Text Mining and Sentiment Analysis

Prof. Annamaria Bianchi
A.Y. 2024/2025

Lecture 12

25 March 2025

UNIVERSITÀ DEGLI STUDI DI BERGAMO | Dipartimento di Scienze Economiche

Outline

Using Bigrams to provide context in SA

SA with negators, amplifiers and deamplifiers

Packages: **tidytext, tidyverse, udpipe**

Functions: `udpipe::udpipe(), udpipe::txt_sentiment()`

Using Bigrams to provide context in SA

By performing SA on the bigrams data, we can examine how often sentiment-associated words are preceded by «not» or other negating words.

Let us use the AFINN lexicon for SA and examine the most frequent words preceded by «not» and associated with a sentiment

```
> not_words = bigrams_separated |>
+ filter(word1 == "not") |>
+ inner_join(afinn, by =c("word2" = "word")) |>
+ count(word2, value, sort = T)
```

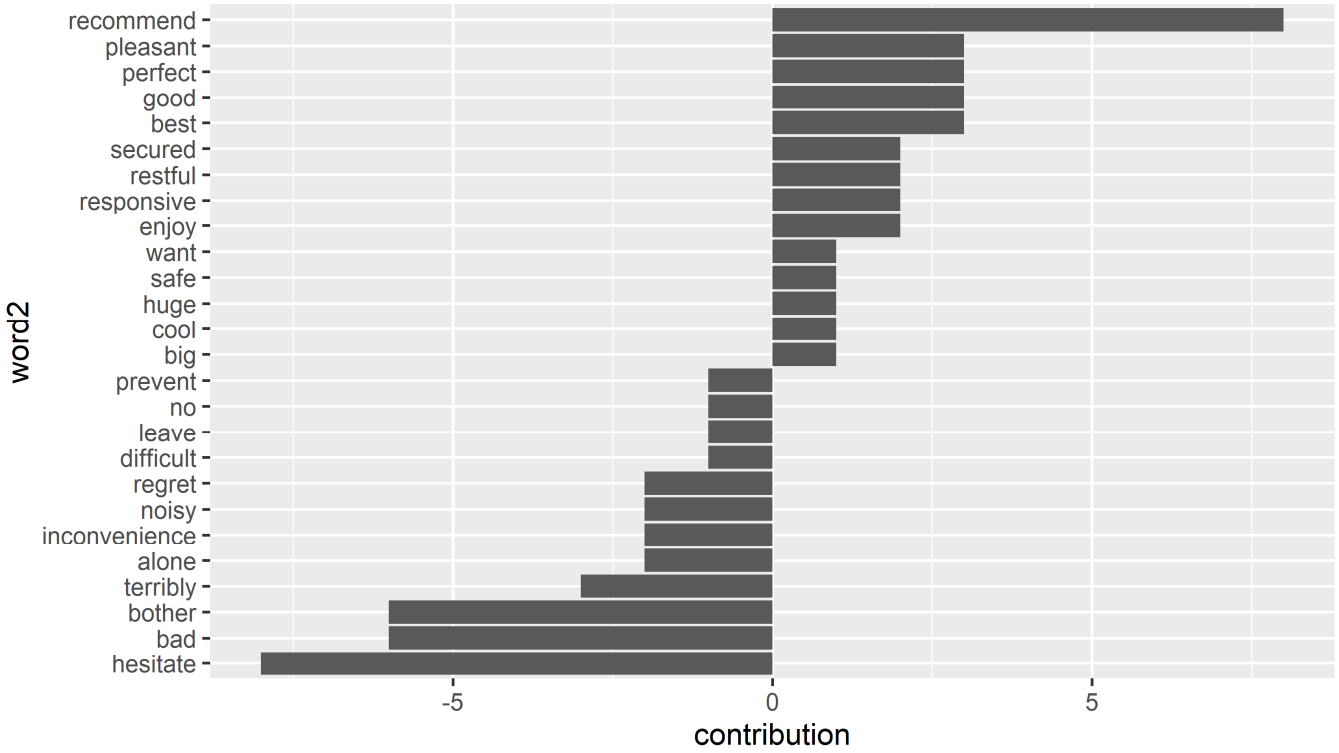
	word2	value	n
1	hesitate	-2	4
2	recommend	2	4
3	bother	-2	3
4	bad	-3	2
5	noisy	-1	2
6	alone	-2	1
7	best	3	1

Using Bigrams to provide context in SA

It's worth asking which words contributed the most to the «wrong direction». To compute that, we shall multiply their score by the number of times they appear and visualize the results with a bar plot

```
> not_words |>
+ mutate(contribution = n*value) |>
+ slice_max(order_by = abs(contribution), n = 20) |>
+ mutate(word2 = reorder(word2, contribution)) |>
+ ggplot(aes(word2, contribution), fill = contribution>0) +
+ geom_col()+
+ coord_flip()
```

Using Bigrams to provide context in SA



Package udpipe

An R package which is an Rcpp wrapper around the UDPipe C++ library

udpipe allows to easily perform a number of important steps in NLP:

(1) Tokenization,

(2) Parts of Speech tagging,

(3) Lemmatization and

(4) Dependency Parsing.

Further, it also allows you to train your own annotator models directly from R and to perform sentiment analysis using negators and amplifiers.

```
> install.packages("udpipe")
> library(udpipe)
```

udpipe function

Currently the package allows you to do tokenisation, tagging, lemmatization and dependency parsing with one convenient function called `udpipe`

```
udpipe(x, object, ...)
```

`x` either a character vector or a data.frame (with columns `doc_id` and `text`) or a list of tokens. All text data should be in UTF-8 encoding

`object` either an object of class `udpipe_model`, the path to the file on disk containing the udpipe model or the language as defined by `udpipe_download_model`. If the language is provided, it will download the model using `udpipe_download_model`.

The function returns a data.frame with one row per `doc_id` and `term_id` containing all the tokens in the data, the lemma, the part of speech tags, the morphological features and the dependency relationship along the tokens.

udpipe function

The output data.frame has the following fields:

- `doc_id`: The document identifier
- `paragraph_id`: The paragraph identifier which is unique within each document.
- `sentence_id`: The sentence identifier which is unique within each document.
- `sentence`: The text of the sentence of the `sentence_id`.
- `start`: Integer index indicating in the original text where the token starts.
- `end`: Integer index indicating in the original text where the token ends.
- `term_id`: A row identifier which is unique within the `doc_id` identifier.
- `token_id`: Token index, integer starting at 1 for each new sentence. May be a range for multiword tokens or a decimal number for empty nodes.
- `token`: The token.
- ……

The columns `paragraph_id`, `sentence_id`, `term_id`, `start`, `end` are integers, the other fields are character data in UTF-8 encoding.

Encoding

The `base::iconv` function allows to set and convert the encoding scheme.

```
base::iconv(x =text_data, from = '', to = '')
```

`x` a character vector

`from` a character string describing the current encoding

`to` a character string describing the target encoding

Encoding

Let us set UTF-8 as encoding for comments in bos.airbnb data.

```
> bos.airbnb$comments = iconv(bos.airbnb$comments, to = "UTF-8")
```

udpipe function

In order to save space, we create a new tibble, called data with the required columns names and we apply the udpipe function.

The `udpipe()` function requires that the columns names are `doc_id` and `text`.

```
> data <- bos.airbnb |>
+ rename(doc_id = ID,
+ text = comments)

> output <- udpipe(data, "english-gum")
> View(output)
```

udpipe function

	doc_id	paragraph_id	sentence_id	sentence	start	end	term_id	token_id	token
1	1	1	1	My daughter and I had a wonderful stay with Maura.	1	2	1	1	My
2	1	1	1	My daughter and I had a wonderful stay with Maura.	4	11	2	2	daughter
3	1	1	1	My daughter and I had a wonderful stay with Maura.	13	15	3	3	and
4	1	1	1	My daughter and I had a wonderful stay with Maura.	17	17	4	4	I
5	1	1	1	My daughter and I had a wonderful stay with Maura.	19	21	5	5	had
6	1	1	1	My daughter and I had a wonderful stay with Maura.	23	23	6	6	a
7	1	1	1	My daughter and I had a wonderful stay with Maura.	25	33	7	7	wonderful
8	1	1	1	My daughter and I had a wonderful stay with Maura.	35	38	8	8	stay
9	1	1	1	My daughter and I had a wonderful stay with Maura.	40	43	9	9	with
10	1	1	1	My daughter and I had a wonderful stay with Maura.	45	49	10	10	Maura
11	1	1	1	My daughter and I had a wonderful stay with Maura.	50	50	11	11	.
12	1	1	2	She kept in close touch with us throughout the day as we weren't arriving til later in the evening.	52	54	12	1	She
13	1	1	2	She kept in close touch with us throughout the day as we weren't arriving til later in the evening.	56	59	13	2	kept
14	1	1	2	She kept in close touch with us throughout the day as we weren't arriving til later in the evening.	61	62	14	3	in
15	1	1	2	She kept in close touch with us throughout the day as we weren't arriving til later in the evening.	64	68	15	4	close

Sentiment analysis with negators and (de)amplifiers

The function `udpipe::txt_sentiment()` identifies words which have a positive/negative meaning, with the addition of some basic logic regarding occurrences of amplifiers/deamplifiers and negators in the neighbourhood of the word which has a positive/negative meaning.

- If a negator is occurring in the neigbourhood, positive becomes negative or vice versa.
- If amplifiers/deamplifiers occur in the neigbourhood, these amplifier weight is added to the sentiment polarity score.

The function works on a udpipe-tokenised dataset.

Sentiment analysis with negators and (de)amplifiers

`udpipe::txt_sentiment()` implements a dictionary-based sentiment analysis considering the following elements:

- **polarity_terms:** a data frame with the `term` and the `polarity`;
- **polarity negators**: multiply the polarity of the term by -1;
- **polarity amplifiers** (or **deamplifiers**): multiply the polarity of the term by a weight (0.8 here);
- **n_before** and **n_after**: is the number of words before/after the terms where to search for amplifiers or negators;
- **constrain**: normalize the score between -1 and 1.

Sentiment analysis with negators and (de)amplifiers

```
txt_sentiment(x, term = "lemma", polarity_terms, polarity_negators = character(),
polarity_amplifiers = character(), polarity_deamplifiers = character(),
amplifier_weight = 0.8, n_before = 4, n_after = 2, constrain = FALSE)
```

`x` a data.frame with the columns doc_id, paragraph_id, sentence_id, upos and the column as indicated in term. This is exactly what udpipe returns.

`term` a character string with the name of a column of x where you want to apply to sentiment scoring upon

`polarity_terms` data.frame containing terms which have positive or negative meaning. This data frame should contain the columns term and polarity where term is of type character and polarity can either be 1 or-1.

`polarity_negators` a character vector of words which will invert the meaning of the `polarity_terms` such that-1 becomes 1 and vice versa

`polarity_amplifiers` a character vector of words which amplify the `polarity_terms`

`polarity_deamplifiers` a character vector of words which deamplify the `polarity_terms`

`amplifier_weight` weight which is added to the polarity score if an amplifier occurs in the neighbourhood

Sentiment analysis with negators and (de)amplifiers

`n_before` indicating how many words before the polarity_terms word one has to look to find negators/amplifiers/deamplifiers to apply its logic

`n_after` integer integer indicating how many words after the polarity_terms word one has to look to find negators/amplifiers/deamplifiers to apply its logic

`constrain` logical indicating to make sure the aggregated sentiment scores is between-1 and 1.

Output. A list containing

- data: the x data.frame with 2 columns added: `polarity` and `sentiment_polarity`.

  The column polarity being just the polarity column of the polarity_terms dataset corresponding to the polarity of the term you apply the sentiment scoring
  The columnn sentiment_polarity is the value where the amplifier/de-amplifier/negator logic is applied on.

- overall: a data.frame with one row per doc_id containing the columns doc_id, sentences, terms, sentiment_polarity, terms_positive, terms_negative, terms_negation and terms_amplification providing the aggregate sentiment_polarity score of the dataset x by doc_id as well as the terminology causing the sentiment, the number of sentences and the number of non punctuation terms in the document.

Simple examples

Let's consider the following sentences with the same sentiment word "love" and a negator (not), an amplifier "really" and a deamplifier "barely".

```
> text = c("I love this car",
+ "I really love this car",
+ "I do not love this car",
+ "I really do not love this car",
+ "I barely love this car")

> udpipe_text <- udpipe(text, "english-gum")
> View(udpipe_text)
```

Simple examples

Exercise. The argument `polarity_terms` in the `txt_sentiment()` function should provide a data.frame containing terms which have positive or negative meaning. This data frame should contain the columns *term* and *polarity* where term is of type character and polarity can either be 1 or-1.

Starting from the lexicon bing in the package tidytext, built a proper data.frame to be used in the `txt_sentiment()` function.

Simple examples

Let us perform a standard SA, without applying any logic regarding occurrences of amplifiers/deamplifiers and negators

```
> scores <- txt_sentiment(x = udpipe_text,term = "token",
+ polarity_terms = bing_dict,
+ polarity_negators = NULL,
+ polarity_amplifiers = NULL,
+ amplifier_weight = 0.8,
+ n_before = 0,
+ n_after = 0,
+ constrain = F)
```

Simple examples

Let us explore the output:

```
> View(scores$data)
> View(scores$overall)
> text
[1] "I love this car" "I really love this car" "I do not love this car" "I really do
not love this car"
[5] "I barely love this car"
> scores$overall$sentiment_polarity
[1] 1 1 1 1 1
```

Simple examples

Let us now add negators:

```
> scores_not <- txt_sentiment(x = udpipe_text,term = "token",
+ polarity_terms = bing_dict,
+ polarity_negators = c("not","no"),
+ amplifier_weight = 0.8,
+ n_before = 1,
+ n_after = 0,
+ constrain = F)

> text
[1] "I love this car" "I really love this car" "I do not love this car" "I really do
not love this car"
[5] "I barely love this car"

> scores_not$overall$sentiment_polarity
[1] 1 1 -1 -1 1
```

Simple examples

Let us now add amplifiers and deamplifiers:

```
> scores_all <- txt_sentiment(x = udpipe_text,term = "token",
+ polarity_terms = bing_dict,
+ polarity_negators = c("not","no"),
+ polarity_amplifiers = c("very","really"),
+ polarity_deamplifiers = c("barely"),
+ amplifier_weight = 0.8,
+ n_before = 2,
+ n_after = 2,
+ constrain = F)

> text
[1] "I love this car" "I really love this car" "I do not love this car" "I really do
not love this car"
[5] "I barely love this car"
> scores_all$overall$sentiment_polarity
[1] 1.0 1.8 -1.0 -1.0 0.2
```

Simple examples

Let us now add amplifiers and deamplifiers:

```
> scores_all <- txt_sentiment(x = udpipe_text,term = "token",
+ polarity_terms = bing_dict,
+ polarity_negators = c("not","no"),
+ polarity_amplifiers = c("very","really"),
+ polarity_deamplifiers = c("barely"),
+ amplifier_weight = 0.8,
+ n_before = 2,
+ n_after = 2,
+ constrain = F)

> text
[1] "I love this car" "I really love this car" "I do not love this car" "I really do
not love this car"
[5] "I barely love this car"
> scores_all$overall$sentiment_polarity
[1] 1.0 1.8 -1.0 -1.0 0.2
```

Exercise. Explain how these values are computed.

Simple examples

Let us now change the values of n_before and n_after:

```
> scores_all_3 <- txt_sentiment(x = udpipe_text,term = "token",
+ polarity_terms = bing_dict,
+ polarity_negators = c("not","no"),
+ polarity_amplifiers = c("very","really"),
+ polarity_deamplifiers = c("barely"),
+ amplifier_weight = 0.8,
+ n_before = 3,
+ n_after = 3,
+ constrain = F)

> text
[1] "I love this car" "I really love this car" "I do not love this car" "I really do
not love this car"
[5] "I barely love this car"
> scores_all_3$overall$sentiment_polarity
[1] 1.0 1.8 -1.0 -0.2 0.2
```

Exercise for you

The following exercises refer to the case study based on Boston Airbnb comments.

Exercise 1

Consider Boston Airbnb reviews tokenized into bigrams.

1) Remove stopwords from bigrams. Hint: In order to remove stopwords, it is convenient to consider bigrams separated into two variables.
2) Which are the most common bigrams, after removing stopwords?
3) Which type of station is most commonly mentioned in the reviews?

Exercise 2

1) Tokenize Boston Airbnb reviews into trigrams.
2) Remove stopwords from trigrams.
3) Which are the most common trigrams? Does this information provide some insights?