Regex in R

S Rao (@SrinivasaRaoRao)

14 January 2019

Regular Expressions

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

- A 'language' to represent text patterns - concept invented in the 1950's
- Bound by a set of rules (syntax); a set of special characters used to denote patterns
- Multi-platform: available (natively or through libraries) in many languages and tools (R, Python, Java, sed, awk)
- Use cases:
    - read files with a specific naming pattern, e.g. 20190114_Mon_P1_W08_R2.csv, 20190114_Thurs_P10_W01_R3.csv
    - search for text patterns
    - replace text patterns

The basics

- Character classes `[]`
  - any character: `.`
  - alphabet: `[A-Z]` or `[:upper:]`, `[a-z]` or `[:lower:]`, `[A-Za-z]` or `[:alpha:]`
  - numeric: `[0-9]` or `[:digit:]` or `\d`
  - alphanumeric: `[A-Za-z0-9]` or `[:alnum:]`
  - whitespace (space, tab, linebreak): `\s`
- quantifiers:
  - one or more (of the preceding character): `+`
  - zero or more: `*`
  - zero or one: `?`
  - specified number: `{m,}`, `{m,n}`

The basics

- anchors:
  - start: `^` (except in the context of `[^ ]`, where it is negation)
  - end: `$`
- capture groups:
  - extract groups: `()`
  - refer to captured groups: `\1`, `\2`, etc.
- metacharacters: `. \ | ( ) [ { ^ $ * + ? ,`

Examples

- https://regexr.com/
- in the string "the cat in the hat has a bat":
  - `[ch]at` matches `cat` and `hat`
  - `.at` matches `cat`, `hat` and `bat`
  - `[:alpha:]{1,2}` matches `in` and `a`
  - `.\s.` matches `e c`, `t i`, `n t`, `e h`, `t h`, `s a`, `a b`

Strings in R

- Strings ("character" class) are represented in R using " or '
- But what about special characters like newlines and tabs? They are represented as escape sequences.`print` prints the escape sequence, whereas `cat` processes them.

```
string = "First\tline\nSecond\tline"
print(string)
```

```
## [1] "First\tline\nSecond\tline"
```

```
cat(string)
```

```
## First    line
## Second   line
```

Strings in R

- What if the string contains an invalid escape character?

```
regex_string = ".\s."
```

```
## Error: '\s' is an unrecognized escape in character string starting "".\s"
```

- Regular expressions are represented as strings in R. But strings are processed first for escape characters. Unrecognised escape characters in strings throw an error, before even reaching the regex parser.
- *Double backslashes* needed for regex escape sequences

```
regex_string = ".\\s."
string = "the cat in the hat has a bat"
regexpr(regex_string, string)
```

[1] 3 attr(,"match.length") [1] 3 attr(,"useBytes") [1] TRUE

Quadruple backslashes!

- How do you match a literal backslash then?

```
string = "Windows paths use \\ instead of /"
cat(string)
```

```
## Windows paths use \ instead of /
```

```
regex_string = "\\\\"
#str_detect(string, regex_string)
regexpr(regex_string, string)
```

```
## [1] 19
## attr(,"match.length")
## [1] 1
## attr(,"useBytes")
## [1] TRUE
```

Base R functions that use regex

- `grep()`
- `grepl()`
- `regexpr()`
- `gregexpr()`
- `sub()`
- `gsub()`
- `strsplit()`
- `list.files()`

Stringr functions

As with other Tidyverse functions, Stringr functions take the *text* as the first argument and the pattern as the second argument

- `str_locate()` - like `regexpr()`, but returns an integer matrix
- `str_detect()` - like `grepl()`
- `str_split()` - like `strsplit()`
- `str_extract()` - like `match = regexpr(pattern, string); substring(string, match, match + attr(match, "match.length") - 1)`

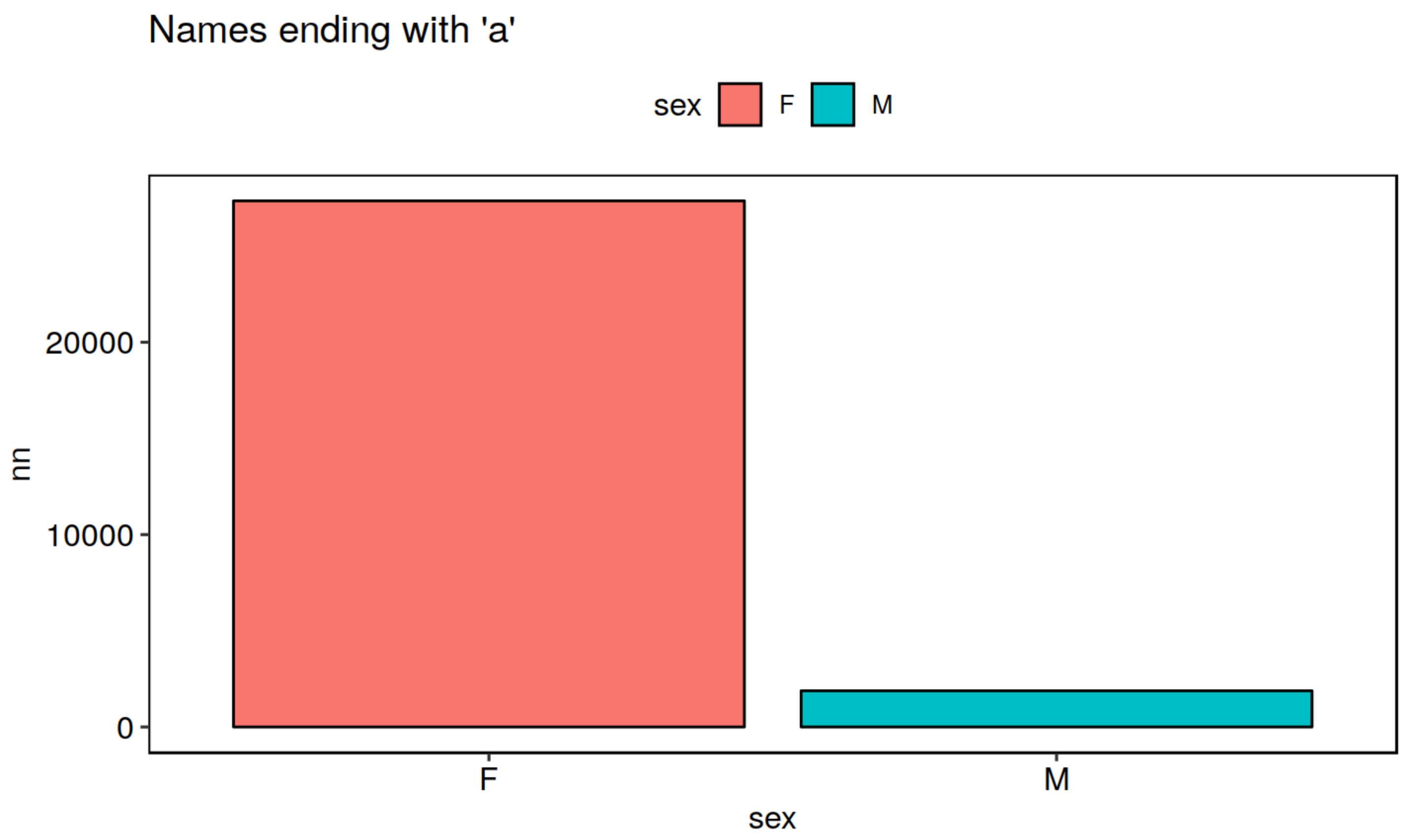
Practical example

https://www.theatlantic.com/notes/2015/08/why-do-so-many-girlss-names-end-in-a/402823/

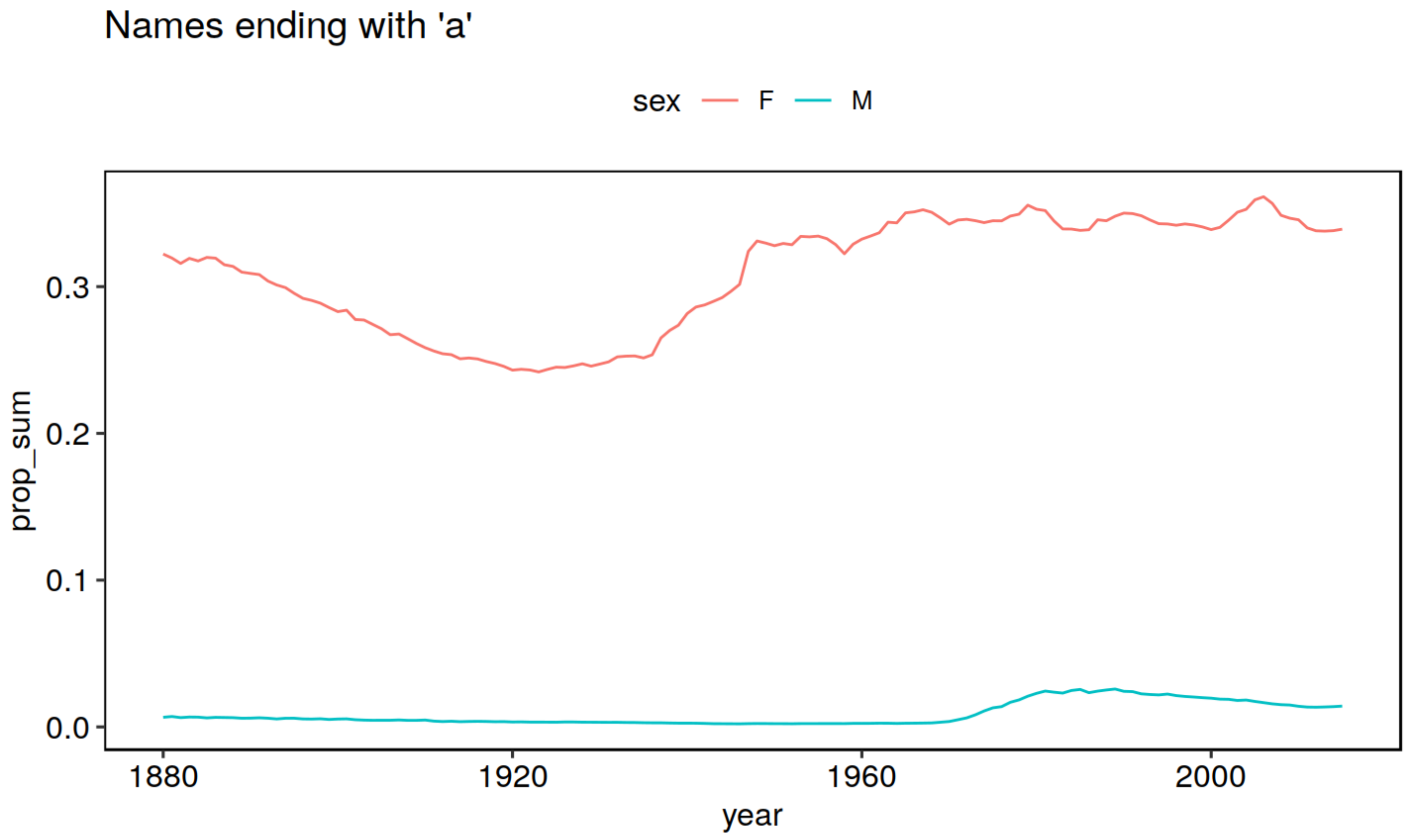
https://qz.com/1278574/a-large-share-of-us-baby-names-end-with-n-for-some-reason/

- Using the babynames package (US Baby Names 1880-2017), let's look at trends in naming babies in the USA

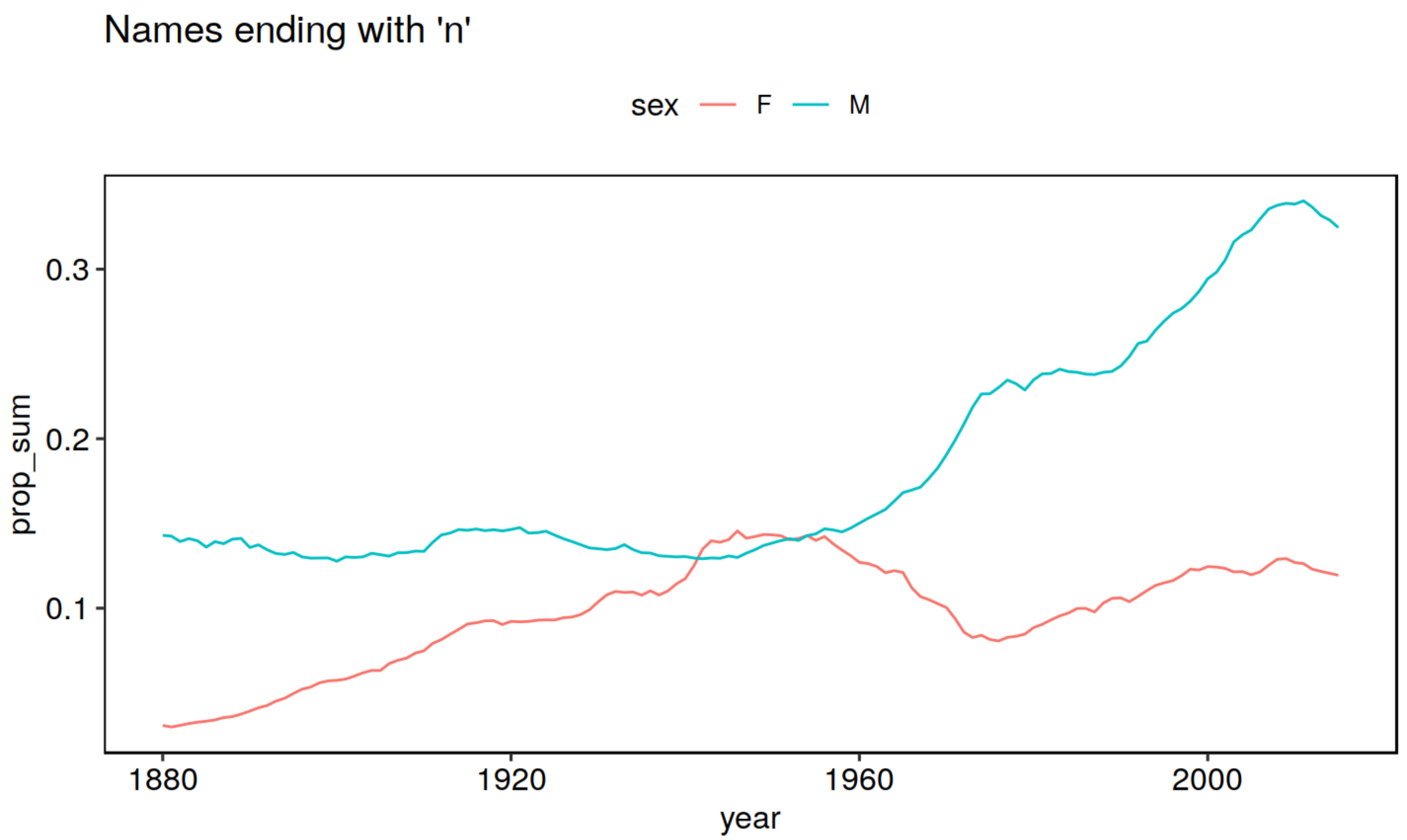
Names ending in 'a'



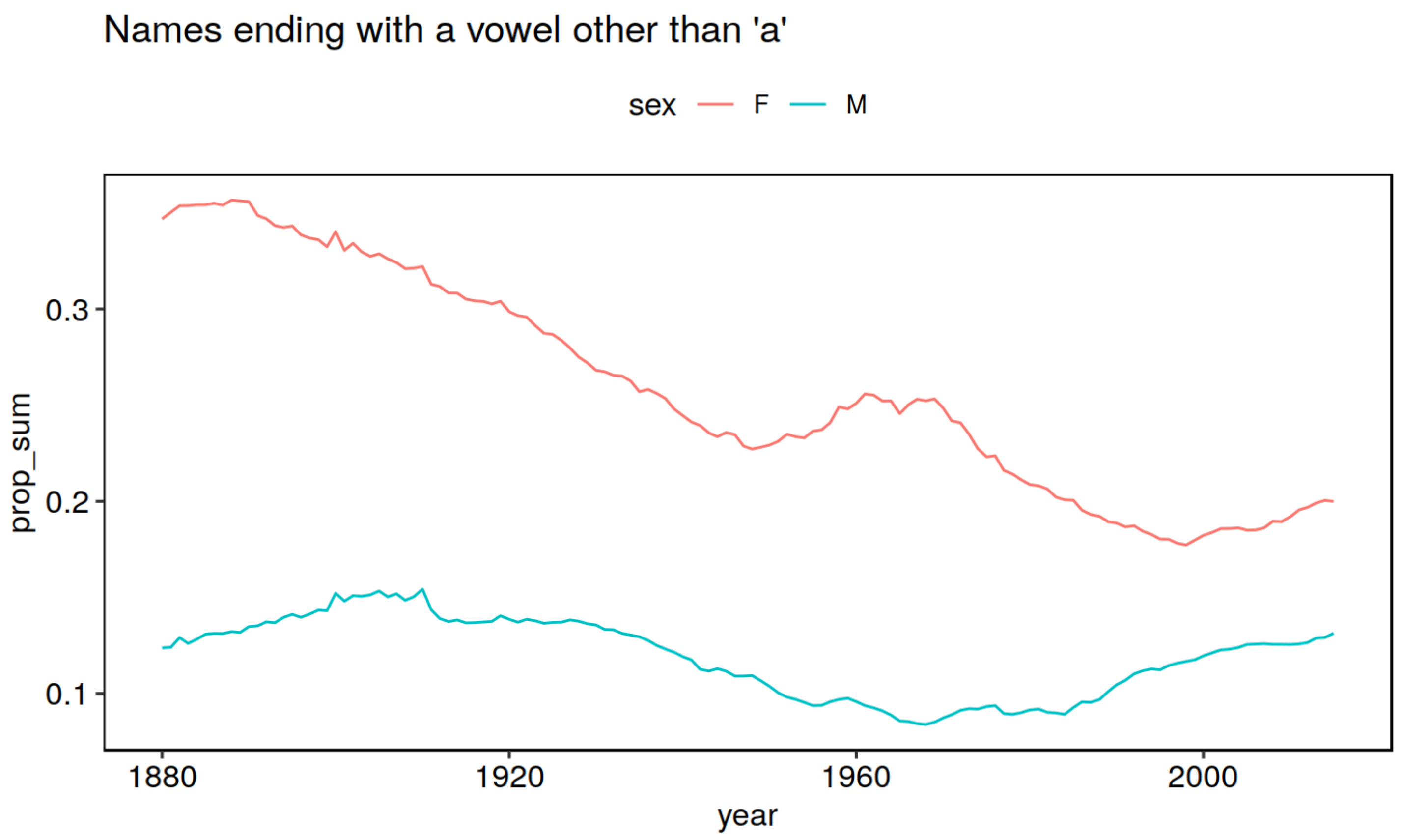
Names ending in 'a': trends



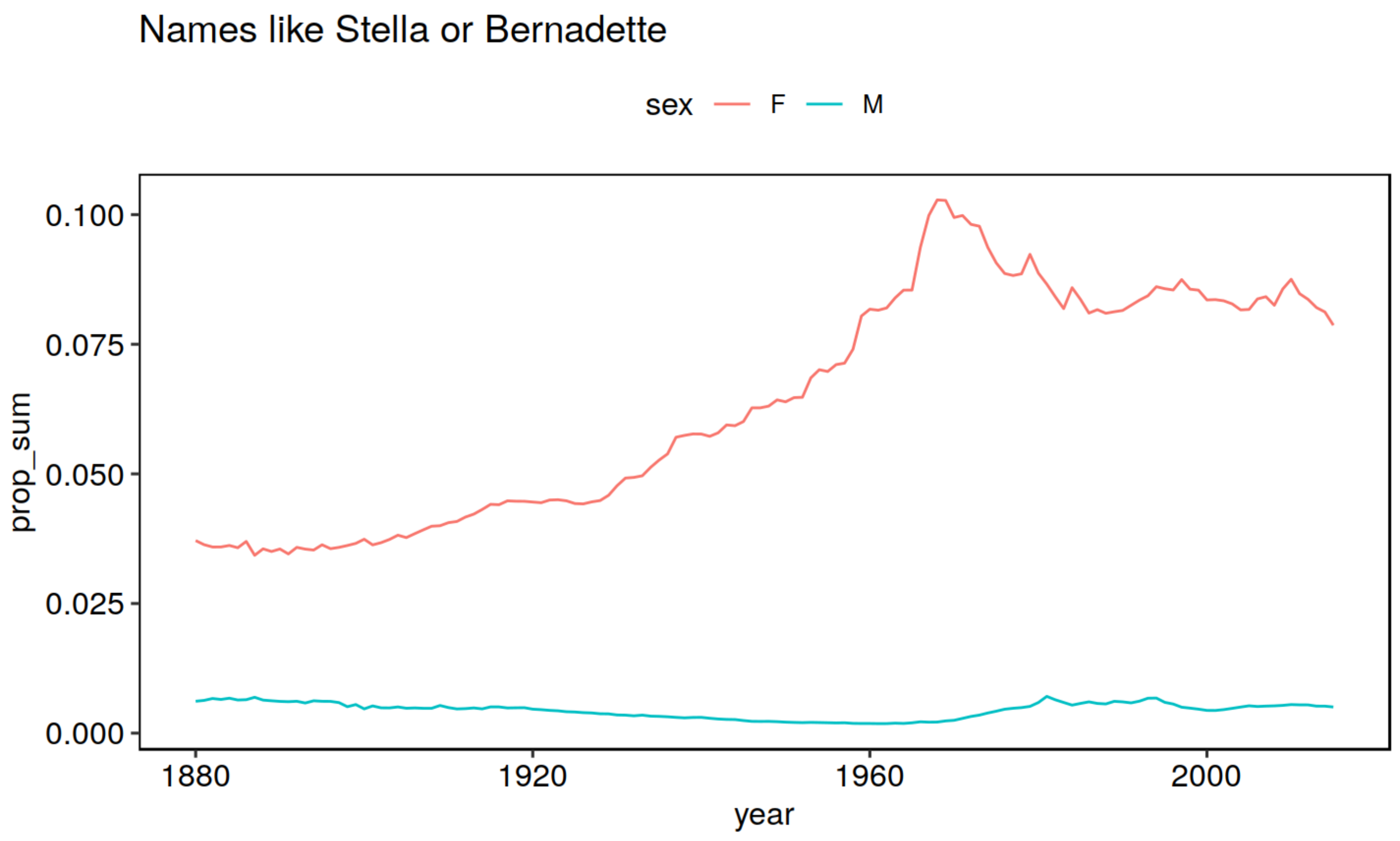
Names ending in 'n': trends



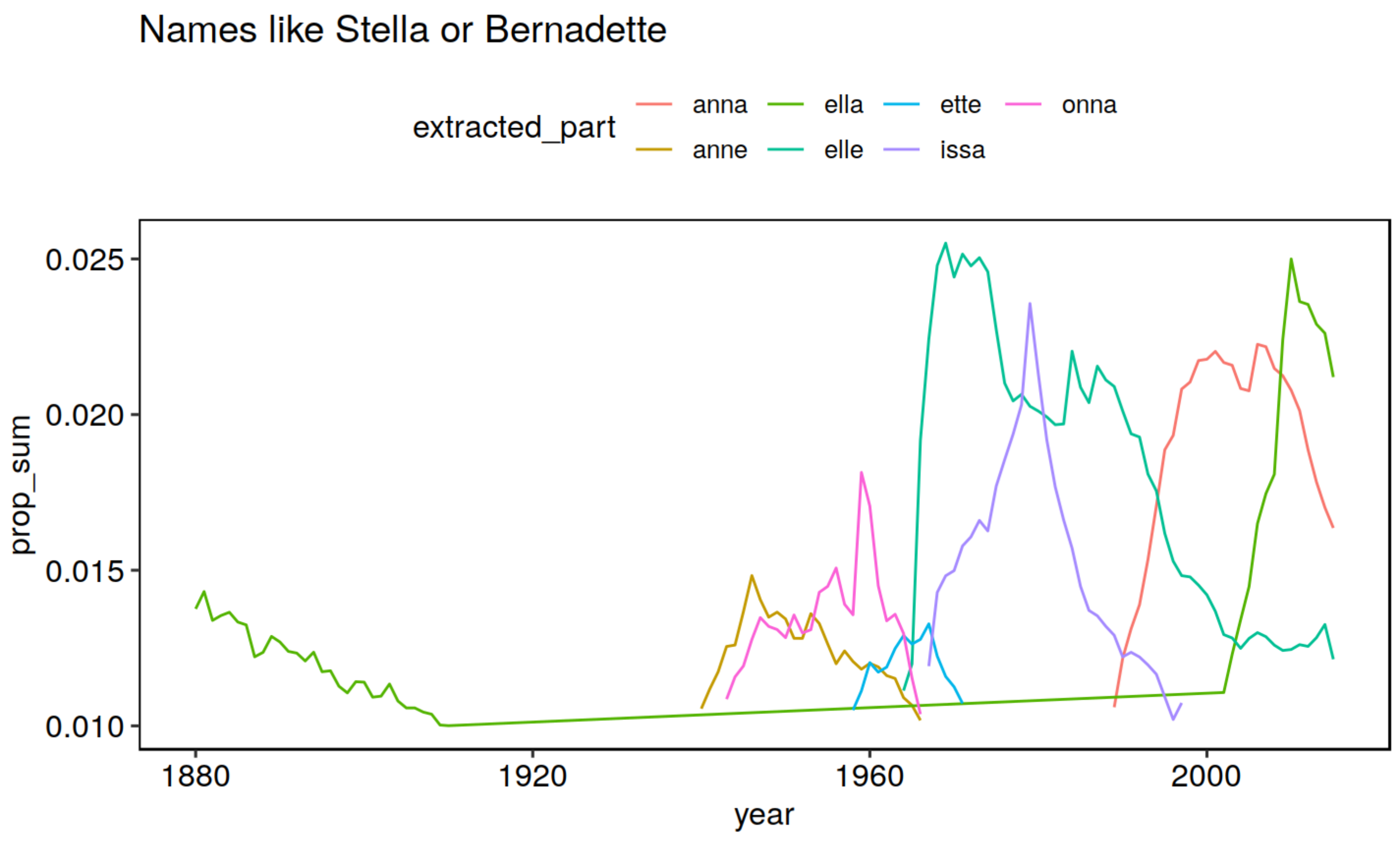
Names ending in a vowel other than 'a': trends



Names ending in a repeating pattern



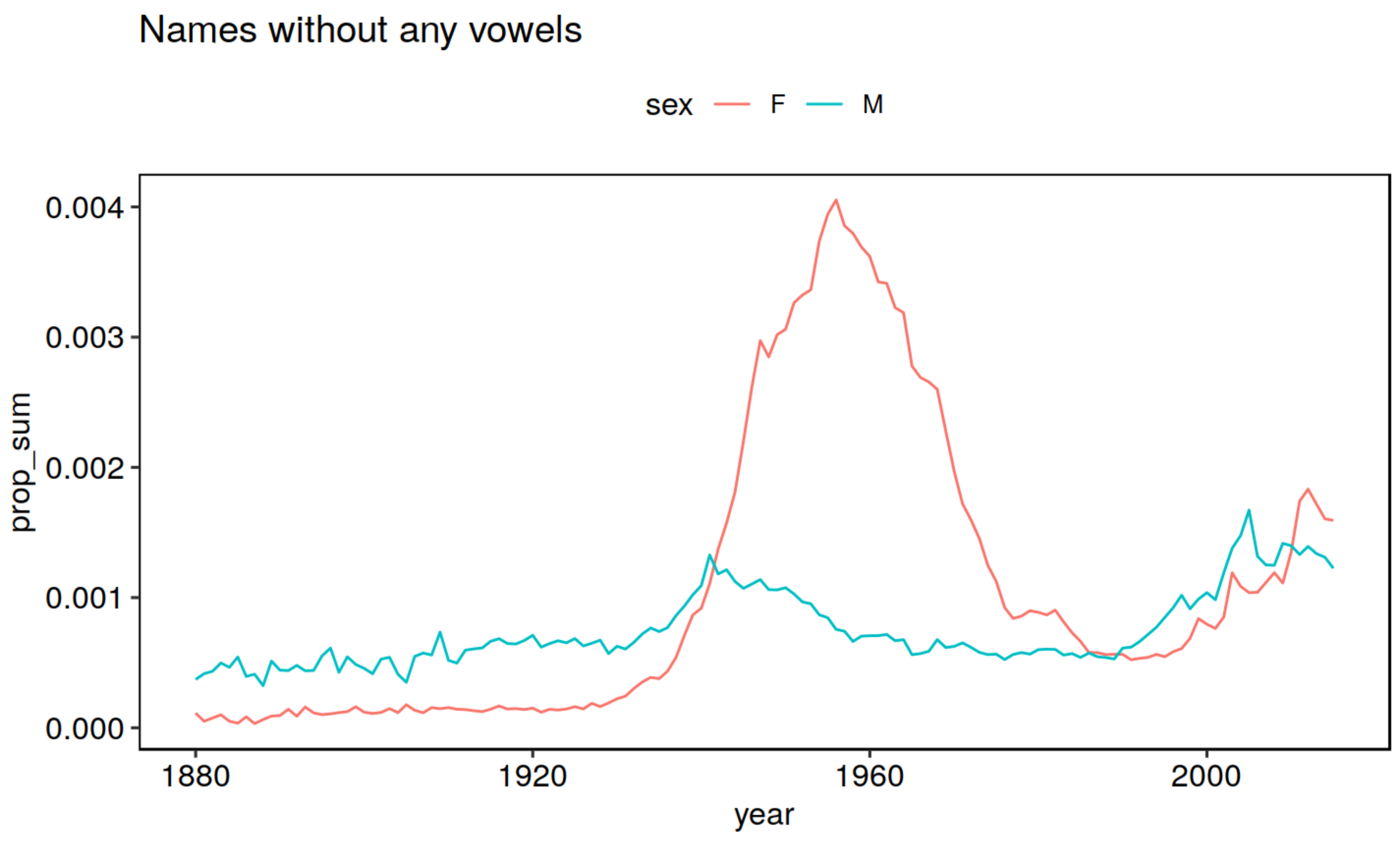
Top repeating patterns



Names with other repeating patterns

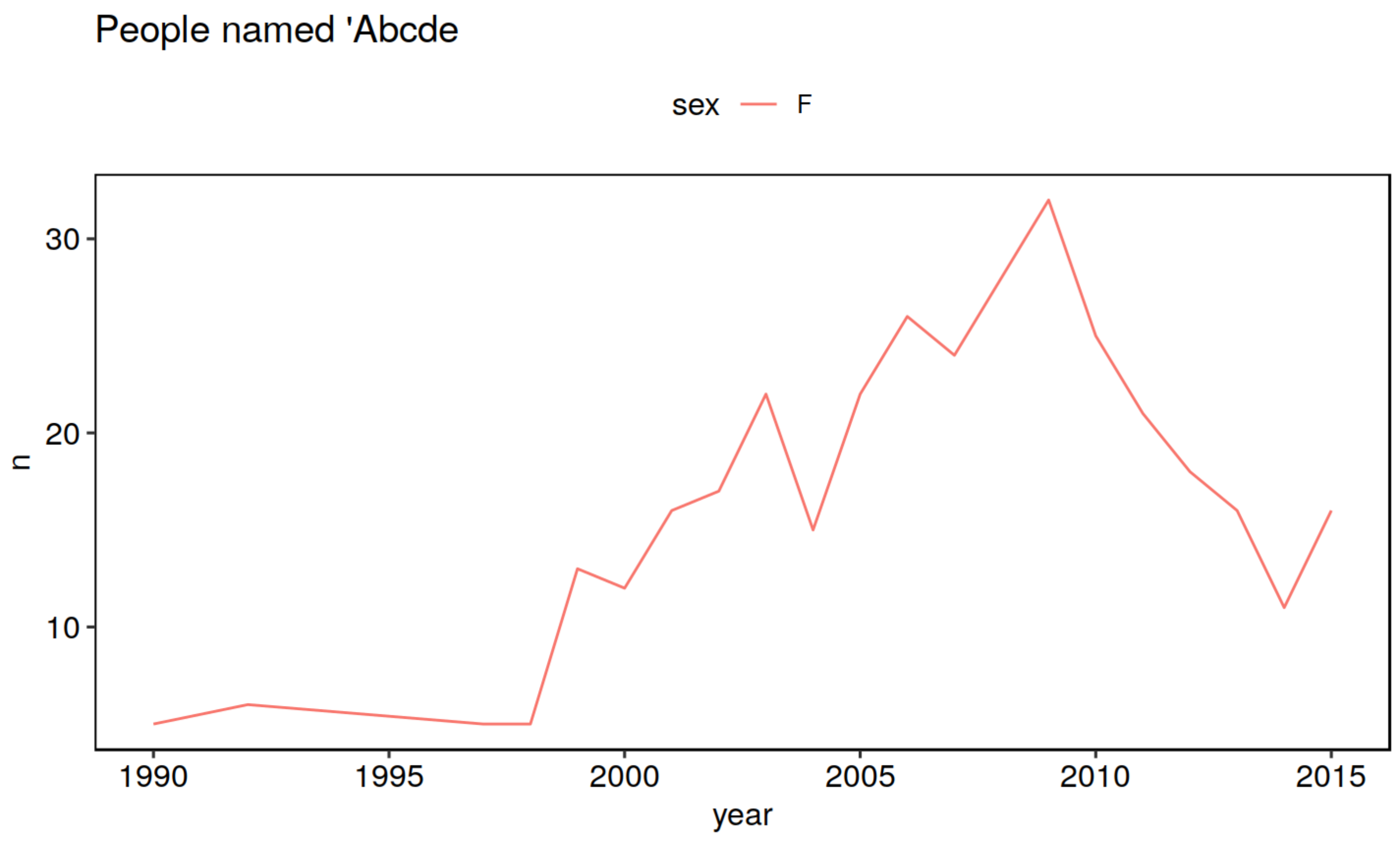
```
## # A tibble: 6 x 3
## # Groups:   name [6]
##   name     sex   total
##   <chr>    <chr> <int>
## 1 Letitia  F     11381
## 2 Jedidiah M      6109
## 3 Janene   F      3544
## 4 Jedediah M      2793
## 5 Jeanene  F      2425
## 6 Hanan    F      1517
```

Names without any vowels

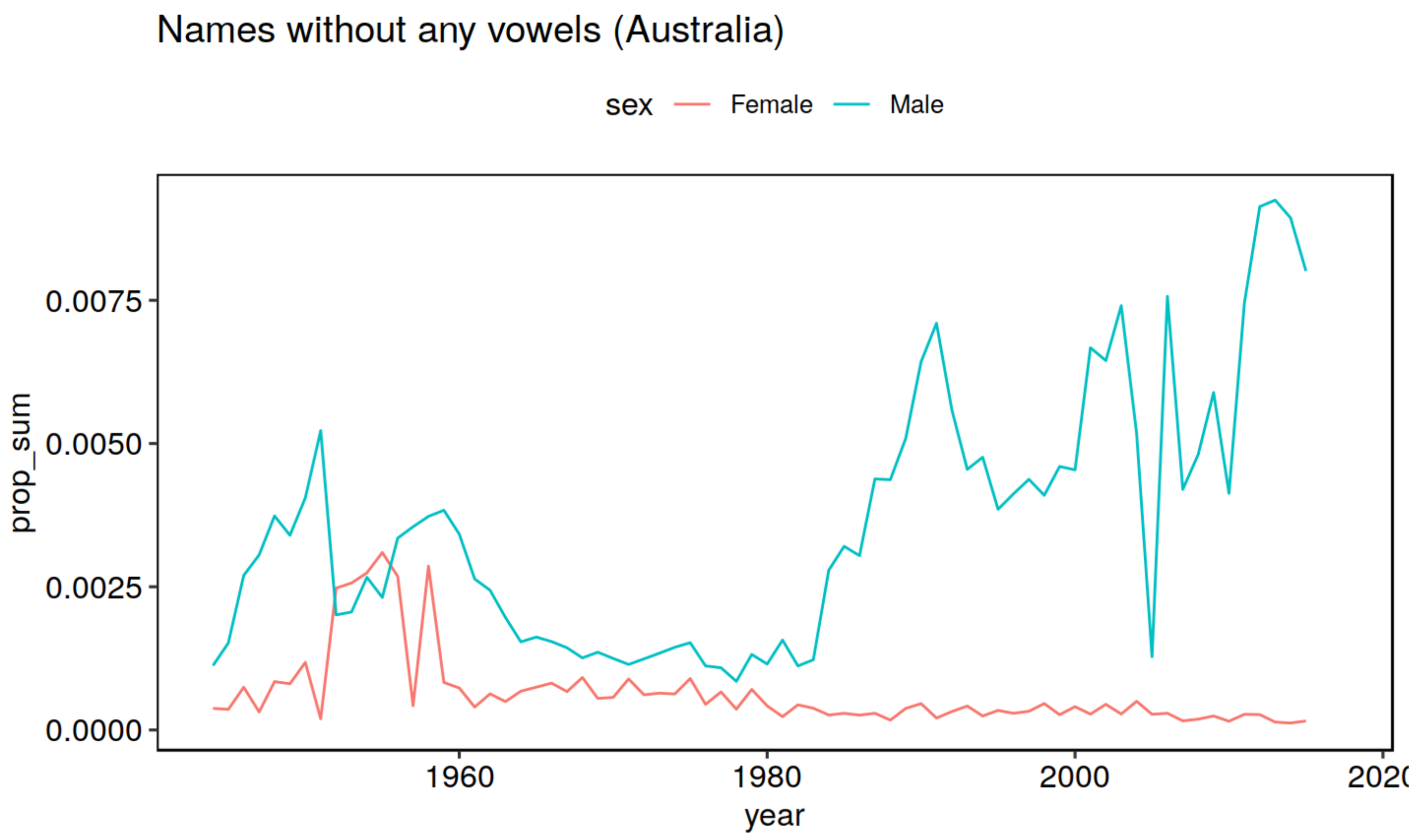


Abcde

https://www.straitstimes.com/world/united-states/us-airline-apologises-after-employee-mocks-child-named-abcde

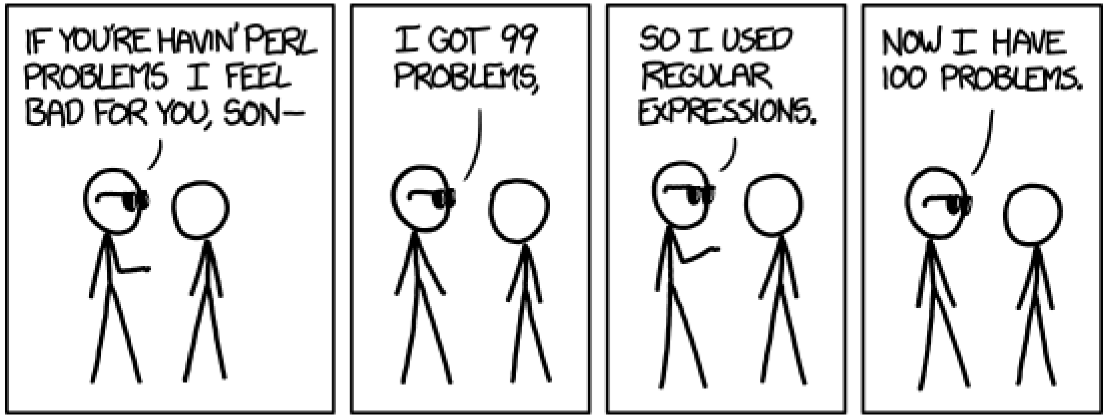


ozbabynames



Two problems?

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems. - Jamie Zawinski


IF YOU'RE HAVIN' PERL PROBLEMS I FEEL BAD FOR YOU, SON—
I GOT 99 PROBLEMS,
SO I USED REGULAR EXPRESSIONS.
NOW I HAVE 100 PROBLEMS.

Now I have 100 problems https://xkcd.com/1171/

https://blog.codinghorror.com/regex-use-vs-regex-abuse/

Commenting within a regex

- natively with `(?#...)`
- `stringr::regex(comments = TRUE)` allows for more legible formatting and commenting of regular expressions
    - ignores spaces and newlines (literal space must be escaped with \)
    - ignores everything after #

```
rep_letter_pattern <- stringr::regex("[aeiou] # a lower case vowel
                                      ([a-z])  # any lower case letter, captured as the 1s
                                      \\1      # the above group is repeated
                                      [aeiou]  # a lower case vowel
                                      $        # the above 4 characters must be at the end
                                      ", comments = TRUE)
str_extract("Arabella", rep_letter_pattern)
```

```
## [1] "ella"
```