Computational Communication Science 2 Week 2 - Lecture

Bottom-up approaches to text analysis: From preprocessing to vectorization

Anne Kroon
a.c.kroon@uva.nl, @annekroon

April 7, 2025

Digital Society Minor, University of Amsterdam

Today's Agenda

Recap

Review of key concepts

- **Text preprocessing:** Cleaning and preparing raw text for analysis.
- **Core techniques:**
  - tokenization
  - Stopword removal
  - Using built-in string methods for text cleaning
- **Methodological approaches:** Combining *bottom-up* (data-driven) and *top-down* (theory-driven) approaches.

Typical preprocessing steps

Preprocessing steps

tokenization How do we (best) split a sentence into tokens (terms, words)?

pruning How can we remove unneccessary words/ punctuation?

lemmatization and stemming How can we make sure that slight variations of the same word are not counted differently?

ngrams Neighbouring terms

Simple Tokenization using .split()

Tokenization: Splitting text into words or subwords is essential for many NLP tasks.

Simple Tokenization with .split(): Python's built-in 'split()' method is a simple way to break a text into tokens by spaces.

```
text = "This is an example sentence."
tokens = text.split()
print(tokens)  # Output: ['This', 'is', 'an', 'example', 'sentence.']
```

Advanced Tokenization with TreebankWordTokenizer

For more sophisticated tokenization that handles punctuation, contractions, etc., we use ‘TreebankWordTokenizer‘ from NLTK.

```
from nltk.tokenize import TreebankWordTokenizer

docs = ["This is an example sentence."]
tokens = [TreebankWordTokenizer().tokenize(d) for d in docs]
print(tokens)  # Output: [['This', 'is', 'an', 'example', 'sentence',
↪  '.']]
```

Understanding N-grams

- An n-gram is a sequence of *n* words treated as a single feature.
- Examples:
  - Unigrams (1-word units): “science”
  - Bigrams (2-word units): “data science”
  - Trigrams (3-word units): “machine learning model”
- **Why use n-grams?** Captures context and word relationships beyond single words.
- **Collocations:** Identifying n-grams that occur frequently and naturally together (e.g., "data science," "machine learning") helps uncover meaningful word patterns in text.

Generating N-grams with Python

Code Example: Generating n-grams using Python's NLTK library.

```python
import nltk
from nltk.util import ngrams
from nltk.tokenize import word_tokenize

# Sample text
text = "Machine learning is a powerful tool for data science."

# Tokenize the text
tokens = word_tokenize(text)

# Create bigrams and trigrams
bigrams = list(ngrams(tokens, 2))
trigrams = list(ngrams(tokens, 3))

# Count the frequency of n-grams
print("Bigrams:", bigrams)
print("Trigrams:", trigrams)
```

Identifying collocations

Code Example: Finding collocations using NLTK's BigramCollocationFinder.

```
from nltk.collocations import BigramCollocationFinder,
↪  BigramAssocMeasures
from collections import Counter

# Find bigram collocations using NLTK's BigramCollocationFinder
bigram_finder = BigramCollocationFinder.from_words(tokens)
bigram_collocations =
↪  bigram_finder.nbest(BigramAssocMeasures.likelihood_ratio, 5)

print("Collocations (Bigram):", bigram_collocations)

```

Stemming: Reducing Words to Their Root

- **Stemming** chops off word endings to reduce words to a common root.
- It's fast and simple, but may produce non-dictionary words.
- Example: "running" → "run", "flies" → "fli"

Code example (PorterStemmer):

```python
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()
words = ["running", "flies", "easily", "fairly"]
stems = [stemmer.stem(word) for word in words]

print(stems)
# Output: ['run', 'fli', 'easili', 'fairli']
```

Lemmatization: Getting the Dictionary Form

- **Lemmatization** maps words to their base (dictionary) form.
- More accurate than stemming, but slower and requires POS tagging for best results.
- Example: "better" → "good", "running" → "run"

Code Example (WordNetLemmatizer):

```python
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
from nltk import pos_tag, word_tokenize

lemmatizer = WordNetLemmatizer()
words = ["running", "flies", "better"]
lemmas = [lemmatizer.lemmatize(word) for word in words]

print(lemmas)
# Output: ['running', 'flies', 'better']
```

Building text representations: vectorizers

Building text representations: vectorizers

General idea

A text as a collections of word

Let us represent a string

```
t = "This this is is is a test test test"
# like this:
print(Counter(t.split()))
```

```
Counter({'is': 3, 'test': 3, 'This': 1, 'this': 1, 'a': 1})
```

Compared to the original string, this representation

- is less repetitive
- preserves word frequencies
- but does *not* preserve word order
- can be interpreted as a vector to calculate with (!!!)

Of course, still a lot of stuff to fine-tune... (for example, This/this)

From vector to matrix

If we do this for multiple texts, we can arrange the vectors in a table.

t1 = "This this is is is a test test test"

t2 = "This is an example"

	a	an	example	is	this	This	test
t1	1	0	0	3	1	1	3
t2	0	1	1	1	0	1	0

What can you do with such a matrix? Why would you want to represent a collection of texts in such a way?

What is a vectorizer

- Transforms a list of texts into a sparse (!) matrix (of word frequencies)
- Vectorizer needs to be "fitted" to the training data (learn which words (features) exist in the dataset and assign them to columns in the matrix)
- Vectorizer can then be re-used to transform other datasets

The cell entries: raw counts versus tf·idf scores

- In the example, we entered simple counts (the "term frequency")

But are all terms equally important?

The cell entries: raw counts versus tf·idf scores

- In the example, we entered simple counts (the "term frequency")
- But does a word that occurs in almost all documents contain much information?
- And isn't the presence of a word that occurs in very few documents a pretty strong hint?
- **Solution: Weigh by *the number of documents in which the term occurs at least once) (the "document frequency")***

⇒ we multiply the "term frequency" (tf) by the inverse document frequency (idf)

(usually with some additional logarithmic transformation and normalization applied, see https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html)

TF-IDF: Weighted Importance of Words

$$\text{tf-idf} = \text{tf}_{i,j} \times \log\left(\frac{N}{\text{df}_i}\right)$$

- $\text{tf}_{i,j}$ = term frequency of term i in document j
- df_i = number of documents containing the term i
- N = total number of documents in the corpus

- **Term Frequency (TF)**: Measures how often a term appears in a document. More frequent terms in a document receive a higher TF score.
- **Inverse Document Frequency (IDF)**: Downscales terms that appear in many documents and boosts those that are rare but important.

Is tf·idf always better?

It depends.

- Ultimately, it's an empirical question which works better ($\rightarrow$ machine learning)
- In many scenarios, "discounting" too frequent words and "boosting" rare words makes a lot of sense (most frequent words in a text can be highly un-informative)
- Beauty of raw tf counts, though: interpretability + describes document in itself, not in relation to other documents

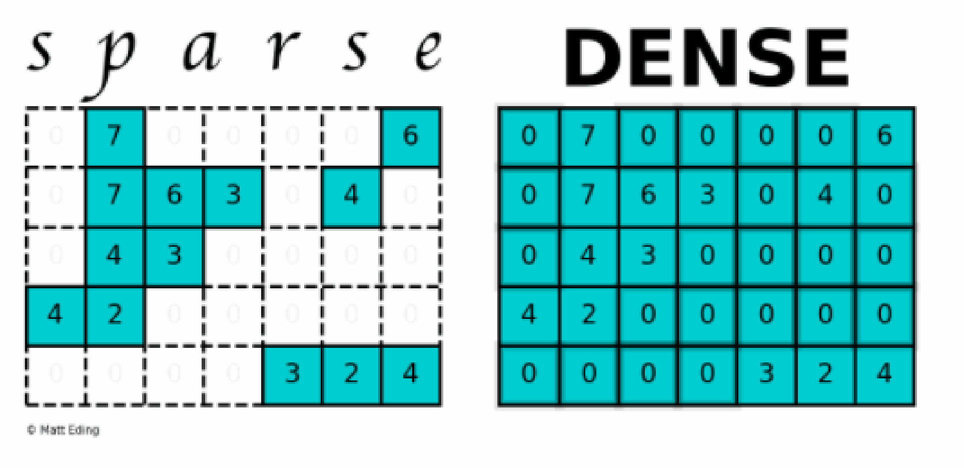
Different vectorizers

1. CountVectorizer (=simple word counts)
2. TfidfVectorizer (word counts ("term frequency") weighted by number of documents in which the word occurs at all ("inverse document frequency"))

Internal representations

Sparse vs dense matrices

- $\rightarrow$ tens of thousands of columns (terms), and one row per document
- Filling all cells is inefficient *and* can make the matrix too large to fit in memory (!!!)
- Solution: store only non-zero values with their coordinates! (sparse matrix)
- dense matrix (or dataframes) not advisable, only for toy examples



https://matteding.github.io/2019/04/25/sparse-matrices/

We learned in week 1 how to tokenize with a list comprehension (and that's often a good idea!).

```
from nltk.tokenize import TreebankWordTokenizer
tokens = [TreebankWordTokenizer().tokenize(d) for d in docs]
```

But what if we want to *directly* get a DTM instead of lists of tokens?

OK, good enough, perfect?

scikit-learn's CountVectorizer (default settings)

- applies lowercasing
- deals with punctuation etc. itself
- minimum word length > 1
- more technically, tokenizes using this regular expression: `r"(?u)\b\w\w+\b"` [1]

```
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
dtm_sparse = cv.fit_transform(docs)
```

[1] ?u = support unicode, \b = word boundary

OK, good enough, perfect?

CountVectorizer supports more

- stopword removal
- custom regular expression
- or even using an external tokenizer
- ngrams instead of unigrams

see
https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

Best of both worlds

Use the Count vectorizer with a NLTK-based external tokenizer! (see book)

This notebook might help to better your understanding of vectorizers!

Building text representations: vectorizers

Pruning

General idea

- Idea behind both stopword removal and tf·idf: too frequent words are uninformative
- (possible) downside stopword removal: a priori list, does not take empirical frequencies in dataset into account
- (possible) downside tf·idf: does not reduce number of features

Pruning: remove all features (tokens) that occur in less than X or more than X of the documents

CountVectorizer, only stopword removal

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
myvectorizer = CountVectorizer(stop_words=mystopwords)
```

CountVectorizer, better tokenization, stopword removal (pay attention that stopword list uses same tokenization!):

```
myvectorizer = CountVectorizer(tokenizer = TreebankWordTokenizer().tokenize, stop_words=mystopwords)
```

Additionally remove words that occur in more than 75% or less than $n = 2$ documents:

```
myvectorizer = CountVectorizer(tokenizer = TreebankWordTokenizer().tokenize, stop_words=mystopwords, max_df=.75, min_df=2)
```

All together: tf·idf, explicit stopword removal, pruning

```
myvectorizer = TfidfVectorizer(tokenizer = TreebankWordTokenizer().tokenize, stop_words=mystopwords, max_df=.75, min_df=2)
```

What is “best”? Which (combination of) techniques to use, and how to decide?

Exercise time: Word cloud

Let's put this into practice!

- Follow the instructions in the exercise material.
- You will create a word cloud from text data.
- Apply preprocessing techniques such as tokenization, stopword removal, and normalization.

Exercise Link: GitHub: Word cloud exercise *Think about how*

different preprocessing choices impact the resulting word cloud

From text to features

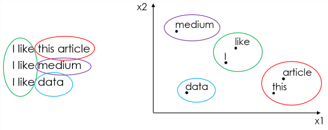
- Text needs to be transformed into numerical representations for computational analysis.
- Tokenization breaks text into meaningful units (words, phrases, n-grams).
- Frequency-based representations allow us to quantify text characteristics.

Introducing embeddings

Why move beyond TF-IDF?

Limitations of TF-IDF and CountVectorizer:

- Create **sparse**, high-dimensional representations.
- Ignore **semantic meaning**—similar words have different vectors.
- Cannot capture **contextual relationships**.



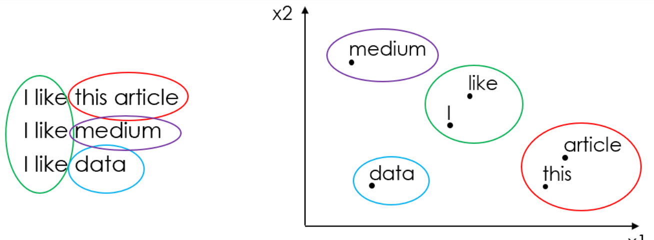
Solution: Word embeddings

Embeddings create **dense**, low-dimensional representations that retain meaning and context.

What are embeddings?

- Map words or documents into **continuous vector spaces**.
- Words with similar meanings have **similar vectors**.
- **Learned** from large text corpora using machine learning models.

Popular Examples: Word2Vec, GloVe, fastText, BERT embeddings.



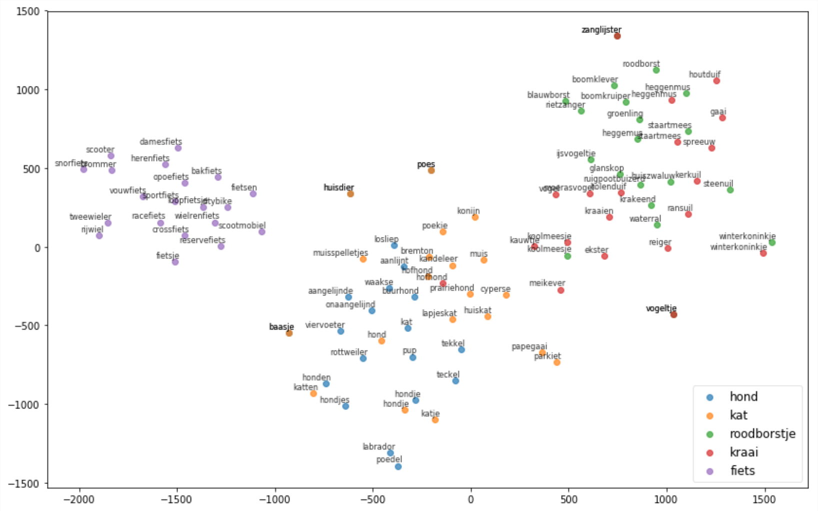
Introducing embeddings

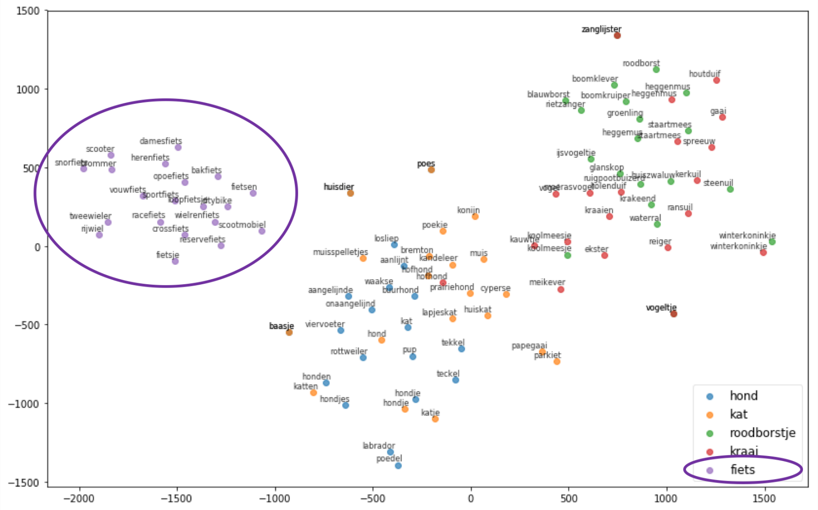
Word embeddings

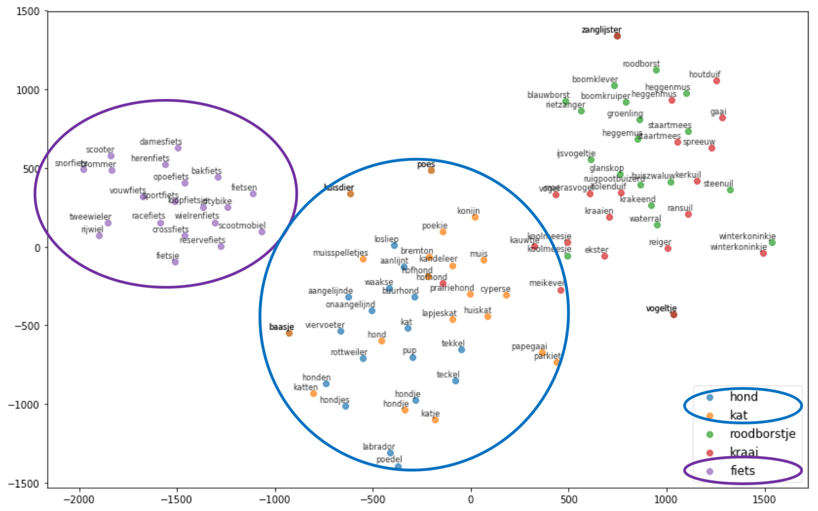
Understanding embeddings

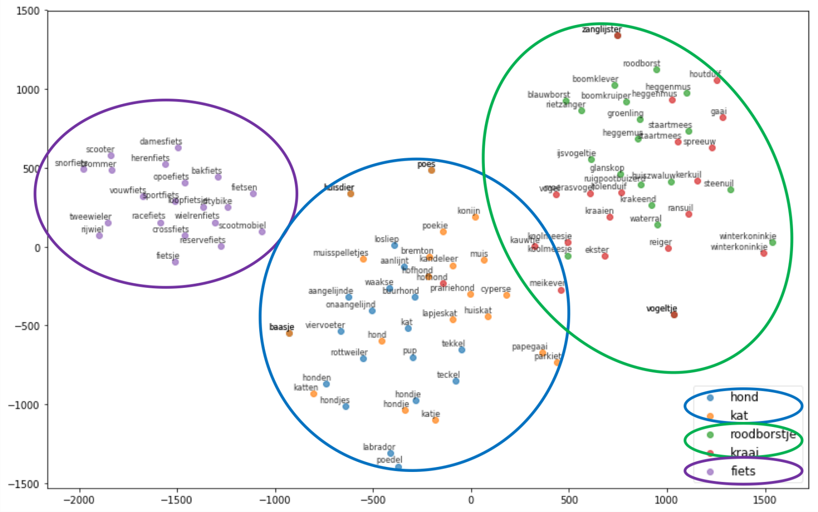
What are word embeddings?

- No technical details here, just the general idea
- Word embeddings help capture the meaning of text
- Word embeddings are low-dimensional vector representations that capture semantic meaning
- Used to be state-of-the-art in NLP (but now: contextualized embeddings, e.g., BERT or GPT)
- *"...a word is characterized by the company it keeps..."* (Firth, 1957)


zanglijster
roodborst
houtduif
boomklever
heggenmus
blauwborst
boomkruiper
rietzanger
groenling
gaai
staartmees
heggemus
spreeuw
ijsvogeltje
glanskop
kerkuil
steenuil
krakeend
kraaien
ransuil
waterral
winterkoninkje
reiger
ekster
koolmeesje
kauwtje
meikever
vogeltje
poes
huisdier
damesfiets
scooter
herenfiets
snorfiets
brommer
bakfiets
opoefiets
fietsen
vouwfiets
sportfiets
tweewieler
racefiets
wielrenfiets
rijwiel
crossfiets
scootmobiel
reservefiets
fietsje
konijn
poekie
losliep
muisspelletjes
bremton
kandeleer
muis
aanlijnt
hofhond
waakse
prairiehond
cyperse
aangelijnde
buurhond
onaangelijnd
lapjeskat
huiskat
baasje
viervoeter
kat
hond
tekkel
papegaai
parkiet
rottweiler
pup
teckel
honden
katten
hondje
hondjes
katje
labrador
poedel
hond
kat
roodborstje
kraai
fiets
1500
1000
500
0
−500
−1000
−1500
−2000
−1500
−1000
−500
0
500
1000
1500


1500
1000
500
0
−500
−1000
−1500
−2000
−1500
−1000
−500
0
500
1000
1500
zanglijster
roodborst
houtduif
boomklever
heggenmus
blauwborst
boomkruiper
rietzanger
groenling
gaai
staartmees
heggemus
spreeuw
ijsvogeltje
glanskop
poes
kerkuil
steenuil
huisdier
krakeend
kraaien
ransuil
waterral
konijn
poekie
winterkoninkje
koolmeesje
reiger
losliep
bremton
kandeleer
muis
ekster
muisspelletjes
aanlijnt
hofhond
waakse
meikever
prairiehond
cyperse
aangelijnde
buurhond
onaangelijnd
lapjeskat
huiskat
vogeltje
kat
baasje
viervoeter
hond
tekkel
papegaai
parkiet
rottweiler
pup
teckel
honden
katten
hondje
hondjes
katje
labrador
poedel
damesfiets
scooter
herenfiets
snorfiets
brommer
bakfiets
opoefiets
fietsen
vouwfiets
sportfiets
citybike
tweewieler
racefiets
wielrenfiets
rijwiel
crossfiets
scootmobiel
reservefiets
fietsje
hond
kat
roodborstje
kraai
fiets


1500
1000
500
0
−500
−1000
−1500
−2000
−1500
−1000
−500
0
500
1000
1500
zanglijster
roodborst
houtduif
boomklever
heggenmus
blauwborst
boomkruiper
rietzanger
groenling
gaai
staartmees
staartmees
spreeuw
ijsvogeltje
glanskop
huiszwaluw
kerkuil
ruigpootbuizerd
steenuil
tolenduif
krakeend
kraaien
ransuil
waterral
winterkoninkje
winterkoninkje
reiger
ekster
meikever
vogeltje
damesfiets
scooter
snorfiets
brommer
herenfiets
bakfiets
opoefiets
fietsen
vouwfiets
sportfiets
citybike
tweewieler
racefiets
wielrenfiets
scootmobiel
rijwiel
crossfiets
reservefiets
fietsje
poes
huisdier
konijn
poekie
losliep
bremton
muisspelletjes
aanlijnt
kandeleer
muis
hofhond
waakse
prairiehond
cyperse
aangelijnde
buurhond
onaangelijnd
lapjeskat
huiskat
baasje
viervoeter
kat
hond
tekkel
papegaai
parkiet
rottweiler
pup
teckel
honden
katten
hondjes
hondje
hondje
katje
labrador
poedel
hond
kat
roodborstje
kraai
fiets



- Explore word embeddings interactively using the TensorFlow Projector.
- Click here to access the TensorFlow Projector.

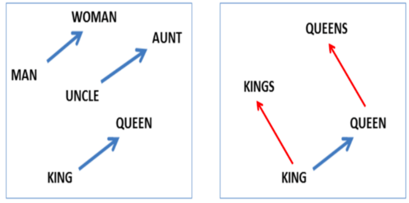
How embeddings work

- Each word (or document) is represented by a **vector** in a high-dimensional space.
- The model learns these vectors by predicting word co-occurrences in text.
- Example: **Word2Vec** uses a neural network to predict surrounding words.

Example: Word similarity with embeddings

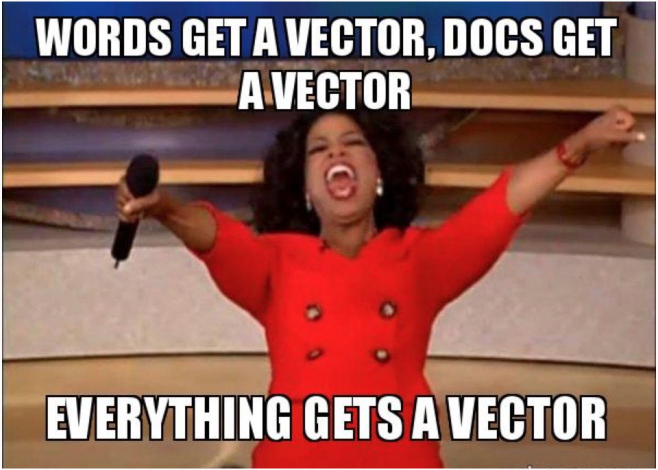
Word2Vec analogy:

- `vec("king") - vec("man") + vec("woman") = vec("queen")`
- Captures **semantic relationships** automatically.
- Unlike TF-IDF, embeddings **understand meaning**.



Example: Getting word embeddings with spaCy

Try it out yourself.. Visualize your embeddings



Embeddings in communication science

Applications:

- **Sentiment analysis:** Understanding audience reactions on social media (Rudkowsky et al., 2018)
- **Topic modeling**: cross-lingual topics
- **Recommender Systems:** Suggesting content based on user preferences (Loecherbach and Trilling, 2020)
- **Text similarity:** Measuring the similarity between texts (Brinberg and Ram, 2021)

Note: We will discuss this next week.

Using pretrained embeddings

Why use pretrained models?

- Trained on massive datasets (Google News, Wikipedia, etc.).
- Capture **rich linguistic structures**.
- Reduce training time and improve performance.

Popular choices:

- Word2Vec (Google News)
- GloVe (Common Crawl, Wikipedia)
- BERT (contextualized embeddings)
- fastText (subword information)

Why Use Embeddings?

- Capture **semantic meaning** of words.
- Handle **synonyms** and **related words** effectively.
- Work well in NLP applications: **text classification, clustering, sentiment analysis**.
- Used in communication science for **media analysis, misinformation detection, and social network studies**.

Key takeaways

- Traditional methods (TF-IDF) are **limited in capturing meaning**.
- Word embeddings create **dense vectors** that capture relationships.
- Pretrained models like **Word2Vec, GloVe, and BERT** help analyze text effectively.
- Embeddings are **widely used** in NLP and communication science.

Thank you!!

Thank you for your attention!

- Questions? Comments?

References i

References

Brinberg, M., & Ram, N. (2021).**Do new romantic couples use more similar language over time? Evidence from intensive longitudinal text messages.** *Journal of Communication*, *71*(3), 454–477. https://doi.org/10.1093/joc/jqab012

Loecherbach, F., & Trilling, D. (2020).**3bij3 – Developing a framework for researching recommender systems and their effects.** *Computational Communication Research*, *2*(1), 53–79. https://doi.org/10.5117/ccr2020.1.003.loec

References ii

Rudkowsky, E., Haselmayer, M., Wastian, M., Jenny, M., Emrich, Š., & Sedlmair, M. (2018).**More than bags of words: Sentiment analysis with word embeddings.** *Communication Methods and Measures*, *12*(2-3), 140–157.