Assignment 1

Steve Kerr (211924)

2023-09-18

Introduction

This first assignment will test your grasp of some of the concepts and methods we covered in sessions 1-2. Give an answer to each question, even if you are not sure. In an ideal world, you should present your work as a document with integrated code, execution and text, but if this doesn't work out a script and a pdf will do fine.

1

Consider the two following texts.

```
texts <- c(
  "I don't like cricket",
  "You like cricket"
)
```

1.1 Write two text processing pipelines that create document feature matrices that **do** and **do not** preserve the differences in the texts.

We'll start by loading the `quanteda` package.

```
library(quanteda)
```

Here's our first text preprocessing pipeline. Note that it **does** preserve the differences between the two texts.

```
texts |>
  tokens() |>
  dfm()
```

```
## Document-feature matrix of: 2 documents, 5 features (30.00% sparse) and 0 docvars.
##        features
## docs    i don't like cricket you
##   text1 1     1    1       1   0
##   text2 0     0    1       1   1
```

And here's our second text preprocessing pipeline. Note that it **does not** preserve any of the differences between the two texts.

```
texts |>
  tokens() |>
  tokens_remove(pattern = stopwords("en")) |>
  dfm()
```

```
## Document-feature matrix of: 2 documents, 2 features (0.00% sparse) and 0 docvars.
##        features
## docs    like cricket
##   text1    1       1
##   text2    1       1
```

1.2. Describe how the preprocessing choices you make affect the representation of those texts.

> Preprocessing choices have the potential to significantly impact the representation of the two texts provided. Specifically, stopword removal (i.e. the removal of common words) can lead to the loss of important context. This may or may not significantly impact the results of a given analysis, depending largely on the level of nuance required to accomplish the task at hand.

1.3. Give one example each of **tasks** where your text preprocessing choices that do not preserve the differences in the texts **would** and **would not** limit our ability to perform the task.

> One such example of a task where failing to preserve the differences between the texts **would not** impact our analysis is topic modeling which seeks to group together texts that deal with a similar topic. In such cases, it wouldn't matter whether differences were preserved since we would still be able to make accurate predictions of whether a given text pertained to "cricket" or even "sports" more broadly, for example.
>
> In contrast, a task where such text preprocessing **would** have the potential to impact our analysis is sentiment analysis that aims to attribute a positive or negative valence to text. Such a task requires context contained in stop words and their proximity to other words in the text.

2

Now consider the following three texts.

```
texts <- c(
  "Climatic change is causing adverse impacts",
  "Changes in the climate have caused impacts to human systems",
  "Chelsea have a goal difference of zero in the premier league this season"
)
```

2.1. Turn these texts into a document feature matrix without any additional pre-processing steps

```
# basic preprocessing pipeline
dfmat <- texts |>
  tokens() |>
  dfm()

dfmat
```

```
## Document-feature matrix of: 3 documents, 25 features (61.33% sparse) and 0 docvars.
##        features
## docs    climatic change is causing adverse impacts changes in the climate
##   text1        1      1  1       1       1       1       0  0   0       0
##   text2        0      0  0       0       0       1       1  1   1       1
##   text3        0      0  0       0       0       0       0  1   1       0
## [ reached max_nfeat ... 15 more features ]
```

2.2. Calculate a simplistic measure of how similar each text is to each other, by reporting the number of columns where both texts contain a non-zero value (you can do this with code or by hand).

```
# load packages
pacman::p_load(
  dplyr,
  glue,
  tidytext,
  tidyr
  )

# write function to count number of words common to two texts
compare_text <- function(dfmat, text1, text2) {

  sim_score <- dfmat |>
    tidy() |>
    filter(document %in% c(text1, text2)) |>
    complete(document, term, fill = list(count = 0)) |>
    summarize(n = sum(count), .by = term) |>
    filter(n > 1) |>
    count() |>
    pull(n)

  return(glue("{text1} and {text2} share {sim_score} word(s)"))

}

# texts 1 & 2
compare_text(dfmat, "text1", "text2")
```

```
## text1 and text2 share 1 word(s)
```

```
# texts 1 & 3
compare_text(dfmat, "text1", "text3")
```

```
## text1 and text3 share 0 word(s)
```

```
# texts 2 & 3
compare_text(dfmat, "text2", "text3")
```

```
## text2 and text3 share 3 word(s)
```

> As we can see, the second and third texts have the most words in common (3 words). Despite this, these two texts pertain to entirely different topics, demonstrating that simply counting the number of words shared between two texts without performing any sort of preprocessing is not an effective method for determining whether two texts are related.

2.3. Create a text processing pipeline that results in a matrix that preserves the similarity we can see intuitively between the texts.

```
# new preprocessing pipeline
dfmat_u <- texts |>
  tokens() |>
  tokens_remove(pattern = stopwords("en")) |>
  tokens_wordstem() |>
  dfm()

# texts 1 & 2
compare_text(dfmat_u, "text1", "text2")
```

```
## text1 and text2 share 4 word(s)
```

```
# texts 1 & 3
compare_text(dfmat_u, "text1", "text3")
```

```
## text1 and text3 share 0 word(s)
```

```
# texts 2 & 3
compare_text(dfmat_u, "text2", "text3")
```

```
## text2 and text3 share 0 word(s)
```

2.4. Comment on why the additional pre-processing steps created a more useful representation of the texts *in this case*.

> As we can see, our new text preprocessing pipeline is able to create more useful representations of the texts as evidenced by the fact that it identifies a stronger similarity between texts 1 and 2 and no similarities between these and text 3 which aligns with our intuition. By introducing stop word removal, we isolate the terms that are most likely to be useful for our analysis, thus removing noise from the data. Additionally, word stemming reduces each term to its most basic form, enabling our code to determine that words that share the same base are similar even when they aren't identical. Together, these additional preprocessing steps significantly improve our code's performance.