(cbind(iteration = i,
                  min = min(y),
                  max = max(y)))
  })
  x <- do.call(rbind, x)
  Sys.time() - start
```

```
## Time difference of 0.6930389 secs
```

4.3.3 While

Whereas for loops require a range or list of values through which to iterate, `while()` statements keep iterating until some condition is met. The `while()` statement is formulated as follows:

```
  while([condition is true]){

    [execute this statement]

  }
```

A simple case may involve drawing a random value x from a normal distribution ($\mu = 1.0$, $\sigma = 0.5$) while x is greater than 0.01. 1

```
  x <- 1
  while(x > 0.01){
    x <- rnorm(1, 1, 0.5)
    print(x)
  }
```

```
## [1] 0.6867731
## [1] 1.091822
## [1] 0.5821857
## [1] 1.79764
## [1] 1.164754
## [1] 0.5897658
## [1] 1.243715
## [1] 1.369162
## [1] 1.287891
## [1] 0.8473058
## [1] 1.755891
## [1] 1.194922
## [1] 0.6893797
## [1] -0.1073499
```

```
  print("done!")
```

```
## [1] "done!"
```

4.4 Functions

Data manipulation tasks are often repeated for many different projects and it is not uncommon for two or more scripts to contain the same exact steps, but the code is hardcoded. Same logic and different variables names equates to a significant amount of time spent editing and modifying programs.

Rather than tediously modifying programs, try to write your code once, then never again. Each set of code can serve as re-usable tools that can be re-applied to similar problems, but only if it is standardized with well-laid logic. This is the basis of *user-defined functions*: a coder can define some set of standard required inputs on which a set of steps can be applied to produce a standard output.

A typical function is constructed as follows. Using `function`, a set of input parameters are specified as placeholders for any kind of object. For example, `df1` represents a data frame and `var1` is a variable name in string format. Within the curley brackets, we insert code treating the parameters of actual data. In the example below, we calculate the mean of `var1` in data frame `df1`, then assign it to a `temp.mean`. These calculations are executed in a *local environment*, meaning that any calculations steps within the function are temporary. Thus, `temp.mean` is wiped once the function finishes. The `temp.mean` object can be extracted by passing it to `return`. All of the above steps are assigned to the `meanToo` object that is treated like any other function.

It is good form to include commentary about how to use the function. At a minimum, there should be comments containing what the function is, the arguments, and the output. In the open source tradition, you should be writing the code as if others will read and use it.

```
meanToo <- function(df1, var1, ...){
  #
  # Calculate mean of a variable in a data frame
  #
  # Args:
  #   df1 = data frame
  #   var1 = variable name (character)
  #
  # Returns:
  #   A numeric value
```

```
  #Code goes here
  temp <- mean(df1[[var1]])

  #Return desired result
  return(temp)
}
```

To execute the function, we will simply need to call the function with a data frame and a variable name. Basically any script can be genericized into a standardized function.

```
  meanToo(data, "x1")
```

4.4.1 DIY: Tracking the supply of online content

Services like Google Trends illustrate the demand for online content, giving a sense of what the public are interested in and what people are reacting to at a given moment. But how about the supply of online content? The supply provides a sense if content producers feel it is worth spending any time on creating new materials.

The opioid epidemic has been a social issue that has long been brewing, but has only become the centered of public attention in recent years. As the crisis deepens, we'd expect more content to be generated. For each year, we could tediously use the web browser to manually copy the number of search results from a platform like Bing, but what if we need to track a large number of search terms. We can package searches on Bing into a simple function that serves as a wrapper that extracts the number of search results. For example, the search query below yields approximately 18,000 search results for the year 2010 (the number may change as this is only an estimate):

`https://www.bing.com/search?q=opioid%20epidemic&filters=ex1%3a%22ez5_12810_13893%22`

Breaking down the URL, we see that the substring `q=opioid%20epidemic` indicates that the search `q=` will be based on the terms that follow `q=` and spaces are encoded as `%20`. Next, the substring `filters=ex1%3a%22ez5_12810_13893%22` indicates a time range filter is applied, specifically `12810_13893` are indexes that represent the range 1/27/2005 to 1/15/2008.

To make this a seamless process, we should construct a function `bingCounts` that will retrieve the number of search results for a given `search.term` in a specified calendar year (`year.filter`). First, we will need to load packages using `require`, which is specifically designed for within-function package loading. In particular, we will use:

- To convert the year into indexes, we use the `lubridate` package to work with date objects.
- To efficiently construct a search URL, we rely on the `stringr` package.
- `RCurl` is used to make use of the Client URL package to make a request to the web for a webpage.
- As the webpage is in HTML format, we use the `XML` package to parse the information.

```
bingCounts <- function(search.term, year.filter){
  #
  # Retrieve number of search results for exact query
  #
  # Args:
  #   search.term = search query (character)
  #   year.term = search year.term (int)
  #
  # Returns:
  #   A numeric value

  #Load package
    require(lubridate)
```

```
  require(stringr)
  require(RCurl)
  require(XML)

#Get date indexes
  origin <- as_date("1970-01-01")
  start.index <- as_date(paste0(year.filter, "01-01")) - origin
  end.index <- as_date(paste0(year.filter, "12-31")) - origin

#Construct search URL by replacing placeholder terms
  search.url <- "https://www.bing.com/search?q=term&filters=ex1%3a%22ez5_tstart_tend%22"
  search.url <- str_replace(search.url, "term", gsub(" ", "%20", search.term))
  search.url <- str_replace_all(search.url, "tstart", as.character(start.index))
  search.url <- str_replace_all(search.url, "tend", as.character(end.index))

#Get URL content as HTML
  search.html <- getURL(search.url)

#Parse HTML
  parse.search <- htmlTreeParse(search.html, useInternalNodes = TRUE)

#Extract result statistics
  nodes <- getNodeSet(parse.search, "//*[@id='b_tween']/span[1]")
  value <- strsplit(xmlValue(nodes[[1]])," ", fixed = TRUE)[[1]][1]
  return(as.numeric(gsub(",", "", value, fixed = TRUE)))

}
```

Rather than copying the above function multiple times, the resulting function can then be neatly wrapped into a loop to extract the number of search results for each year in a nine-year period between 2009 and 2017. We find the number of search results has grown 44-times over the period. Based on CDC data, the number of opioid-related deaths increased 2.4-times from approximately 20,400 to 49,000.[2] Thus, we can see the search queries give a measure of direction of growth, but are misleading in their magnitude.

```
#Set parameters
  term <- "opioid epidemic"
  results <- data.frame()

#Loop through
  for(i in 2009:2017){
    results <- rbind(results,
                     data.frame(year = i,
                                cnt = bingCounts(term, i)))
  }
```

4.5 Style

Every coder approaches coding differently. Thus, each person's code is a digital equivalent of a fingerprint and has a unique touch. This is all the more reason why a style guide for coding is useful for not only producing functional code, but readable code.

[2] https://www.drugabuse.gov/related-topics/trends-statistics/overdose-death-rates

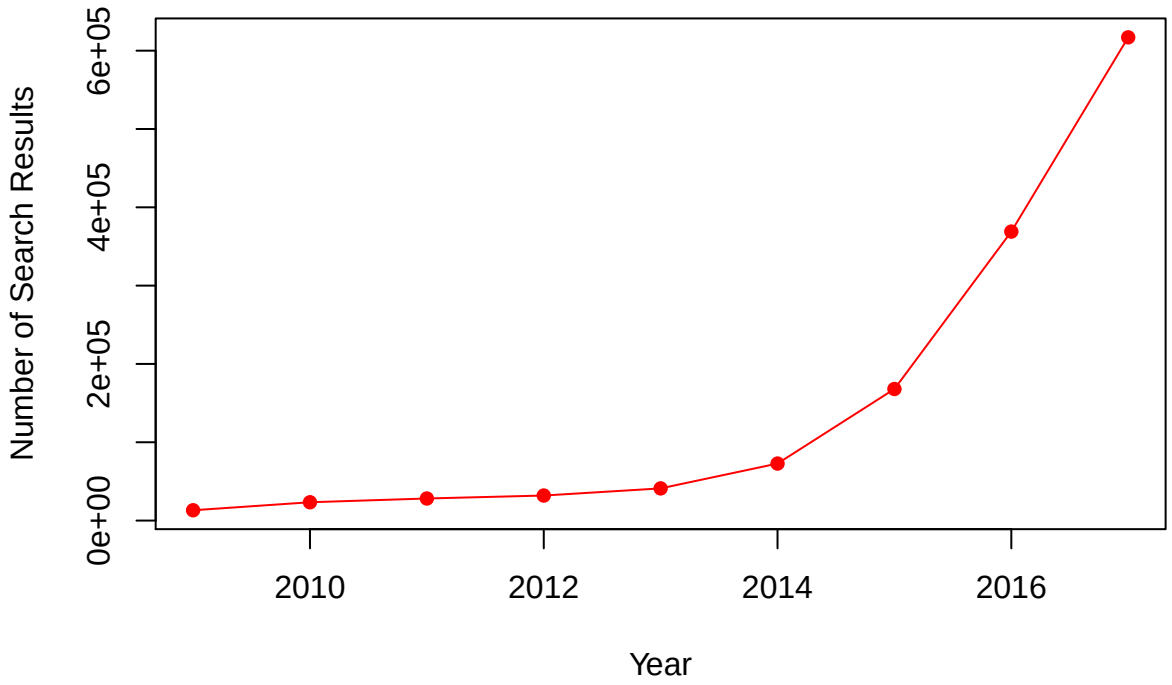

Figure 1: Number of search result by year

5 Feature engineering

Define: Feature engineering is the process of transforming data to create new variables that provide potentially more insight about the phenomenon of interest. deeper insight. As we'll see later in the book, feature engineering is critical for optimizing the accuracy of machine learning applications – the models are only as good as the available data

Feature engineering typically depend on domain knowledge to construct new variables. Examples: Continuous values can be summarized as counts, minimum, maximum, average, percentile among others.

Example: Prevailing weather conditions – give a benchmark of what has happened

Continuous values can also be interacted with one another to mix the signal.

Example: Spending per capita

In Text data, create a count or binary variable for each keyword Example: "One hot encoding" for keywords

Among discrete variables, combine sparse classes into aggregate classes that are more meaningful.
Example: For example, NYC 311 service has 1,700+ complaint + descriptor combinations

- Flavors of feature engineering. Feature engineering is important but expensive due to the time cost Automated methods like FeatureTools (https://www.featuretools.com/) can construct a large number of new variables for prediction problems.
- DIY: Augment a US Senate Roll Call
- Goal is to produce additional features that are correlated with the likely outcome of each vote.
- Get data from senate site – 101st to 115th for sessions 1 and 2 https://www.senate.gov/legislative/LIS/roll_call_lists/vote_menu_101_1.xml https://www.senate.gov/legislative/LIS/roll_call_lists/vote_menu_101_2.xml
- Clean up dates

- For each record, we'll calculate a number of additional variables on a 90 day moving window
- Extract a matrix of key words
- We can see that a simple data set can give way to thousands of new variables