Big Data and Economics

Tidy text activity

Kyle Coombs (adapted from Joshua C. Fjelstul)

Bates College | DCS/ECON 368

Contents

The following activity is adapted from work by Joshua C. Fjelstul and Julia Silge and David Robinson.

Software requirements

R packages

Here's what we'll be starting with to do some text tricks today.

- New: **tidytext**, **fuzzyjoin**, **wordcloud**, **RColorBrewer**
- Already used: **tidyverse**, **gsheet**

```
knitr::opts_chunk$set(echo = TRUE, cache = TRUE, dpi=300)
if (!require(pacman)) install.packages(pacman)
pacman::p_load(tidyverse,tidytext,fuzzyjoin,gsheet,RColorBrewer,wordcloud)
theme_set(theme_minimal())
```

Regular expressions

Regular expressions are a way to describe patterns in text. They are used in many different programming languages to find and manipulate text. In R, regular expressions are implemented in the `stringr` package.

Using the `stringr` package

The `stringr` package has a number of functions that allow you to manipulate text using regular expressions. The most commonly used functions are:

- `str_extract()` extracts the first match
- `str_extract_all()` extracts all matches
- `str_detect()` detects if a string matches a pattern
- `str_count()` counts the number of matches
- `str_locate()` locates the position of the first match
- `str_locate_all()` locates the position of all matches
- `str_replace()` replaces the first match
- `str_replace_all()` replaces all matches
- `str_split()` splits a string into a vector of strings
- `str_subset()` returns a subset of strings that match a pattern

Regular expression terms: There are many special symbols that can be used in regular expressions. Here are some of the most commonly used symbols:

- '\d' or '[0-9]' match any digit as does '[[:digit:]]' in **stringr**
- '\D' or '[^0-9]' match any non-digit as does '[[^:digit:]]' in **stringr**
- '\s' or '[[:space:]]" match any whitespace character
- '\S' or '[^[:space:]]' match any non-whitespace character
- '\w' or '[[:word:]]' match any word character (letter, number, underscore)
- '\W' or '[^[:word:]]' match any non-word character
- '\b' or '\B' match word boundaries or non-word boundaries
- '.' match any character except a newline
- '^', '$' match the start and end of a string
- '|' match either the expression before or after the pipe
- '\' precedes any special character to match it literally

For more on regular expressions, see resources like regex101, RegExplain, **stringr** Cheatsheet.

Today, you'll walk through an example of how you can use regular expressions to replicate text. I'll get you started out and then I'll ask you to try a few on your own.

First, we'll pull in the Ask A Manager dataset for 2023. You'll do some basic cleaning with it.

The following code reads in the data from the Ask A Manager Google Sheet. Then it immediately names the columns meaningful things. Finally, it converts all string variables to lowercase.

```
column_names <- c('timestamp','age','industry','area','jobtitle','jobtitle2',
        'annual_salary','income_additional','currency','currency_other',
        'income_additional','country','state','city','remote','experience_overall',
        'experience_field','education','gender','race')

managers2023 <- read_csv(gsheet::gsheet2text('https://docs.google.com/spreadsheets/d/  1ioUjhnz6ywSpEbA
  col_names = column_names,
  skip = 1)
```

```
## No encoding supplied: defaulting to UTF-8.
```

```
## New names:
## Rows: 17070 Columns: 20
## -- Column specification
## -------------------------------------------------------- Delimiter: "," chr
## (18): timestamp, age, industry, area, jobtitle, jobtitle2, currency, cur... dbl
## (2): annual_salary, income_additional...8
## i Use `spec()` to retrieve the full column specification for this data. i
## Specify the column types or set `show_col_types = FALSE` to quiet this message.
## * `income_additional` -> `income_additional...8`
## * `income_additional` -> `income_additional...11`
```

```
head(managers2023)
```

```
## # A tibble: 6 x 20
##   timestamp          age   industry       area  jobtitle jobtitle2 annual_salary
##   <chr>              <chr> <chr>          <chr> <chr>    <chr>            <dbl>
## 1 4/11/2023 11:02:00 35-44 Government & ~ Engi~ Materia~ <NA>            125000
## 2 4/11/2023 11:02:07 25-34 Galleries, Li~ Gall~ Assista~ <NA>             71000
## 3 4/11/2023 11:02:12 35-44 Education (Hi~ Educ~ Directo~ <NA>             60000
## 4 4/11/2023 11:02:15 25-34 Education (Hi~ Gove~ Adminis~ <NA>             42000
## 5 4/11/2023 11:02:25 18-24 Accounting, B~ Admi~ Executi~ <NA>             65000
## 6 4/11/2023 11:02:29 25-34 Government & ~ Law   Counsel  <NA>             88000
```

```
## # i 13 more variables: income_additional...8 <dbl>, currency <chr>,
## #   currency_other <chr>, income_additional...11 <chr>, country <chr>,
## #   state <chr>, city <chr>, remote <chr>, experience_overall <chr>,
## #   experience_field <chr>, education <chr>, gender <chr>, race <chr>
```

Homogenize Text Let's homogenize the text to make better sense of it. That means we want to lower case things and remove special characters.

If you look at the data, you likely see many special characters and different cases. Sometimes these are useful, sometimes they are not. For example, if you want to count the number of times a word appears in a text, you don't want to count "word" and "word." as two different words. Alternatively, to pick out percentages, you'll want to keep percent.

The following code uses the `tolower` and `str_replace_all()` functions to replace all special characters other than spaces with nothing. In `str_replace_all()`, the first argument is the string you want to replace. The second argument is the pattern you want to replace. The third argument is what you want to replace the pattern with. In this case, we want to replace the pattern with nothing.

Below I remove all of the non-alphanumeric and space characters from the `jobtitle` variable. Try adding a similar command to remove all of the non-alphanumeric and space characters other than the percent sign from the `income_additional` variable.

```
managers2023_no_special <- managers2023 %>%
  mutate(across(where(is.character), tolower),
      jobtitle=str_replace_all(jobtitle, "[^[:alnum:][:space:]]", ""))
```

Picking out forms of additional income Now, let's pick out the percentages from the `income_additional` variable. We'll use the `str_extract_all()` function to extract all of the percentages from the `income_additional` variable. The pattern we want to match is any number followed by a percent sign. The pattern for any number is '\d+'. The pattern for a percent sign is '\%'. The pattern for any number followed by a percent sign is '\d+\%'.

```
managers2023 %>% select(income_additional) %>%
  filter(!is.na(income_additional)) %>%
  mutate(add_percentage=str_extract(income_additional, '')) %>%
  head(5)
```

Are there other types of numbers in the `income_additional` variable that are not percentages? How can you find them? For example, someone may mention that if they exceed X hours they get a bonus. That will require a different pattern to flag and you want to save it as a separate column if you want to use it later. Why? You don't want to accidentally count the number of hours someone works as a percentage of their income in a calculation of their bonus pay.

Fuzzy match industry The industry variable is a mess. There are many different ways to describe the same industry. For example, "software" and "software development" are the same industry. This is because of free-form text entry. That's neat, but it makes it hard to analyze the data. Let's use the North American Industry Classification System (NAICS) data that the Census and BLS uses.

Naics follows a 2-6 digit system. The first two digits are the sector. The first three digits are the subsector. The first four digits are the industry group. The first five digits are the NAICS industry. The first six digits are the national industry.

For example, 54 is professional, scientific, and technical services. 541 is professional, scientific, and technical services. 5413 is architectural, engineering, and related services. 54133 is engineering services. 541330 is engineering services.

```
naics <- read_csv('https://www.bls.gov/cew/classifications/industry/industry-titles.csv')
```

```
## Rows: 2678 Columns: 2
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): industry_code, industry_title
##
```

```
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Let's use the `fuzzyjoin` package to match the industry variable to the NAICS data. The `fuzzyjoin` package allows you to match strings that are not exactly the same. This is useful when you have free-form text entry.

The `fuzzyjoin` package has a number of functions that allow you to match strings. There is a great vignette explaining how it works. The most commonly used functions are:

- `stringdist_left_join()` joins two data frames based on a string distance
- `stringdist_inner_join()` joins two data frames based on a string distance
- `stringdist_semi_join()` returns all rows from the first data frame that have a match in the second data frame
- `stringdist_anti_join()` returns all rows from the first data frame that do not have a match in the second data frame
- `stringdist_full_join()` joins two data frames based on a string distance

First we'll need to prepare the NAICS data a little bit. We'll need to make the industries lowercase to match the managers data. You'll also need to change the `str_replace_all()` function to remove all the leading numeric characters and the word "naics" from the `industry_title` variable, which I rename to `industry`.

There may be other changes you want to include like removal of special characters OR homogenizing of special characters. For example, use `str_replace_all()` to replace all the "&" in the NAICS data with "and."

Notably, if a NAICS code has no further subdivisions, the name is repeated and a zero added to the end of the code. For example, 11116 is Rice farming, which is as granular as that industry gets. 111160 is also Rice farming to make sure all code hierarchies terminate at six digits. To work around that, we'll just remove distinct duplicates.

I'm also removing the "NAICS" for 10, which is "Total, all industries" as that is a bit silly to match on.

```
naics_prep <- naics %>%
  rename(industry=industry_title) %>%
  mutate(industry=str_to_lower(industry),
    industry=str_replace_all(industry,'pattern','replacement')) %>%
  distinct(industry) %>%
  filter(industry_code!="10", industry!="unclassified")
```

Now to fuzzy join Now we'll use the `stringdist_left_join()` function to match the `industry` variable in the `managers2023` data to the `industry` variable in the `naics_prep` data. The `stringdist_left_join()` function takes four arguments. The first argument is the data frame you want to add the new variable to. The second argument is the data frame you want to match to. The third argument is the variable you want to match on in the first data frame. The fourth argument is the variable you want to match on in the second data frame. The `max_dist` argument is the maximum distance between the two strings. The `distance_col` argument is the name of the column that will be added to the first data frame that contains the distance between the two strings. The `method` argument is the method used to calculate the distance between the two strings. The `ignore_case` argument is whether or not to ignore case when calculating the distance between the two strings. The `full_join` argument is whether or not to do a full join. The `max_dist` argument is the maximum distance between the two strings. The `distance_col` argument is the name of the column that will be added to the first data frame that contains the distance between the two strings. The `method` argument is the method used to calculate the distance between the two strings. The `ignore_case` argument is whether or not to ignore case when calculating the distance between the two strings.

```
managers2023_naics <- head(managers2023_no_special,25) %>%
  stringdist_left_join(naics_prep, by=c('industry'='industry'), max_dist=.4, distance_col='distance', m
```

Check the data you got back. Notice any issues? For one, I got 4,972 rows back from 25 rows of data. That's a lot of rows. Why? Because there are many matches per row of data. For example, "government & public administration" matched to "natural resources and mining" with a 0.354 distance. That's not a good match, but it was permitted.

What will we get out of a better match? Instead of trying to match the groups of industries in the raw data, we'll have them immediately mapped to one key industry – the NAICS industry. That will make it easier to link to other datasets.

Of course, in the end it may be easier to do this by hand for the largest industries and then use the `fuzzyjoin` package to match the rest or even just drop the rest.

Make a wordcloud

Let's make a wordcloud of the income_additional variable. I'm curious what language is used to described unusual compensation schemes.

Tokens

We'll need to start by tokenizing the text. Tokenization is the process of breaking a string into tokens. A token is a sequence of characters that represents a unit of meaning. For example, a word is a token in a sentence. Tokenization is useful because it breaks a string into meaningful units that can be analyzed.

Navigate to the slides from today's lecture and augment the code below to make a wordcloud of the income_additional variable.

```
tokens <- managers2023 %>%
  select(jobtitle) %>%
  unnest_tokens(word,jobtitle) %>%
  count(word,sort=T)
tokens
```

```
## # A tibble: 2,384 x 2
##    word            n
##    <chr>       <int>
##  1 manager      3483
##  2 senior       1924
##  3 director     1856
##  4 engineer     1088
##  5 of            976
##  6 assistant     945
##  7 analyst       916
##  8 specialist    852
##  9 associate     800
## 10 coordinator   662
## # i 2,374 more rows
```

Stop words

There's probably a lot of words in the tokens data frame that are not useful. For example, "the" and "a" are not useful words. These are called stop words. Stop words are words that are so common that they are not useful for analysis.

The `tidytext` package has a built-in list of stop words. You can access it using the `stop_words` function. Use the `anti_join` function to remove the stop words from the tokens data frame.

```
data(stop_words)

tokens_no_stops <- tokens
```

Wordcloud

Last, let's make a wordcloud! I use pal `brewer.pal()` to make it more pleasing to the eye.

```
pal <- brewer.pal(8,"Dark2") # define a nice color palette with function from RColorBrewer

tokens_no_stops %>%
  with(wordcloud(word, n, random.order = FALSE, max.words = 50, colors=pal))
```

Bonus: 2-part ngrams

Can you make the same thing with 2-word ngrams?