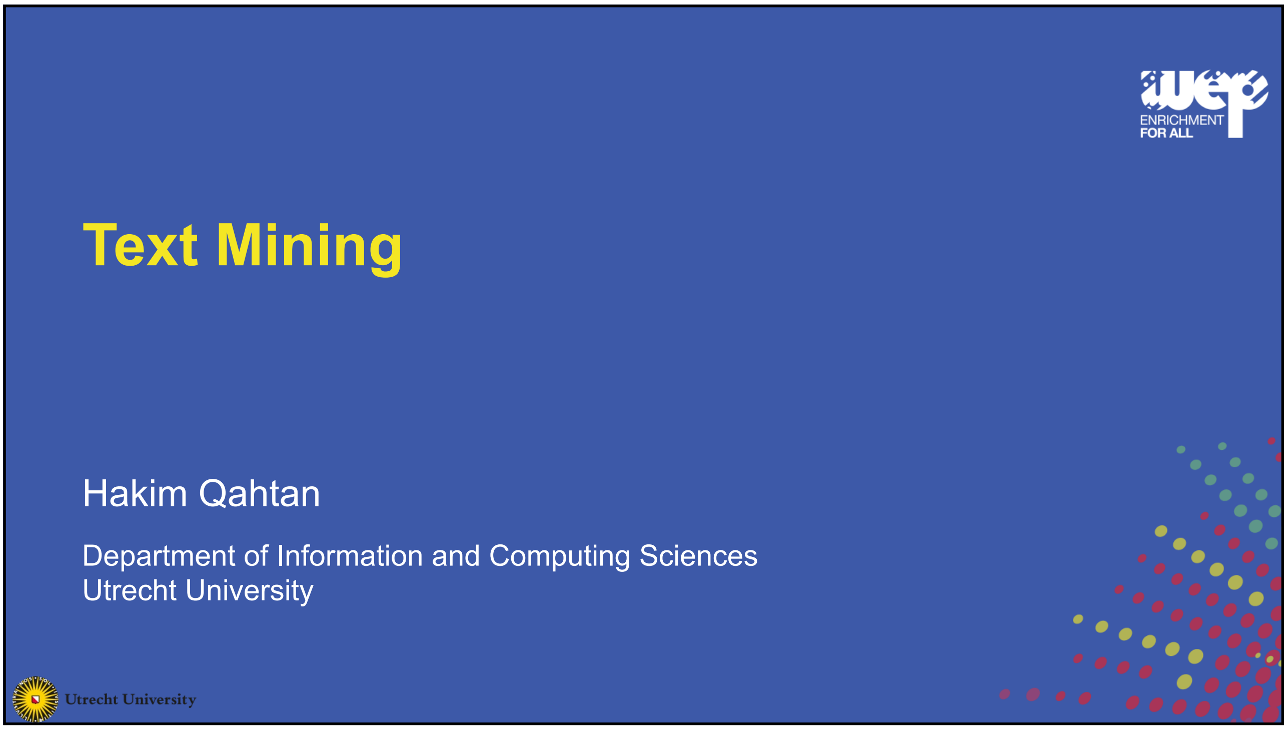
ENRICHMENT FOR ALL

Digital Adventure Ride to the Future

7 – 18 January, 2024

Utrecht University

1



ENRICHMENT FOR ALL

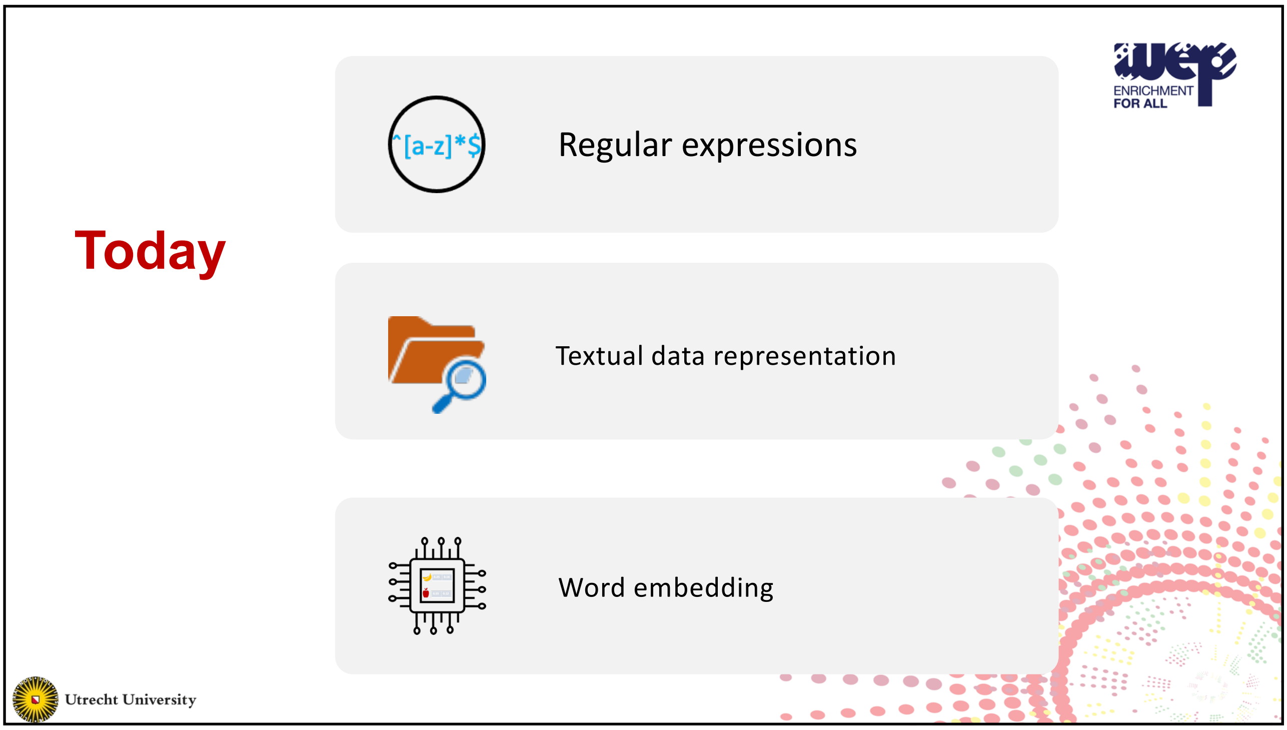
Text Mining

Hakim Qahtan

Department of Information and Computing Sciences
Utrecht University

Utrecht University

2



Today

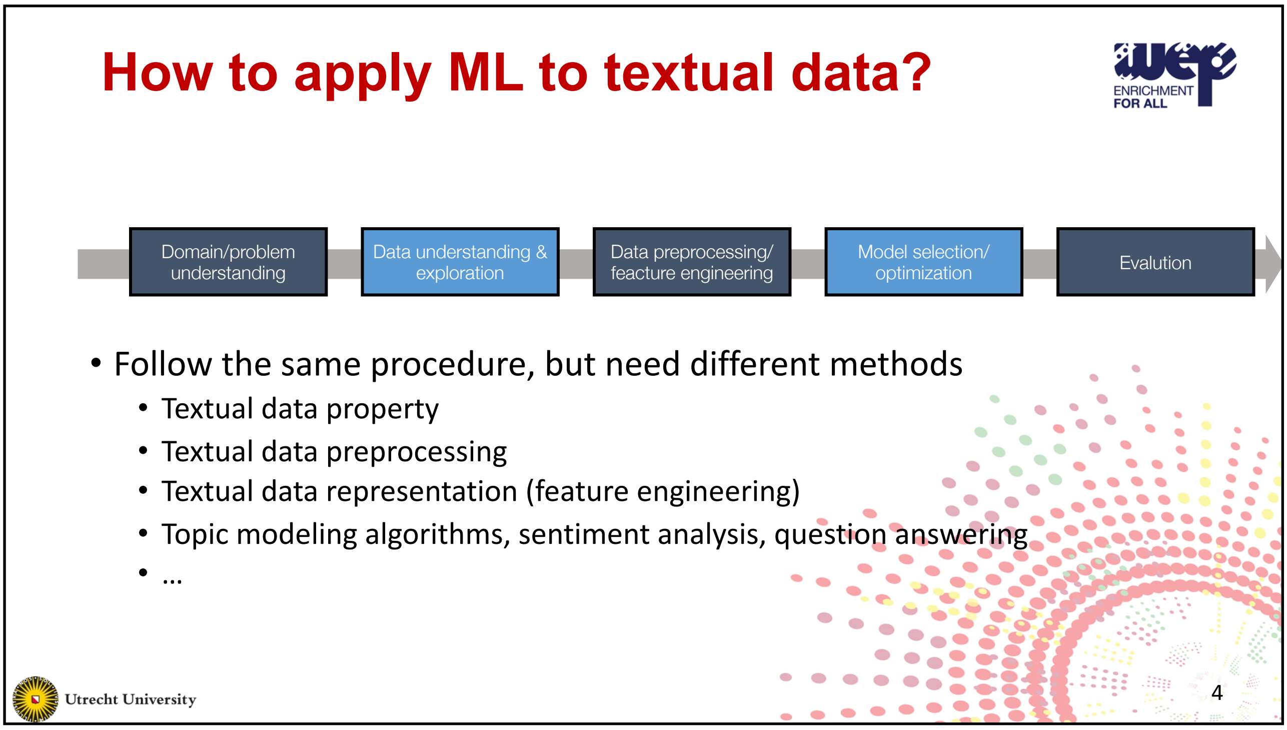
^[a-z]*$ Regular expressions

Textual data representation

Word embedding

Utrecht University

3



How to apply ML to textual data?

Domain/problem understanding → Data understanding & exploration → Data preprocessing/ feacture engineering → Model selection/ optimization → Evalution

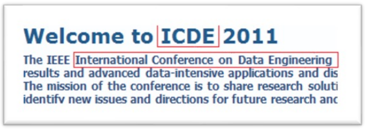
- Follow the same procedure, but need different methods
  - Textual data property
  - Textual data preprocessing
  - Textual data representation (feature engineering)
  - Topic modeling algorithms, sentiment analysis, question answering
  - …

Utrecht University

4

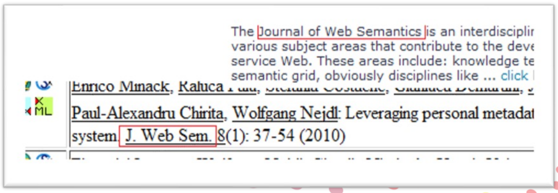
Language is hard

- Text variations:
  - Misspellings, acronyms, transformations, abbreviations, etc.
- Context dependency
  - “You have very nice shoes”
- Figurative language
  - “He has a heart of stone”
- Same words with different meanings
  - “sanction”



Welcome to ICDE 2011

The IEEE International Conference on Data Engineering
results and advanced data-intensive applications and dis
The mission of the conference is to share research soluti
identifv new issues and directions for future research anc



The Journal of Web Semantics is an interdisciplir
various subject areas that contribute to the deve
service Web. These areas include: knowledge te
semantic grid, obviously disciplines like ... click

Enrico Minack, Raluca

Paul-Alexandru Chirita, Wolfgang Nejdl: Leveraging personal metadat
system. J. Web Sem. 8(1): 37-54 (2010)

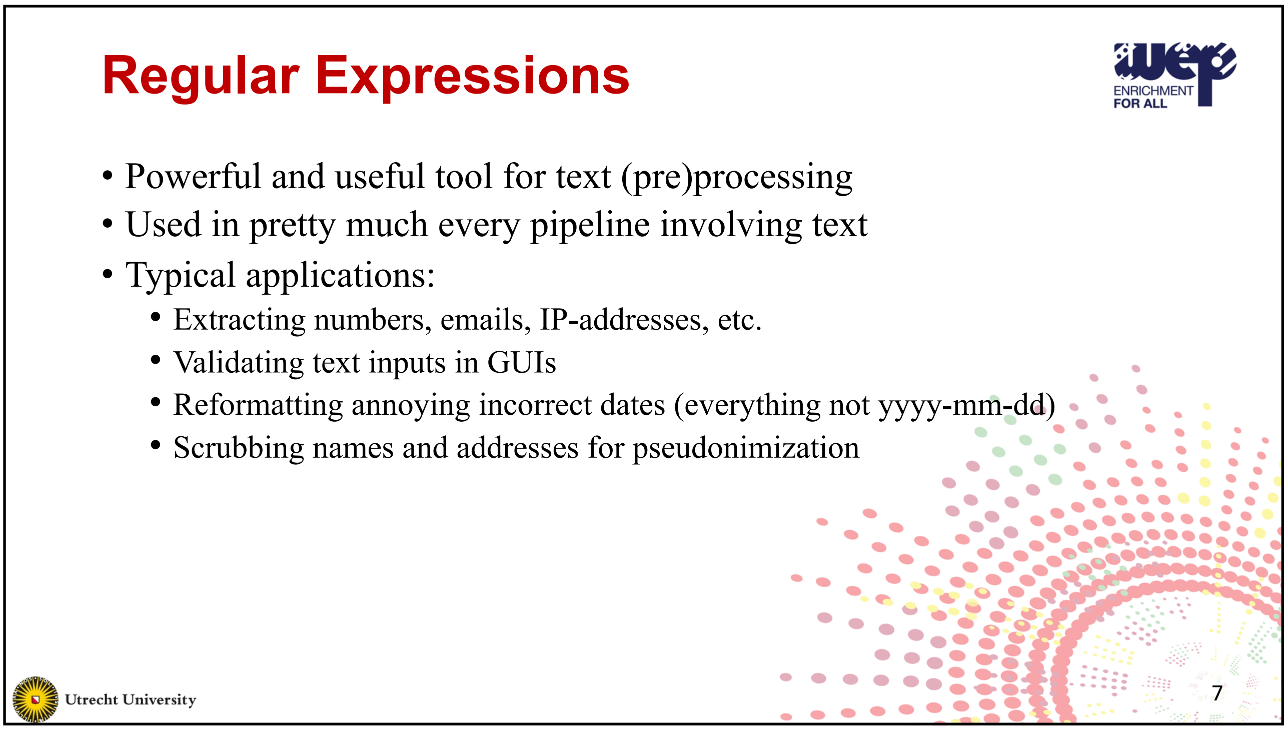
Utrecht University

5

Regular Expressions

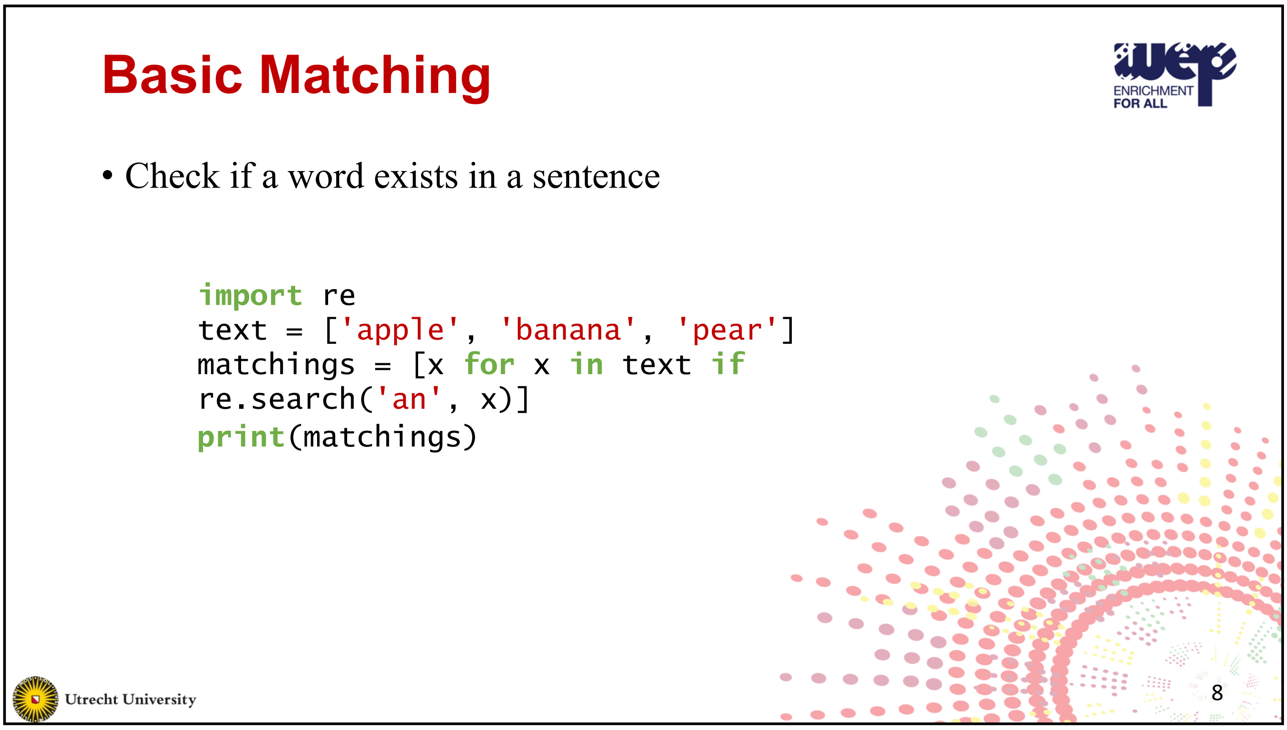
Utrecht University

6



Regular Expressions

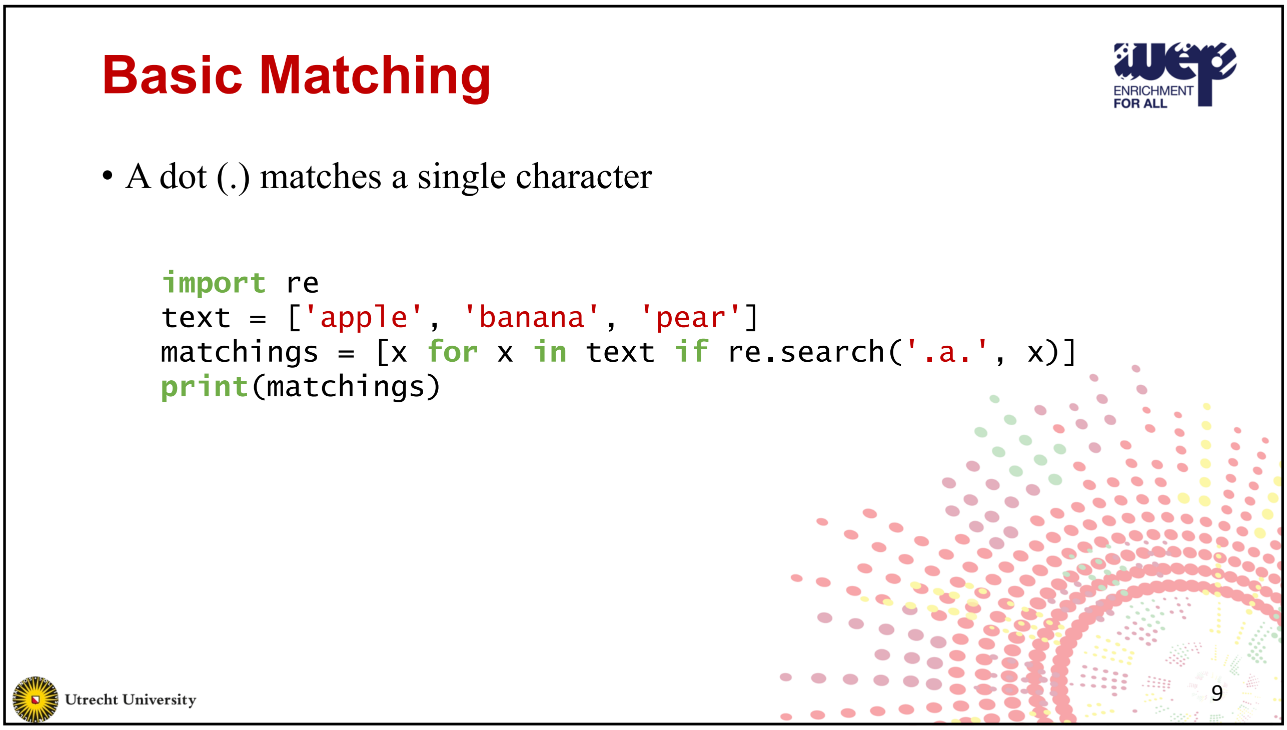
- Powerful and useful tool for text (pre)processing
- Used in pretty much every pipeline involving text
- Typical applications:
  - Extracting numbers, emails, IP-addresses, etc.
  - Validating text inputs in GUIs
  - Reformatting annoying incorrect dates (everything not yyyy-mm-dd)
  - Scrubbing names and addresses for pseudonimization



Basic Matching

- Check if a word exists in a sentence

```
import re
text = ['apple', 'banana', 'pear']
matchings = [x for x in text if
re.search('an', x)]
print(matchings)
```

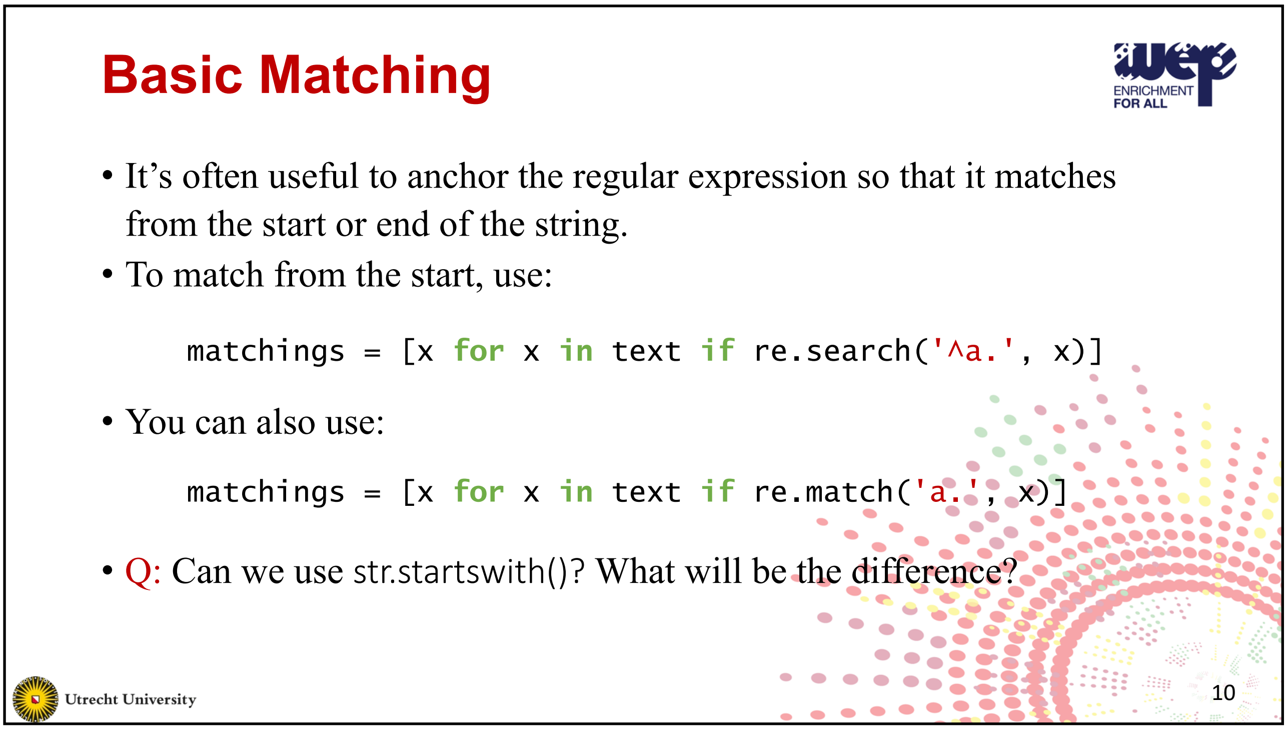


Basic Matching

- A dot (.) matches a single character

```
import re
text = ['apple', 'banana', 'pear']
matchings = [x for x in text if re.search('.a.', x)]
print(matchings)
```

9



Basic Matching

- It’s often useful to anchor the regular expression so that it matches from the start or end of the string.
- To match from the start, use:

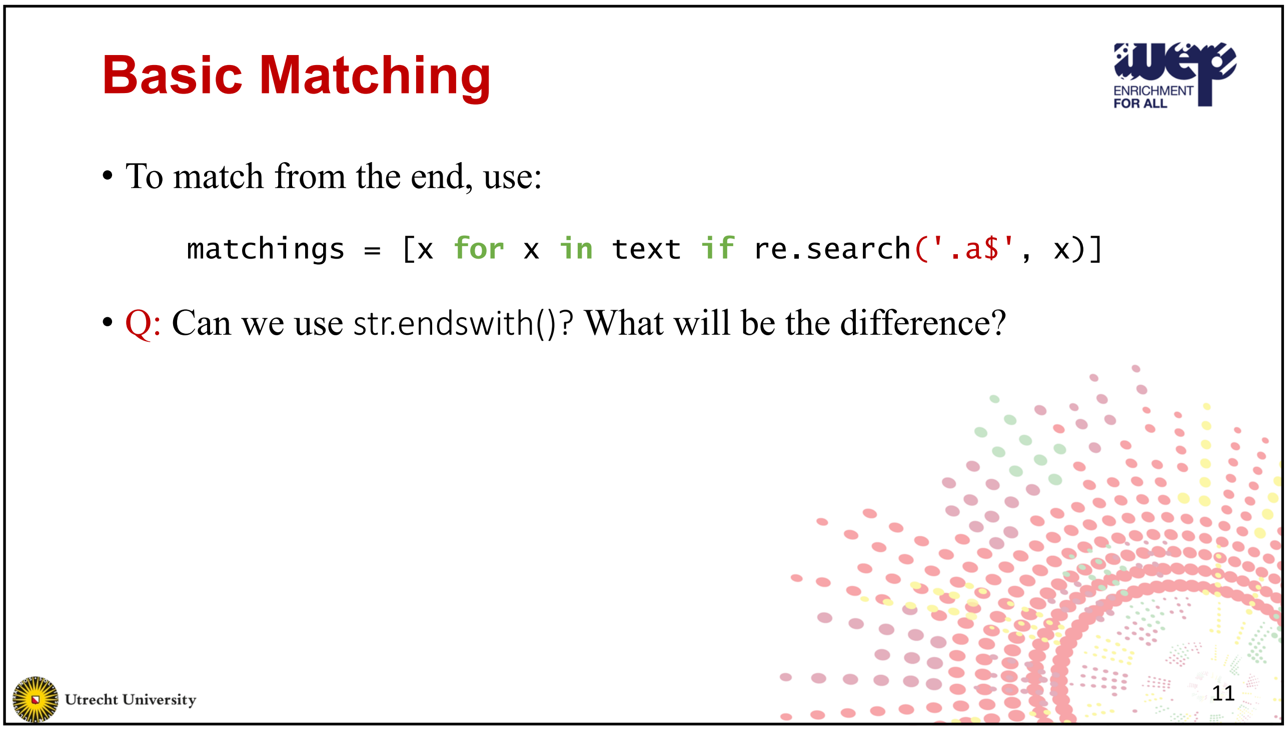
```
matchings = [x for x in text if re.search('^a.', x)]
```

- You can also use:

```
matchings = [x for x in text if re.match('a.', x)]
```

- Q: Can we use str.startswith()? What will be the difference?

10



Basic Matching

- To match from the end, use:

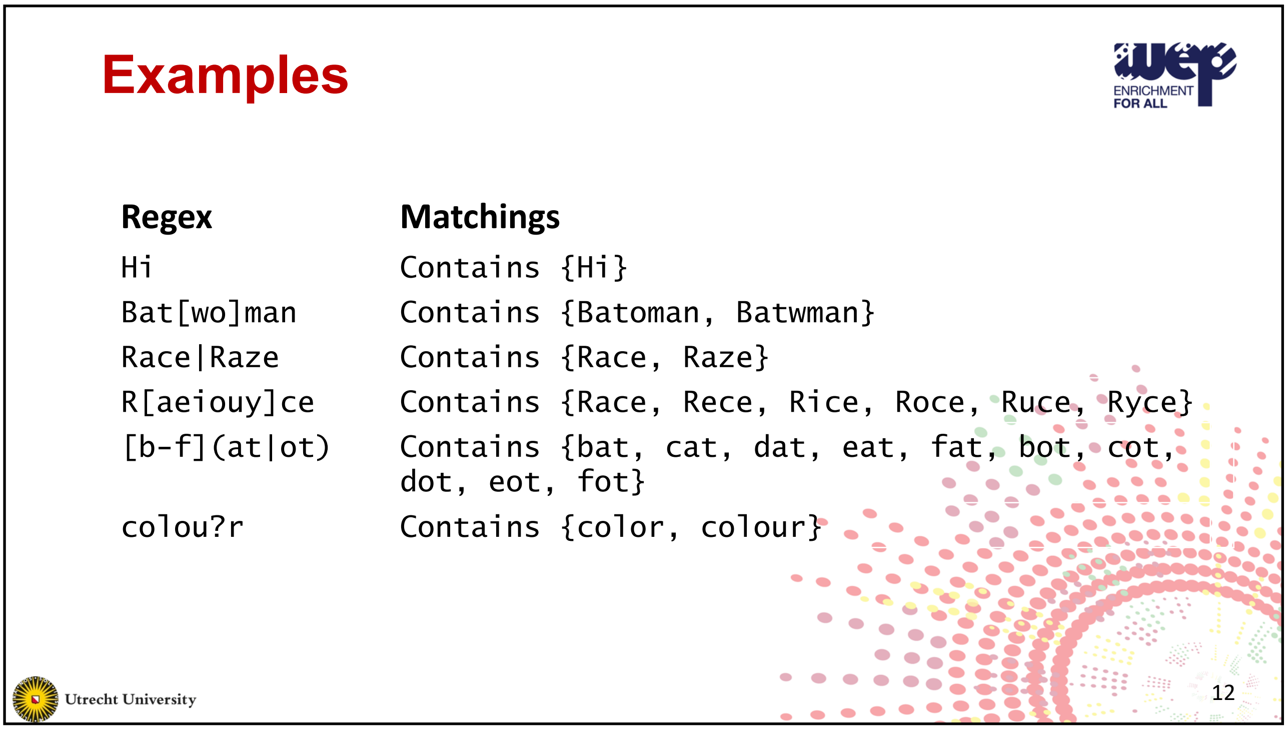
```
matchings = [x for x in text if re.search('.a$', x)]
```

- Q: Can we use str.endswith()? What will be the difference?

Utrecht University

11

11



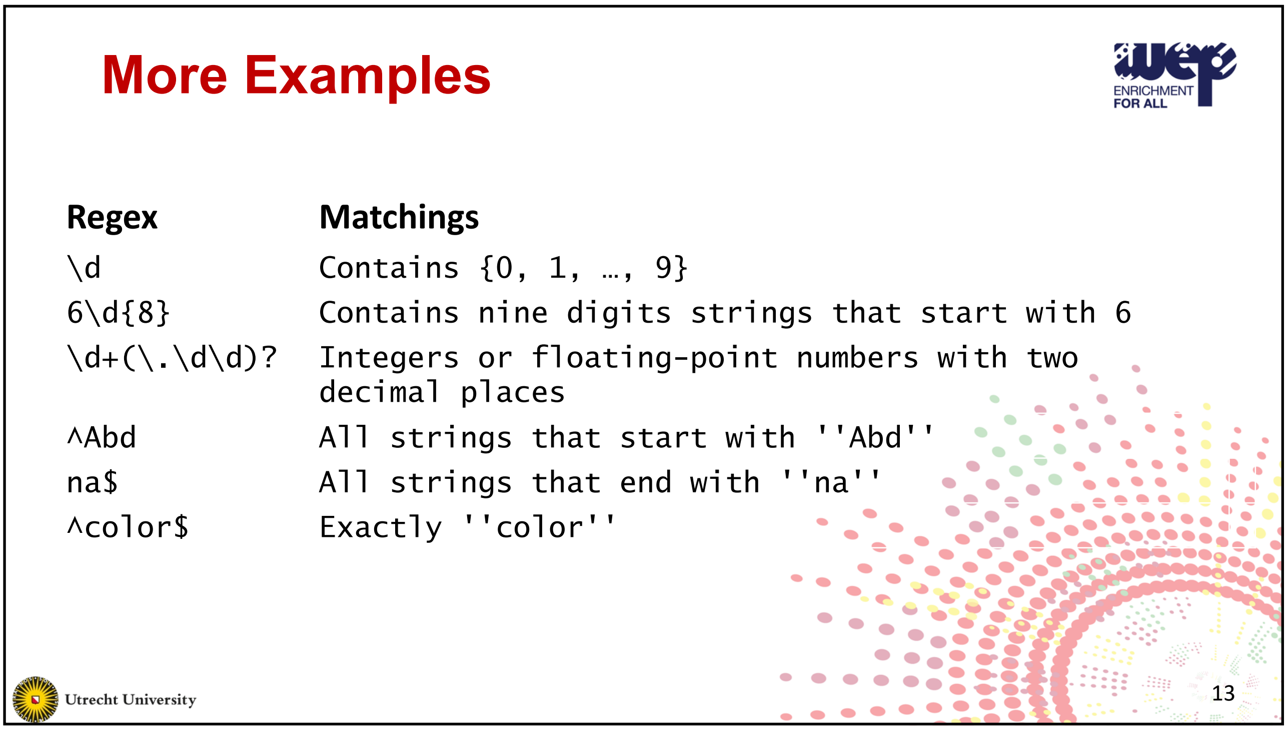
Examples

Regex	Matchings
`Hi`	`Contains {Hi}`
`Bat[wo]man`	`Contains {Batoman, Batwman}`
`Race|Raze`	`Contains {Race, Raze}`
`R[aeiouy]ce`	`Contains {Race, Rece, Rice, Roce, Ruce, Ryce}`
`[b-f](at|ot)`	`Contains {bat, cat, dat, eat, fat, bot, cot, dot, eot, fot}`
`colou?r`	`Contains {color, colour}`

Utrecht University

12

12



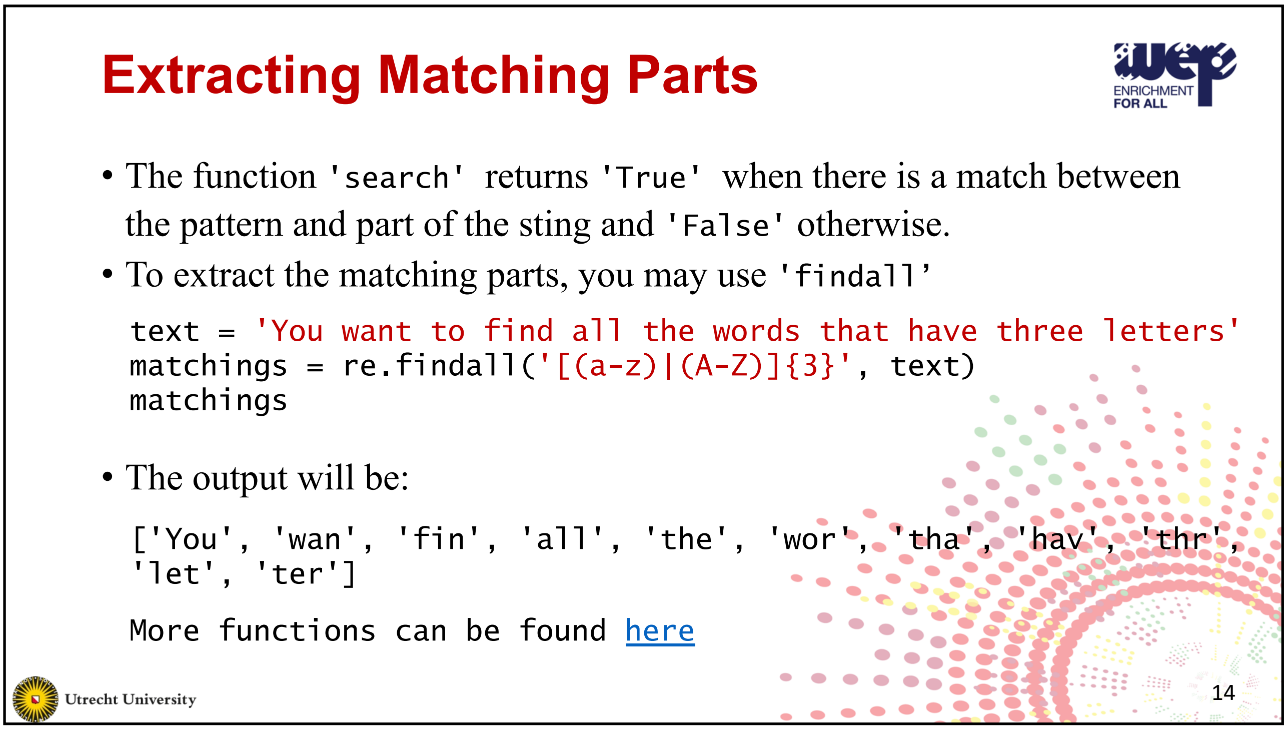
More Examples

Regex	Matchings
`\d`	Contains {0, 1, …, 9}
`6\d{8}`	Contains nine digits strings that start with 6
`\d+(\.\d\d)?`	Integers or floating-point numbers with two decimal places
`^Abd`	All strings that start with ''Abd''
`na$`	All strings that end with ''na''
`^color$`	Exactly ''color''

Utrecht University

13

13



Extracting Matching Parts

- The function `'search'` returns `'True'` when there is a match between the pattern and part of the sting and `'False'` otherwise.
- To extract the matching parts, you may use `'findall'`

```
text = 'You want to find all the words that have three letters'
matchings = re.findall('[(a-z)|(A-Z)]{3}', text)
matchings
```

- The output will be:

```
['You', 'wan', 'fin', 'all', 'the', 'wor', 'tha', 'hav', 'thr',
'let', 'ter']
```

More functions can be found here

Utrecht University

14

14

Textual Data Preprocessing

Utrecht University

15

Definitions

- **Document:** a sequence of words and punctuation, following the grammatical rules of a language
- **Term:** usually a word, but can be a word-pair or phrase
- **Corpus:** a collection of documents
- **Lexicon:** set of all unique words in a corpus

Utrecht University

16

16

Textual Data Preprocessing Tasks

- TEXT = ``Morning and afternoon are parts of the day.”
- Typical steps:
  - Tokenization (``afternoon”, `and’, ``Morning”, `are’, ``parts”, `of’, `the’, ``day”)
  - Stemming (``parts” ``part”) or Lemmatization (``are” ``is”)
  - Lowercasing (`` Morning” ``morning”)
  - Stop-word removal (``Morning and afternoon are parts of the day.”)
  - Punctuation removal (``Morning and afternoon are parts of the day.”)
  - Number removal (``day 3” ``part 2”)
  - Spell correction (``aftrnoon” ``afternoon”)
- You may not need to perform all of these tasks

Utrecht University

17

17

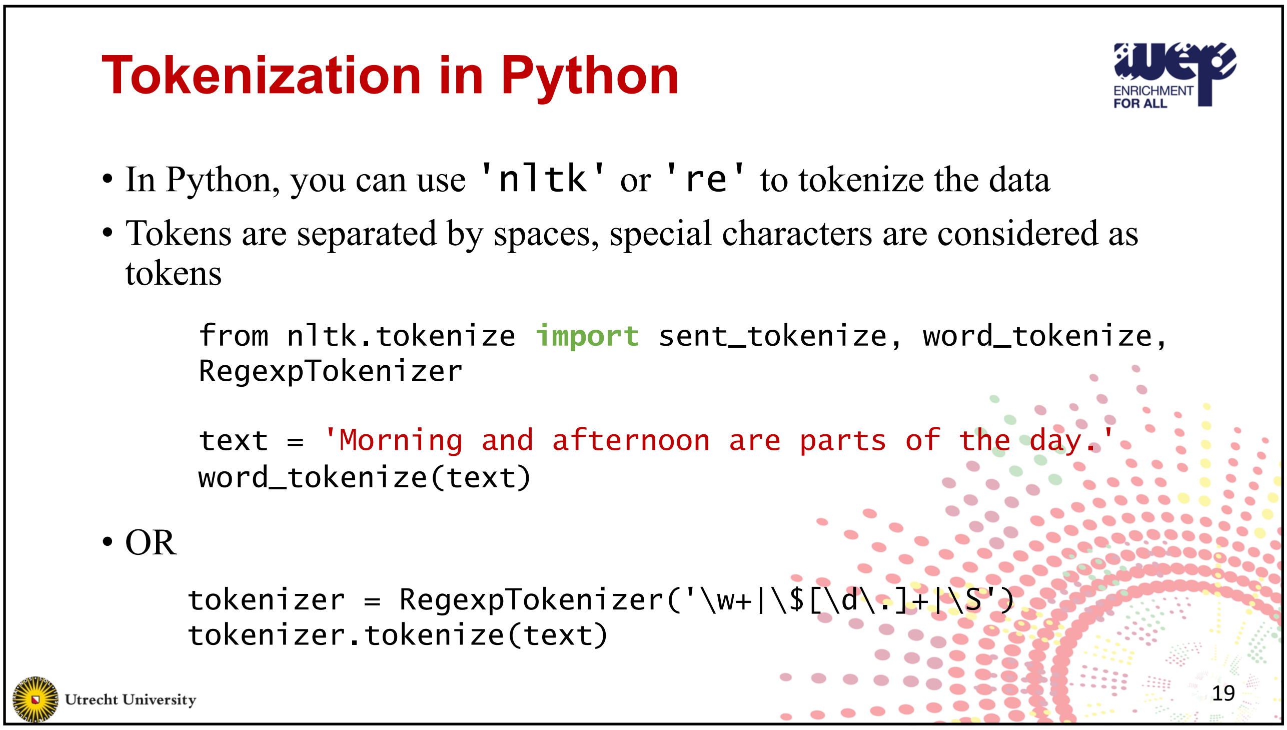
Text Preprocessing – Tokenization

- Convert a sentence into a sequence of tokens, i.e., words.
- Tokenizing English sentences is quite straightforward:
  - Just use spaces and punctuation as boundaries.
- Potentially many exceptions
- Examples:
  - Tom’s ? a Possessive ending? Or Tom is? Or Tom has?
  - Medicine is not nearly as evidence-based as we'd like
- The assumption that words are separated by non-letters is not always true
  - it is useful in practice
- The assumption that a word equals a token is not always true
  - **New York** is a **U.S.** city

Utrecht University

18

18



Tokenization in Python

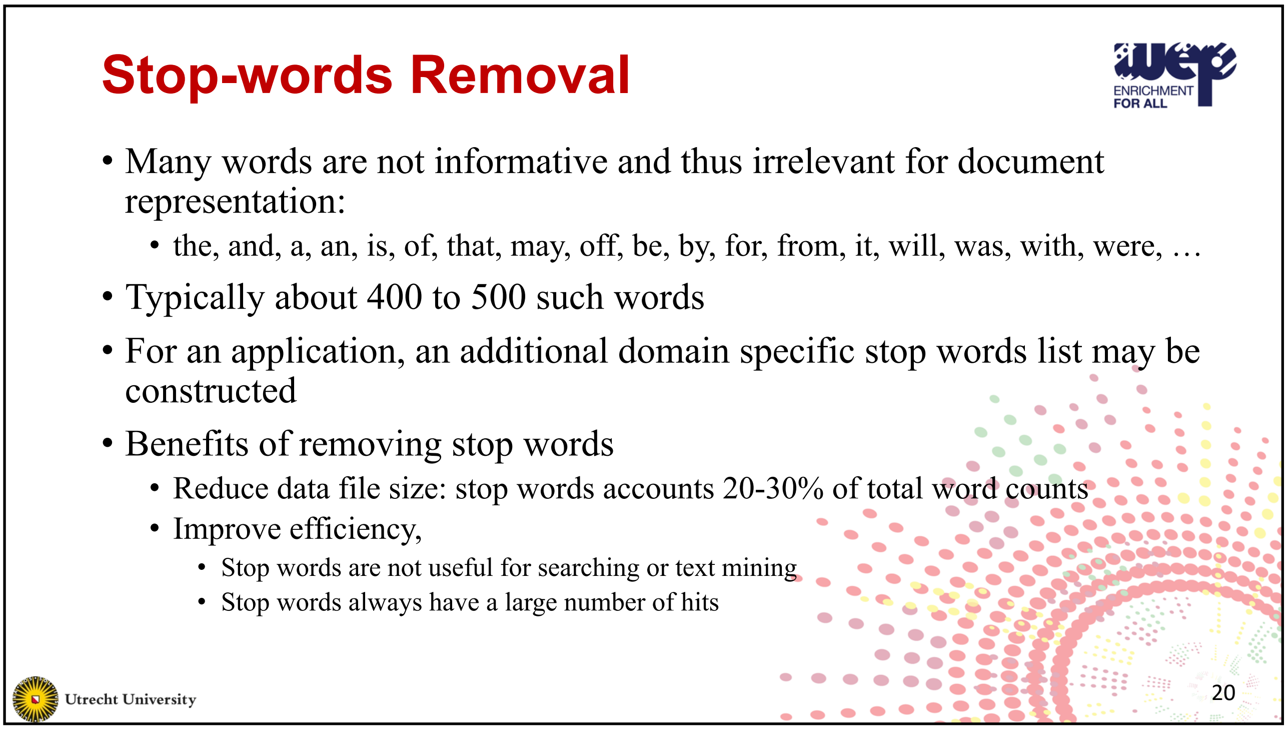
- In Python, you can use `'nltk'` or `'re'` to tokenize the data
- Tokens are separated by spaces, special characters are considered as tokens

```
from nltk.tokenize import sent_tokenize, word_tokenize,
RegexpTokenizer

text = 'Morning and afternoon are parts of the day.'
word_tokenize(text)
```

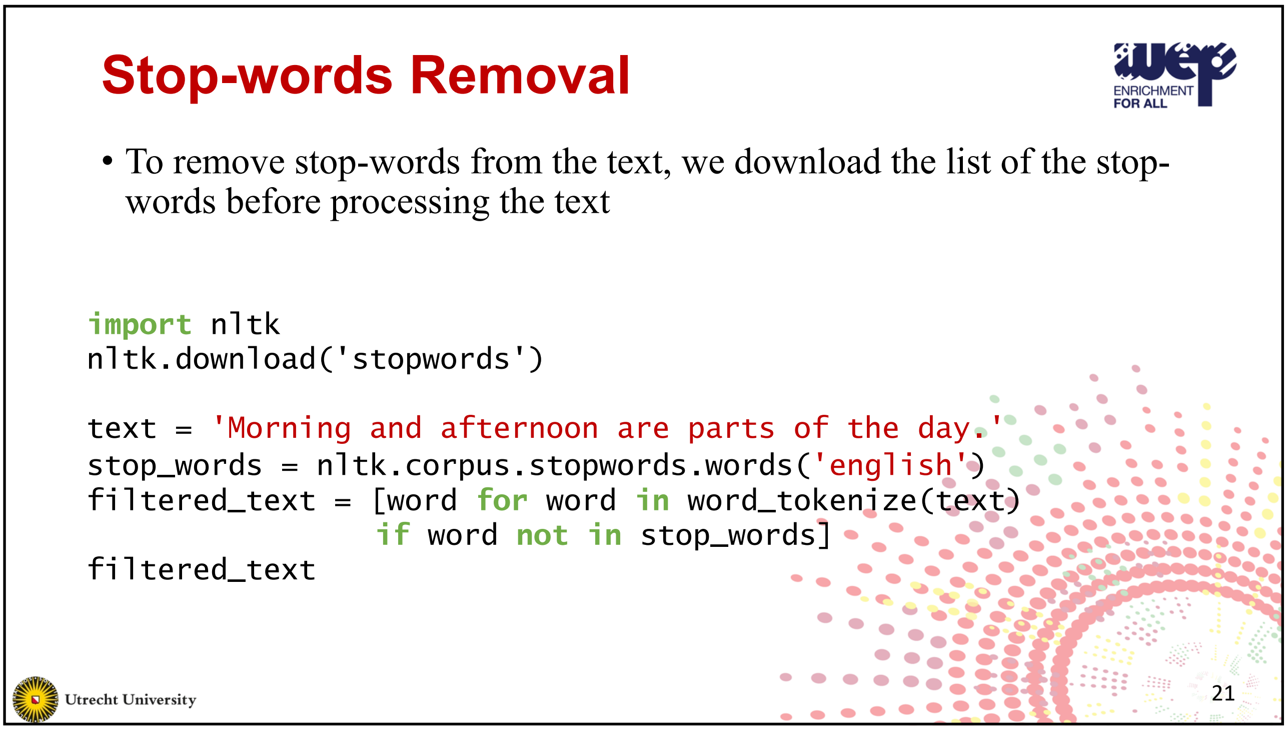
- OR

```
tokenizer = RegexpTokenizer('\w+|\$[\d\.]+|\S')
tokenizer.tokenize(text)
```



Stop-words Removal

- Many words are not informative and thus irrelevant for document representation:
  - the, and, a, an, is, of, that, may, off, be, by, for, from, it, will, was, with, were, …
- Typically about 400 to 500 such words
- For an application, an additional domain specific stop words list may be constructed
- Benefits of removing stop words
  - Reduce data file size: stop words accounts 20-30% of total word counts
  - Improve efficiency,
    - Stop words are not useful for searching or text mining
    - Stop words always have a large number of hits



Stop-words Removal

- To remove stop-words from the text, we download the list of the stop-words before processing the text

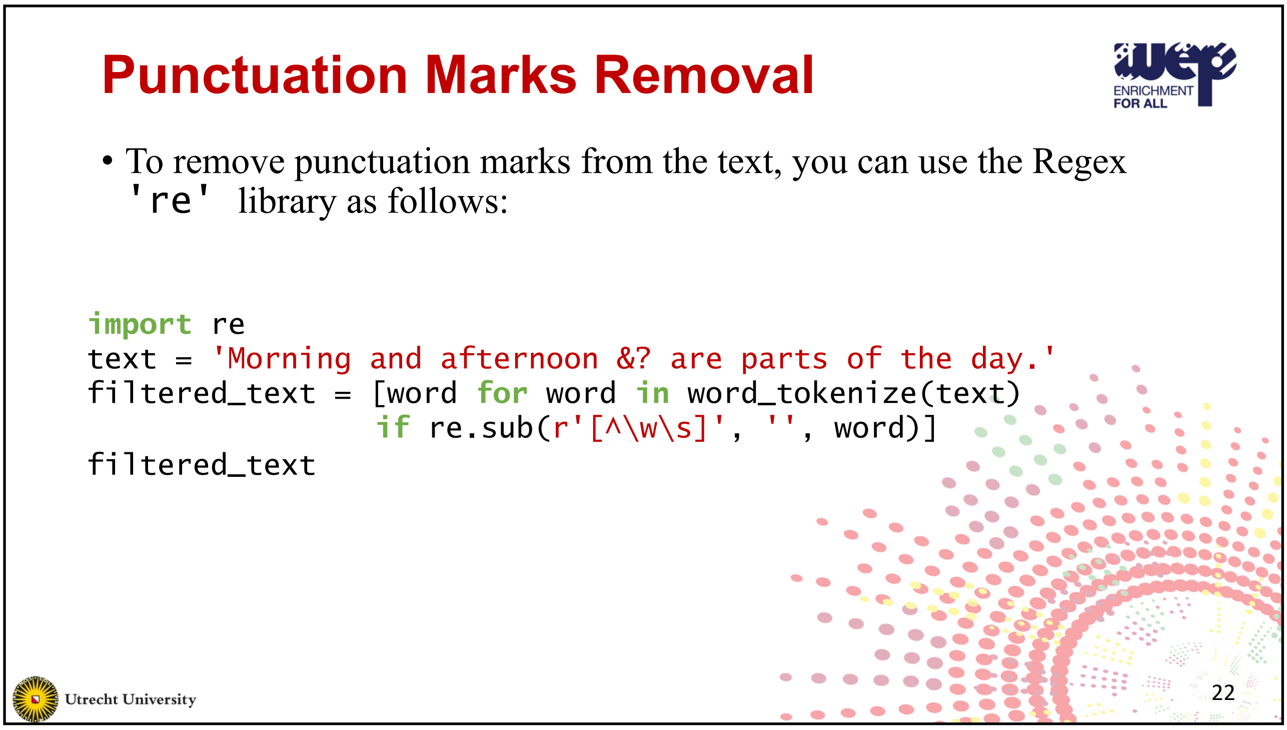
```
import nltk
nltk.download('stopwords')

text = 'Morning and afternoon are parts of the day.'
stop_words = nltk.corpus.stopwords.words('english')
filtered_text = [word for word in word_tokenize(text)
                 if word not in stop_words]
filtered_text
```

Utrecht University

21

21



Punctuation Marks Removal

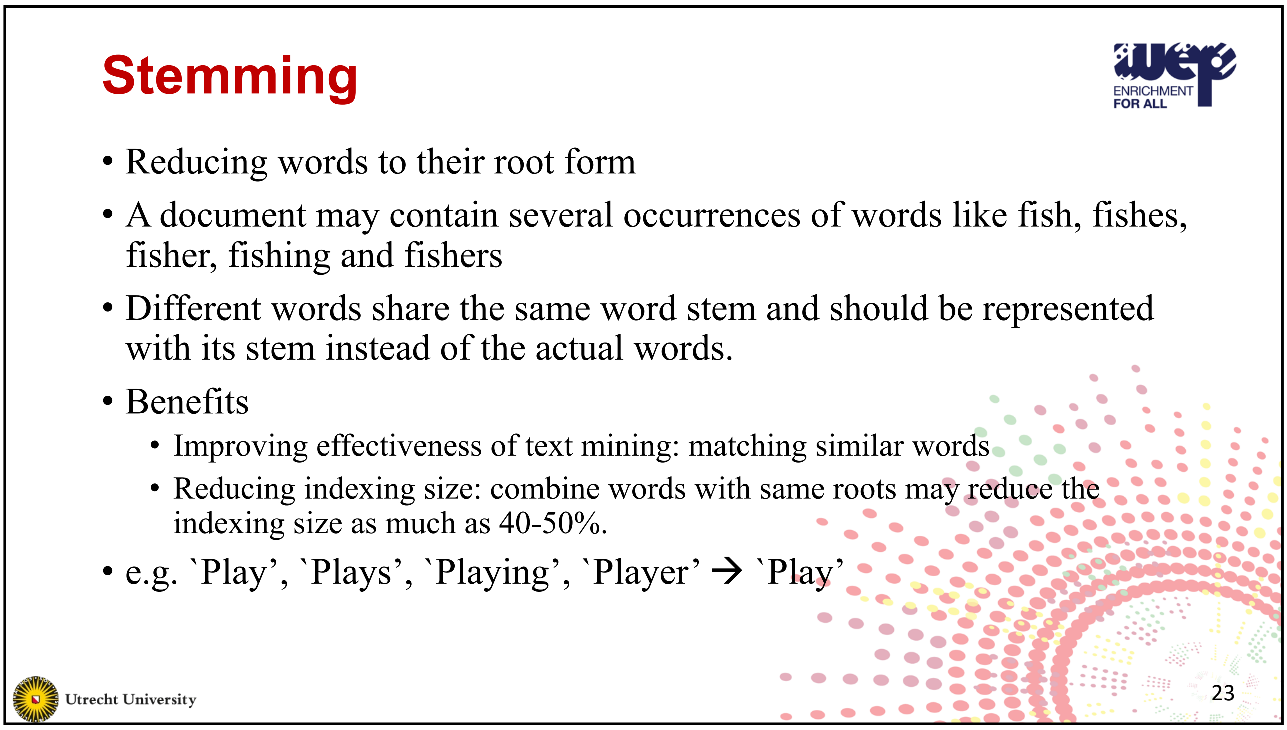
- To remove punctuation marks from the text, you can use the Regex `'re'` library as follows:

```
import re
text = 'Morning and afternoon &? are parts of the day.'
filtered_text = [word for word in word_tokenize(text)
                 if re.sub(r'[^\w\s]', '', word)]
filtered_text
```

Utrecht University

22

22



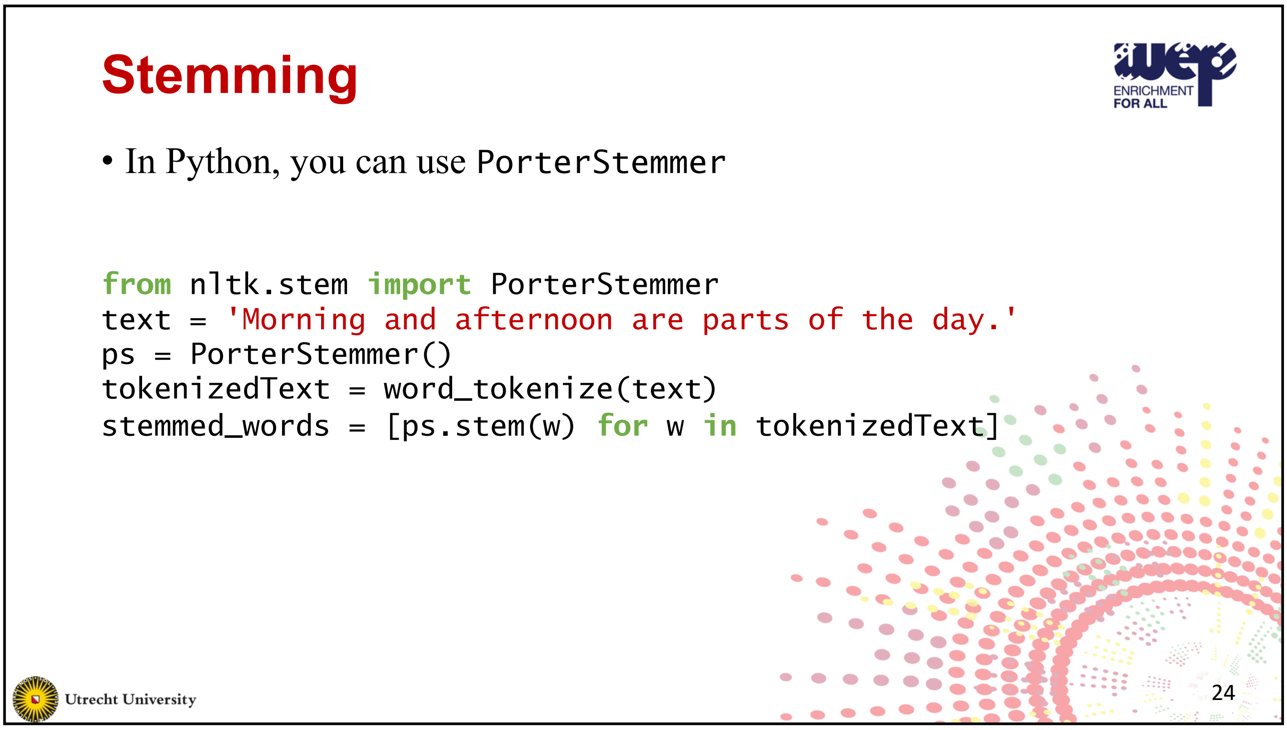
Stemming

- Reducing words to their root form
- A document may contain several occurrences of words like fish, fishes, fisher, fishing and fishers
- Different words share the same word stem and should be represented with its stem instead of the actual words.
- Benefits
  - Improving effectiveness of text mining: matching similar words
  - Reducing indexing size: combine words with same roots may reduce the indexing size as much as 40-50%.
- e.g. \`Play’, \`Plays’, \`Playing’, \`Player’ → \`Play’

Utrecht University

23

23



Stemming

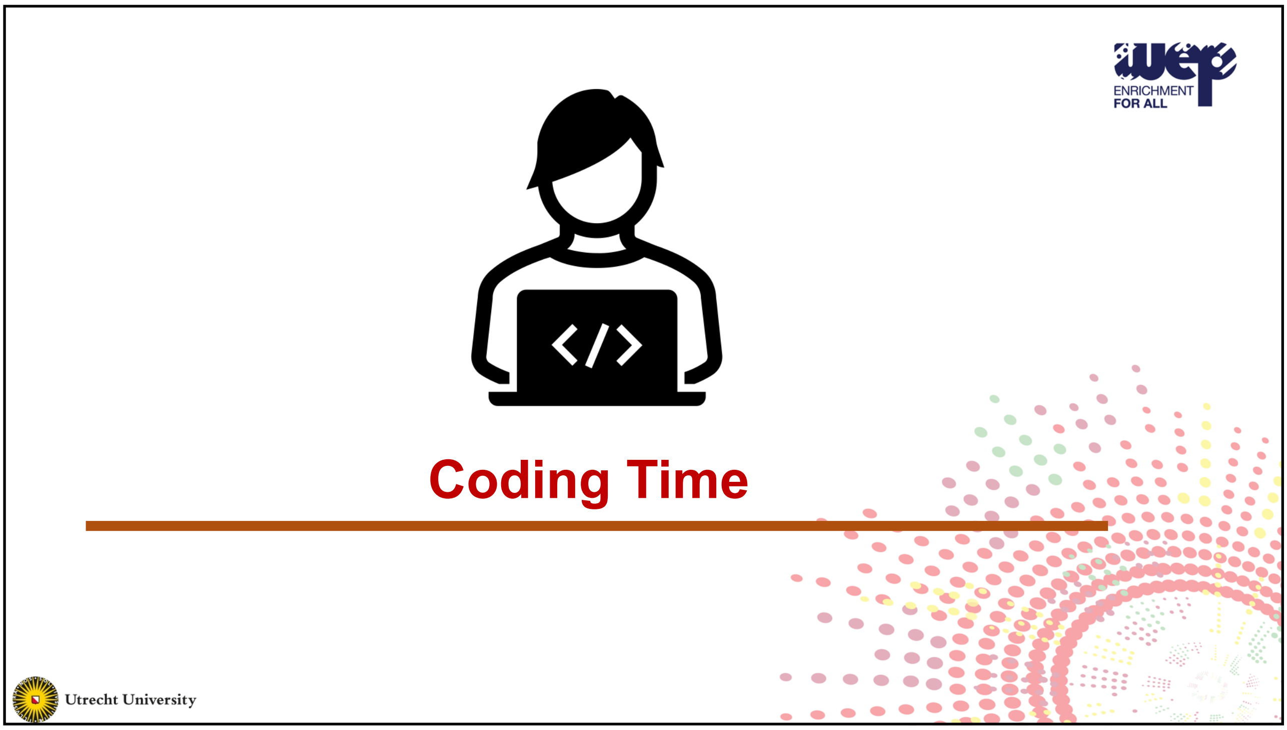
- In Python, you can use `PorterStemmer`

```
from nltk.stem import PorterStemmer
text = 'Morning and afternoon are parts of the day.'
ps = PorterStemmer()
tokenizedText = word_tokenize(text)
stemmed_words = [ps.stem(w) for w in tokenizedText]
```

Utrecht University

24

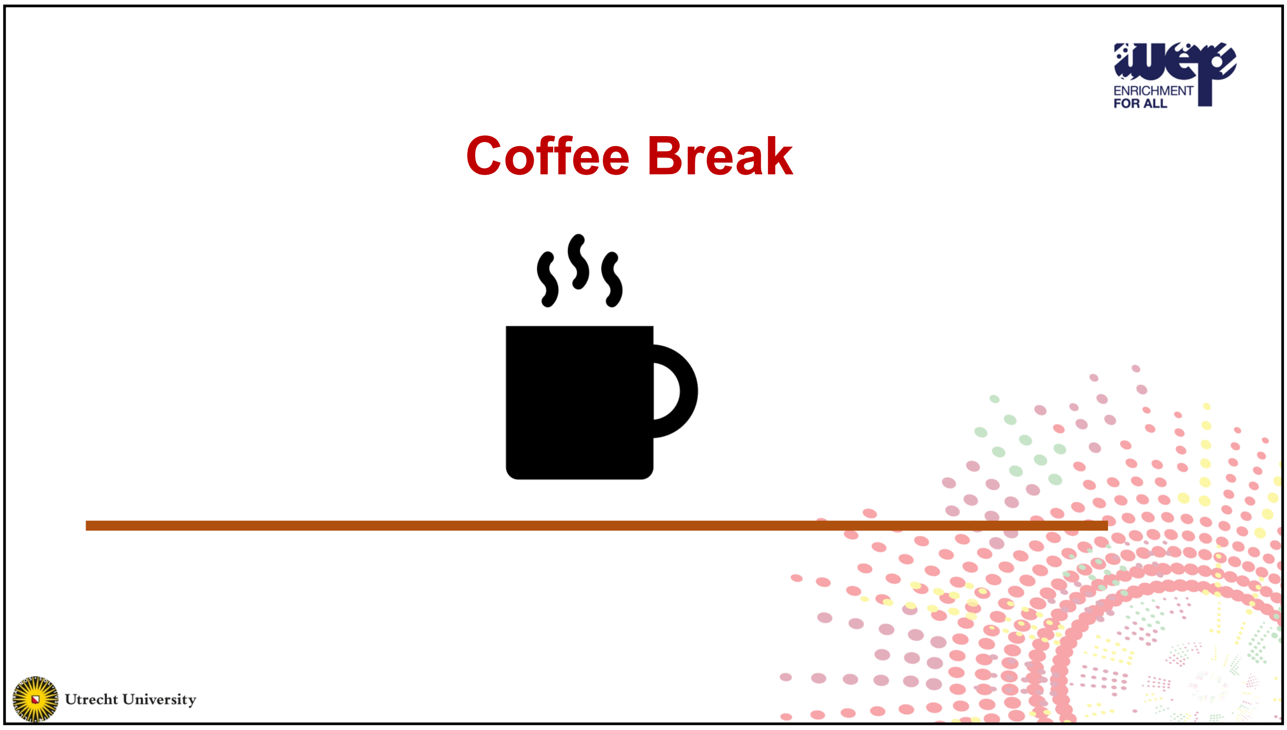
24



Coding Time

ENRICHMENT FOR ALL
Utrecht University

25



Coffee Break

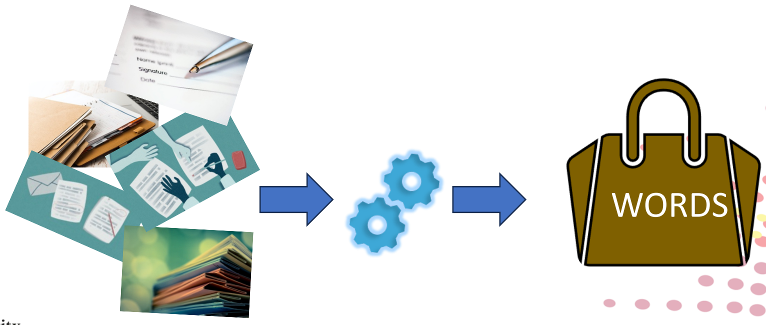
ENRICHMENT FOR ALL
Utrecht University

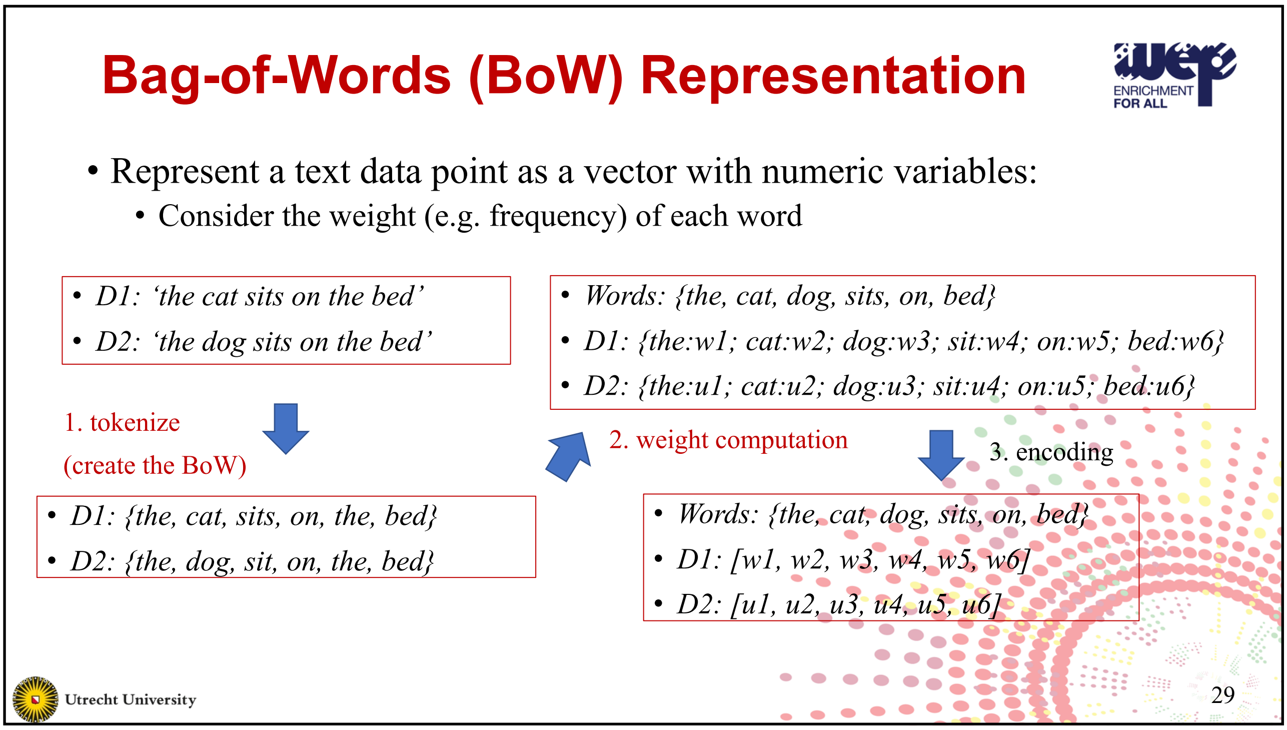
26

Textual Data Representation

Create Representation

- Bag-of-Words
  - Goal: convert the text into a bag of words (repetition is allowed)
    - Ignore grammar and ordering relations between the words
  - Idea: Words are the basic units of the textual data –
    - Understanding the text requires understanding the used words





Bag-of-Words (BoW) Representation

- Represent a text data point as a vector with numeric variables:
  - Consider the weight (e.g. frequency) of each word

- *D1: 'the cat sits on the bed'*
- *D2: 'the dog sits on the bed'*

1. tokenize
(create the BoW)

- *D1: {the, cat, sits, on, the, bed}*
- *D2: {the, dog, sit, on, the, bed}*

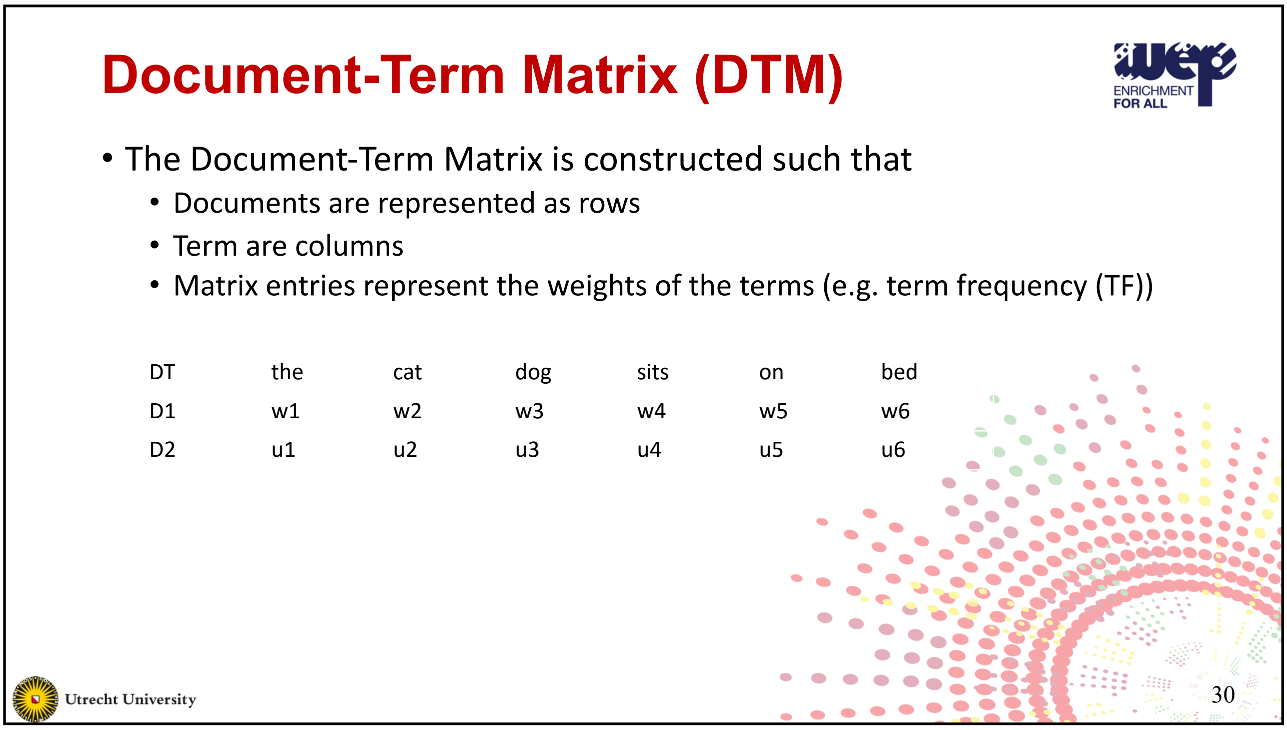
2. weight computation

- *Words: {the, cat, dog, sits, on, bed}*
- *D1: {the:w1; cat:w2; dog:w3; sit:w4; on:w5; bed:w6}*
- *D2: {the:u1; cat:u2; dog:u3; sit:u4; on:u5; bed:u6}*

3. encoding

- *Words: {the, cat, dog, sits, on, bed}*
- *D1: [w1, w2, w3, w4, w5, w6]*
- *D2: [u1, u2, u3, u4, u5, u6]*

Utrecht University

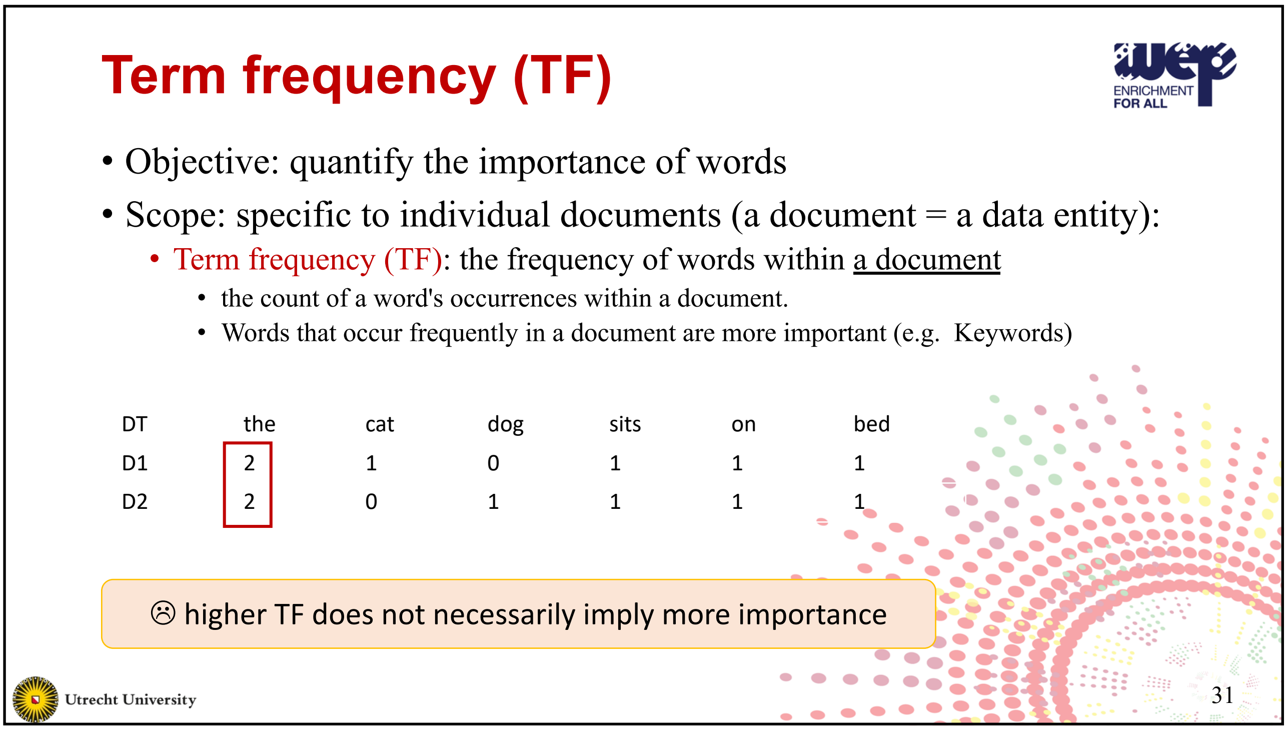


Document-Term Matrix (DTM)

- The Document-Term Matrix is constructed such that
  - Documents are represented as rows
  - Term are columns
  - Matrix entries represent the weights of the terms (e.g. term frequency (TF))

DT	the	cat	dog	sits	on	bed
D1	w1	w2	w3	w4	w5	w6
D2	u1	u2	u3	u4	u5	u6

Utrecht University

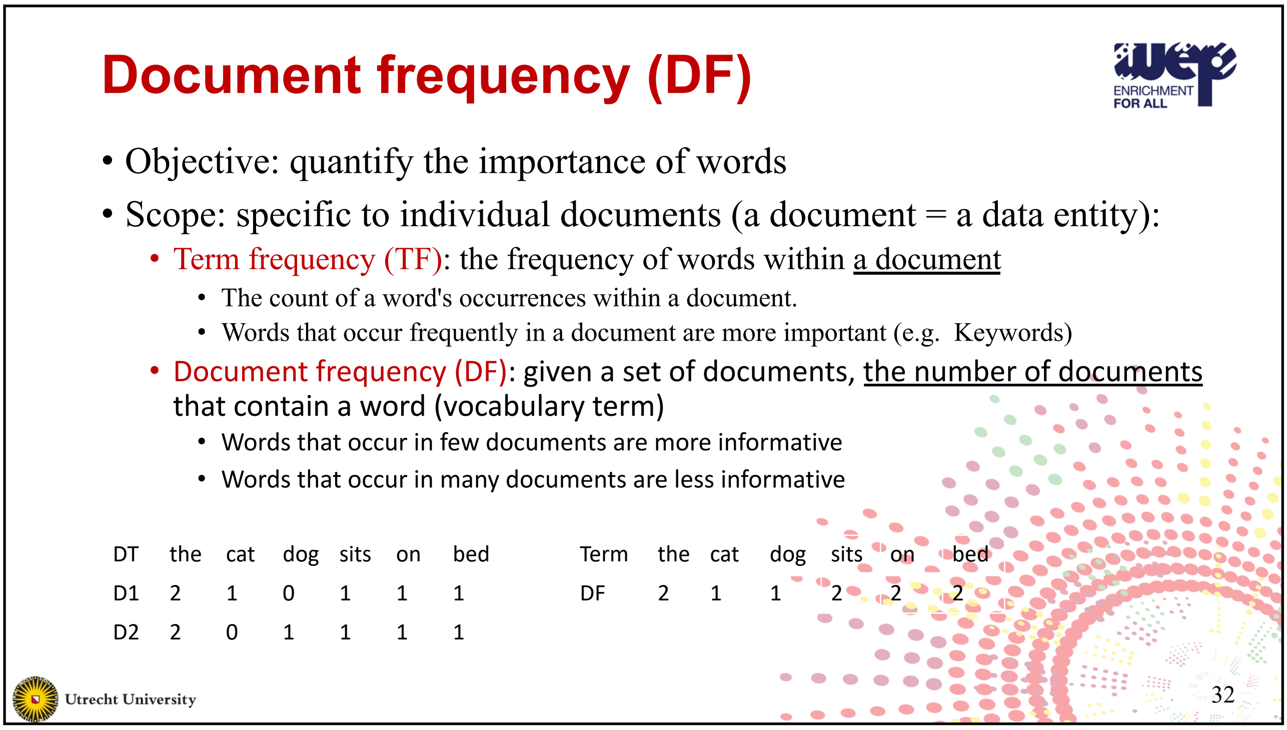


Term frequency (TF)

- Objective: quantify the importance of words
- Scope: specific to individual documents (a document = a data entity):
  - Term frequency (TF): the frequency of words within a document
    - the count of a word's occurrences within a document.
    - Words that occur frequently in a document are more important (e.g. Keywords)

DT	the	cat	dog	sits	on	bed
D1	2	1	0	1	1	1
D2	2	0	1	1	1	1

☹ higher TF does not necessarily imply more importance

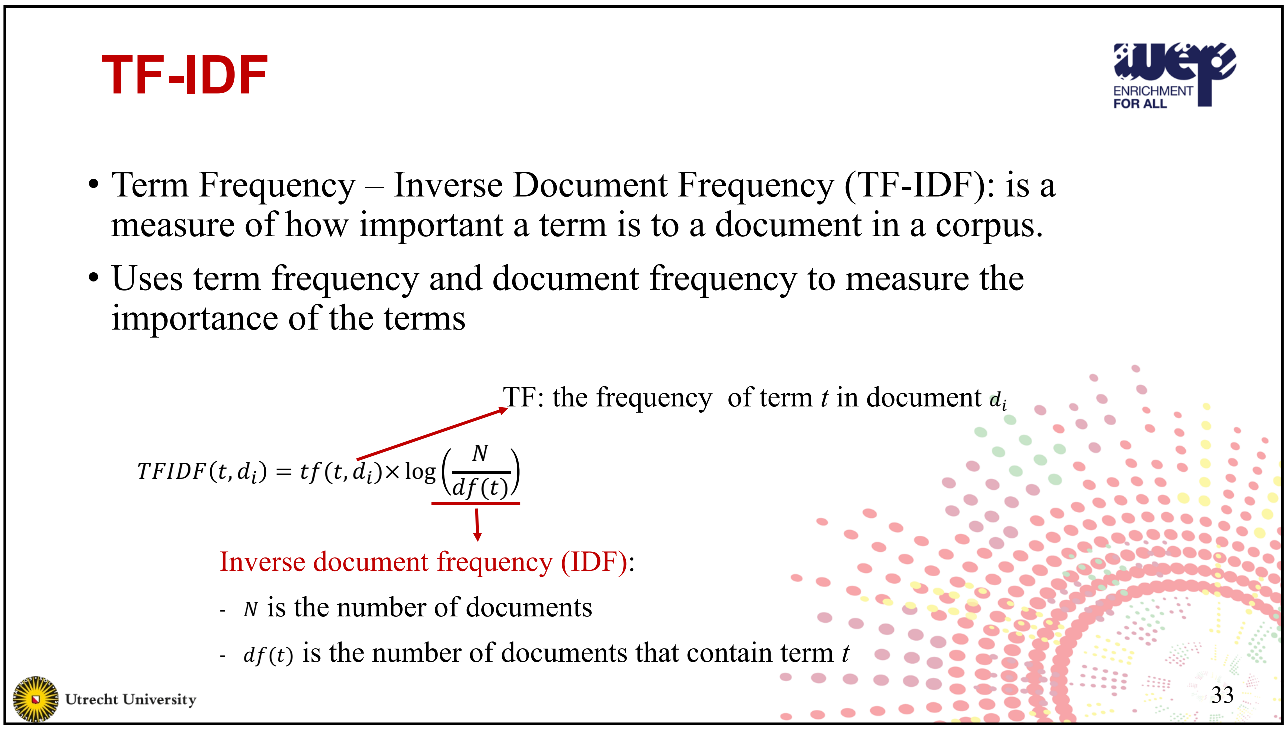


Document frequency (DF)

- Objective: quantify the importance of words
- Scope: specific to individual documents (a document = a data entity):
  - Term frequency (TF): the frequency of words within a document
    - The count of a word's occurrences within a document.
    - Words that occur frequently in a document are more important (e.g. Keywords)
  - Document frequency (DF): given a set of documents, the number of documents that contain a word (vocabulary term)
    - Words that occur in few documents are more informative
    - Words that occur in many documents are less informative

DT	the	cat	dog	sits	on	bed
D1	2	1	0	1	1	1
D2	2	0	1	1	1	1

Term	the	cat	dog	sits	on	bed
DF	2	1	1	2	2	2



TF-IDF

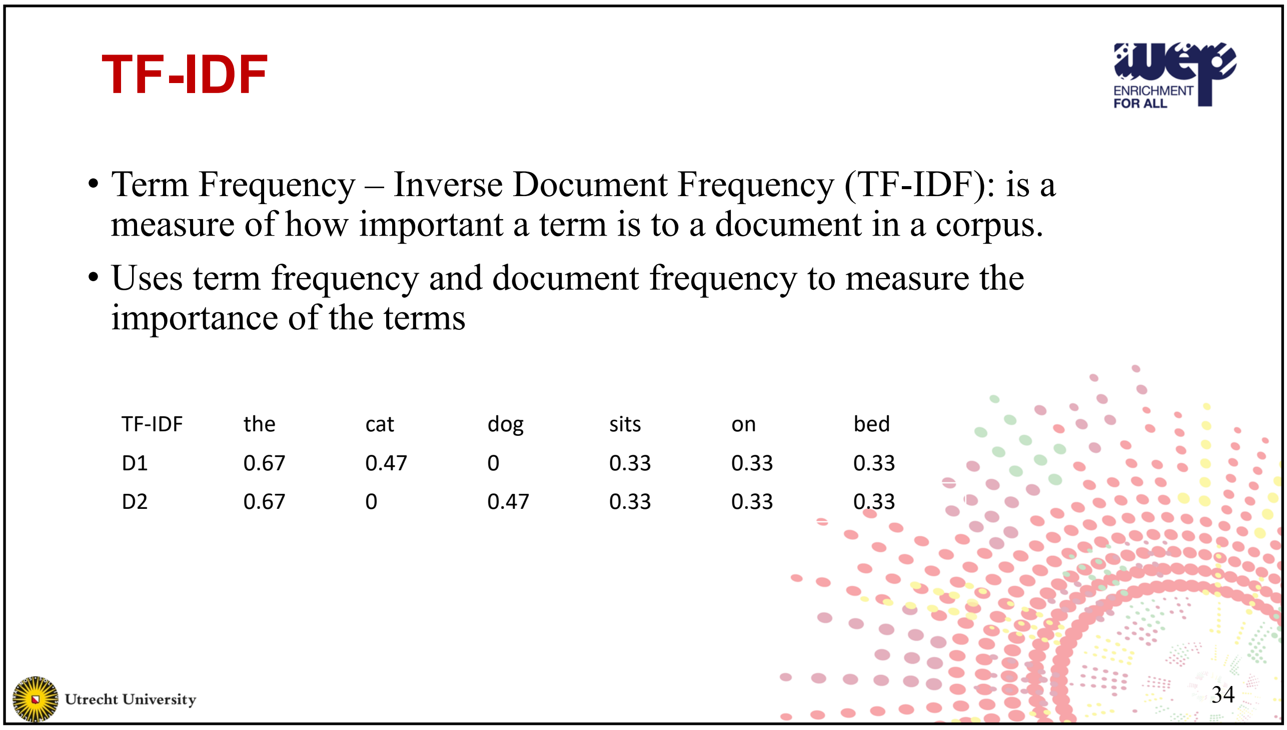
- Term Frequency – Inverse Document Frequency (TF-IDF): is a measure of how important a term is to a document in a corpus.
- Uses term frequency and document frequency to measure the importance of the terms

TF: the frequency of term t in document d_i

$$TFIDF(t,d_i) = tf(t,d_i)\times\log\left(\frac{N}{df(t)}\right)$$

Inverse document frequency (IDF):

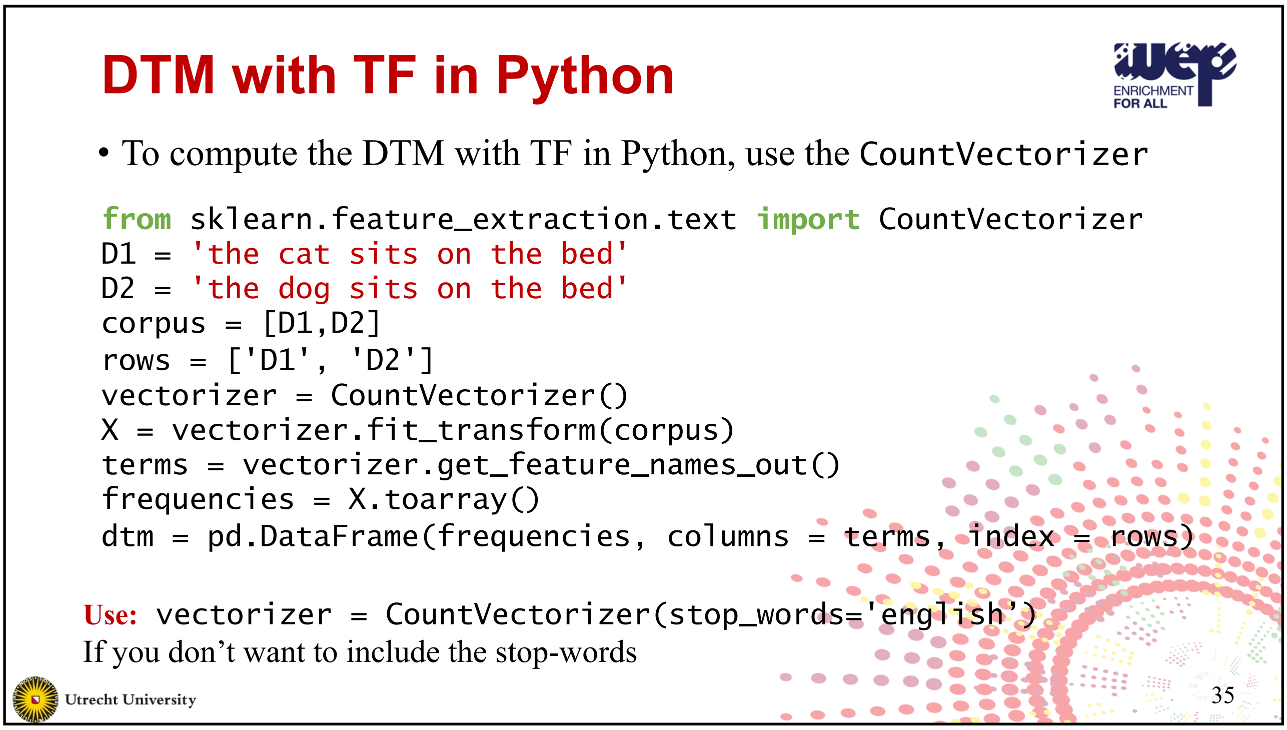
- N is the number of documents
- $df(t)$ is the number of documents that contain term t



TF-IDF

- Term Frequency – Inverse Document Frequency (TF-IDF): is a measure of how important a term is to a document in a corpus.
- Uses term frequency and document frequency to measure the importance of the terms

TF-IDF	the	cat	dog	sits	on	bed
D1	0.67	0.47	0	0.33	0.33	0.33
D2	0.67	0	0.47	0.33	0.33	0.33



DTM with TF in Python

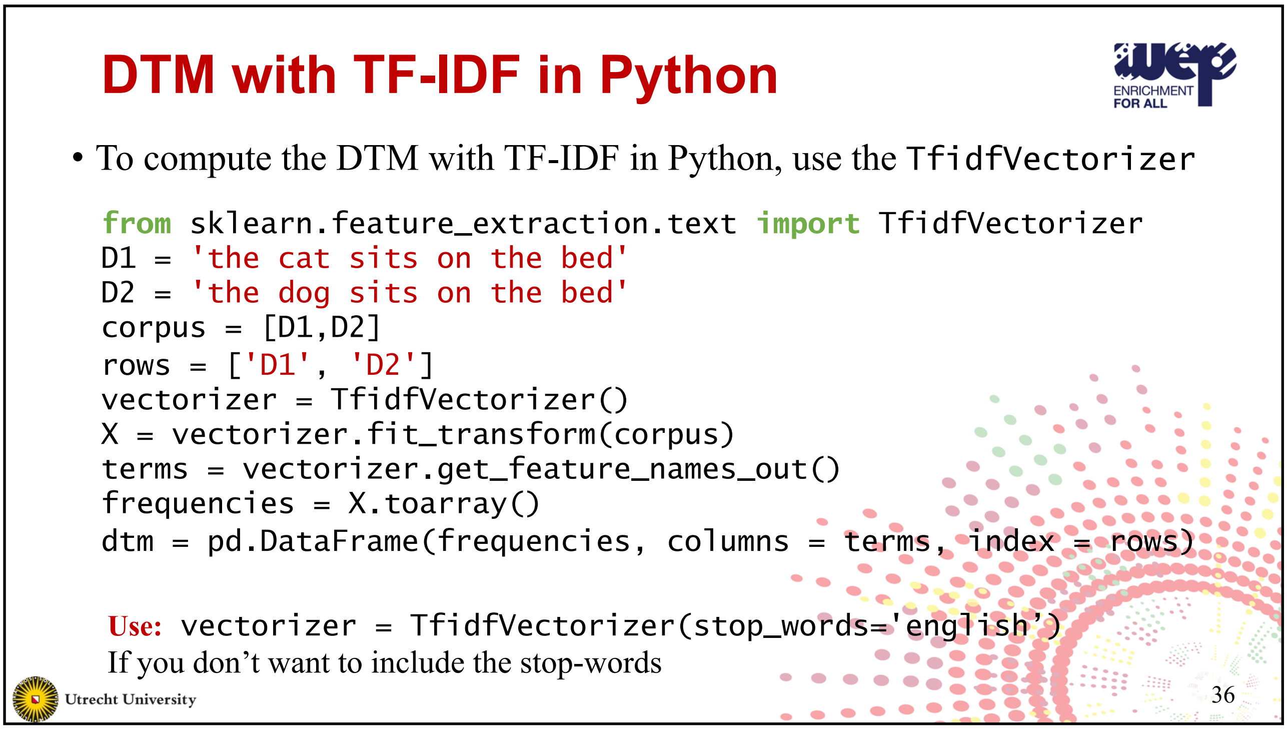
- To compute the DTM with TF in Python, use the `CountVectorizer`

```
from sklearn.feature_extraction.text import CountVectorizer
D1 = 'the cat sits on the bed'
D2 = 'the dog sits on the bed'
corpus = [D1,D2]
rows = ['D1', 'D2']
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)
terms = vectorizer.get_feature_names_out()
frequencies = X.toarray()
dtm = pd.DataFrame(frequencies, columns = terms, index = rows)
```

Use: `vectorizer = CountVectorizer(stop_words='english’)`
If you don’t want to include the stop-words

Utrecht University 35

35



DTM with TF-IDF in Python

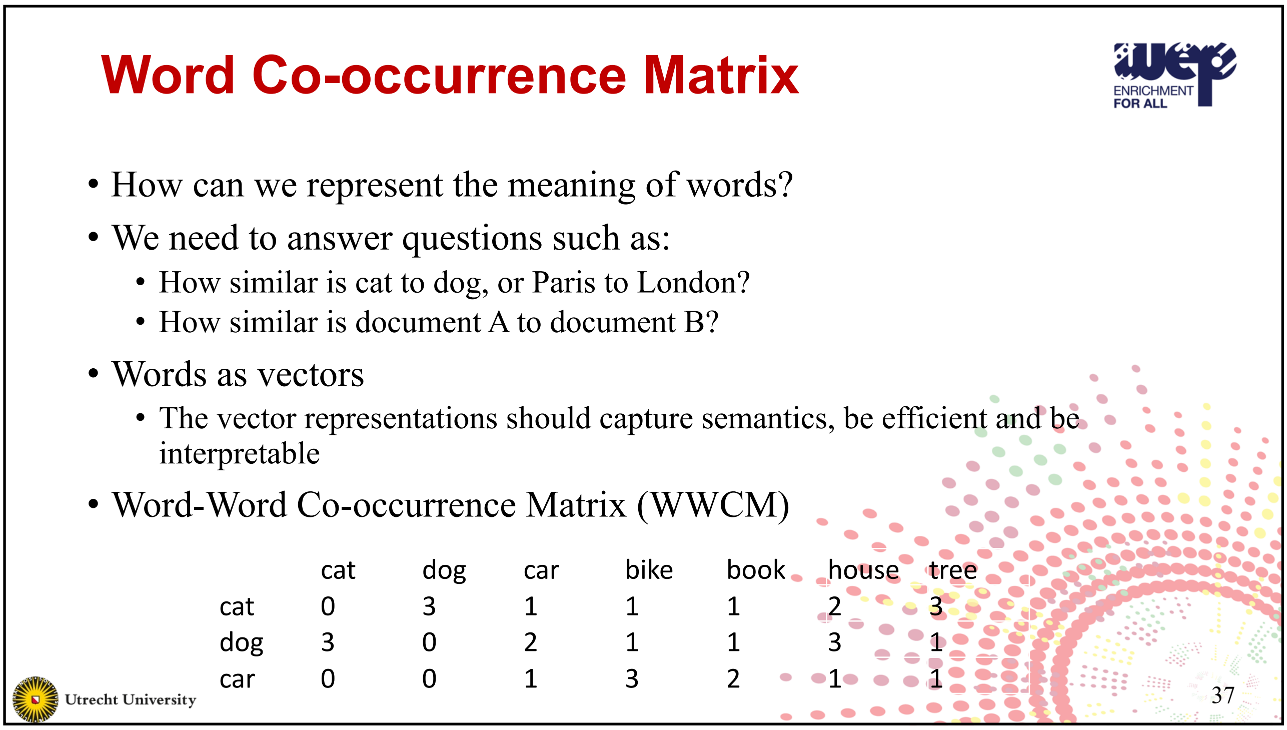
- To compute the DTM with TF-IDF in Python, use the `TfidfVectorizer`

```
from sklearn.feature_extraction.text import TfidfVectorizer
D1 = 'the cat sits on the bed'
D2 = 'the dog sits on the bed'
corpus = [D1,D2]
rows = ['D1', 'D2']
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
terms = vectorizer.get_feature_names_out()
frequencies = X.toarray()
dtm = pd.DataFrame(frequencies, columns = terms, index = rows)
```

Use: `vectorizer = TfidfVectorizer(stop_words='english’)`
If you don’t want to include the stop-words

Utrecht University 36

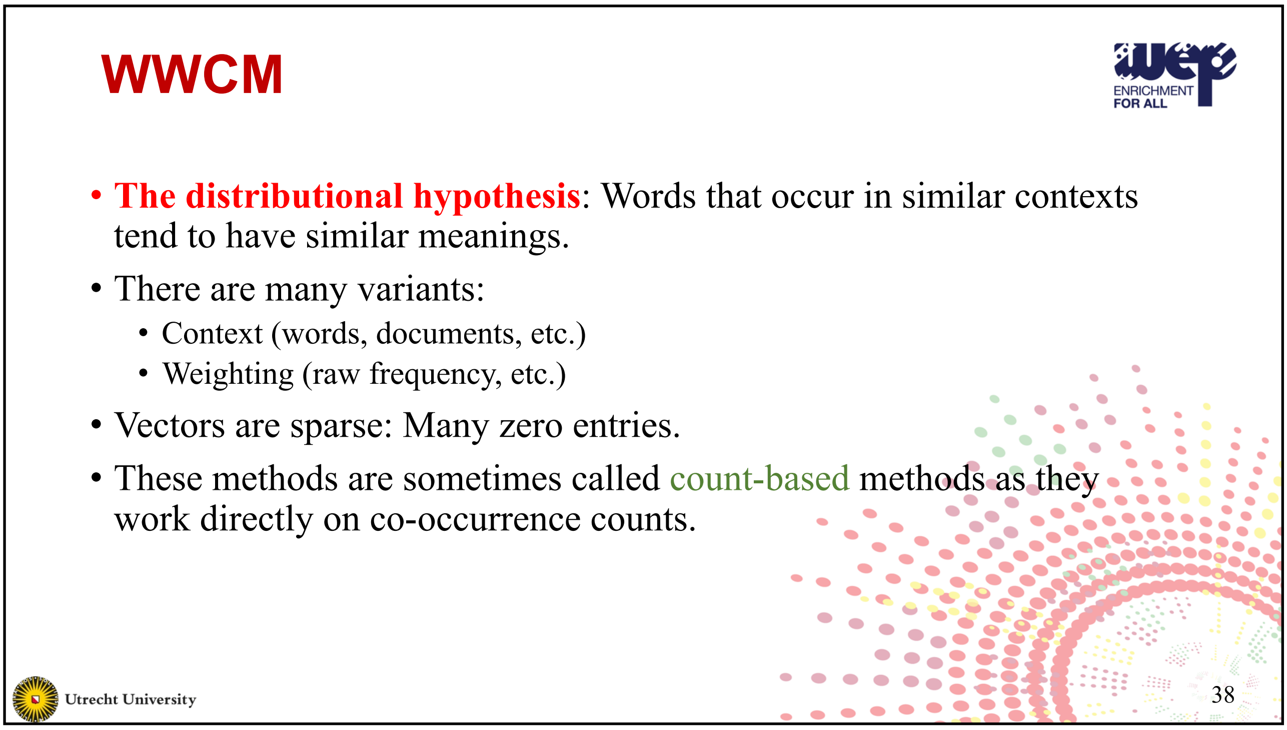
36



Word Co-occurrence Matrix

- How can we represent the meaning of words?
- We need to answer questions such as:
  - How similar is cat to dog, or Paris to London?
  - How similar is document A to document B?
- Words as vectors
  - The vector representations should capture semantics, be efficient and be interpretable
- Word-Word Co-occurrence Matrix (WWCM)

	cat	dog	car	bike	book	house	tree
cat	0	3	1	1	1	2	3
dog	3	0	2	1	1	3	1
car	0	0	1	3	2	1	1



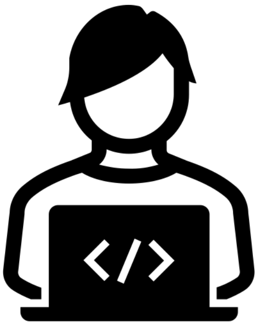
WWCM

- **The distributional hypothesis**: Words that occur in similar contexts tend to have similar meanings.
- There are many variants:
  - Context (words, documents, etc.)
  - Weighting (raw frequency, etc.)
- Vectors are sparse: Many zero entries.
- These methods are sometimes called count-based methods as they work directly on co-occurrence counts.

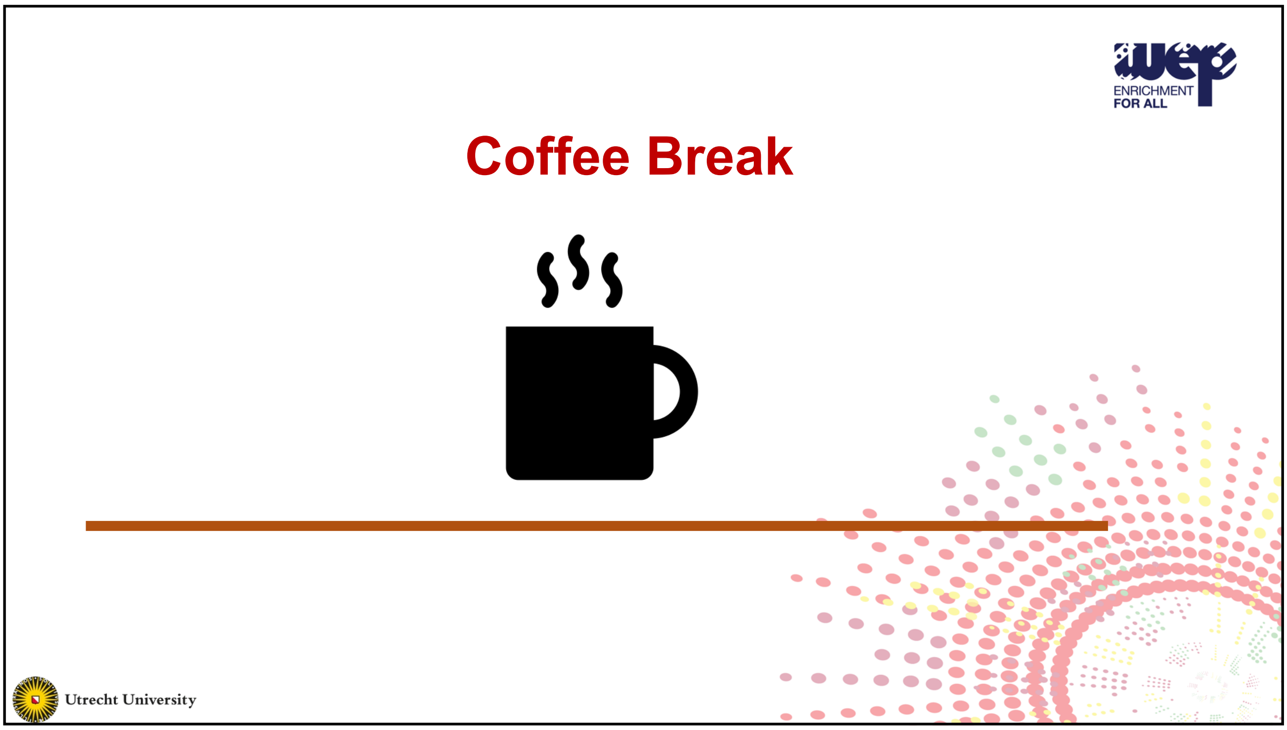
WWCM in Python

- To compute the WWCM in Python, use the CountVectorizer

```
from sklearn.feature_extraction.text import CountVectorizer
D1 = 'the cat sits on the bed'
D2 = 'the dog sits on the bed'
corpus = [D1,D2]
rows = ['D1', 'D2']
cv = CountVectorizer(ngram_range=(1,1), stop_words = 'english')
X = cv.fit_transform(corpus)
Xc = (X.T * X)
Xc.setdiag(0)
names = cv.get_feature_names_out()
df = pd.DataFrame(data = Xc.toarray(), columns = names, index = names)
```

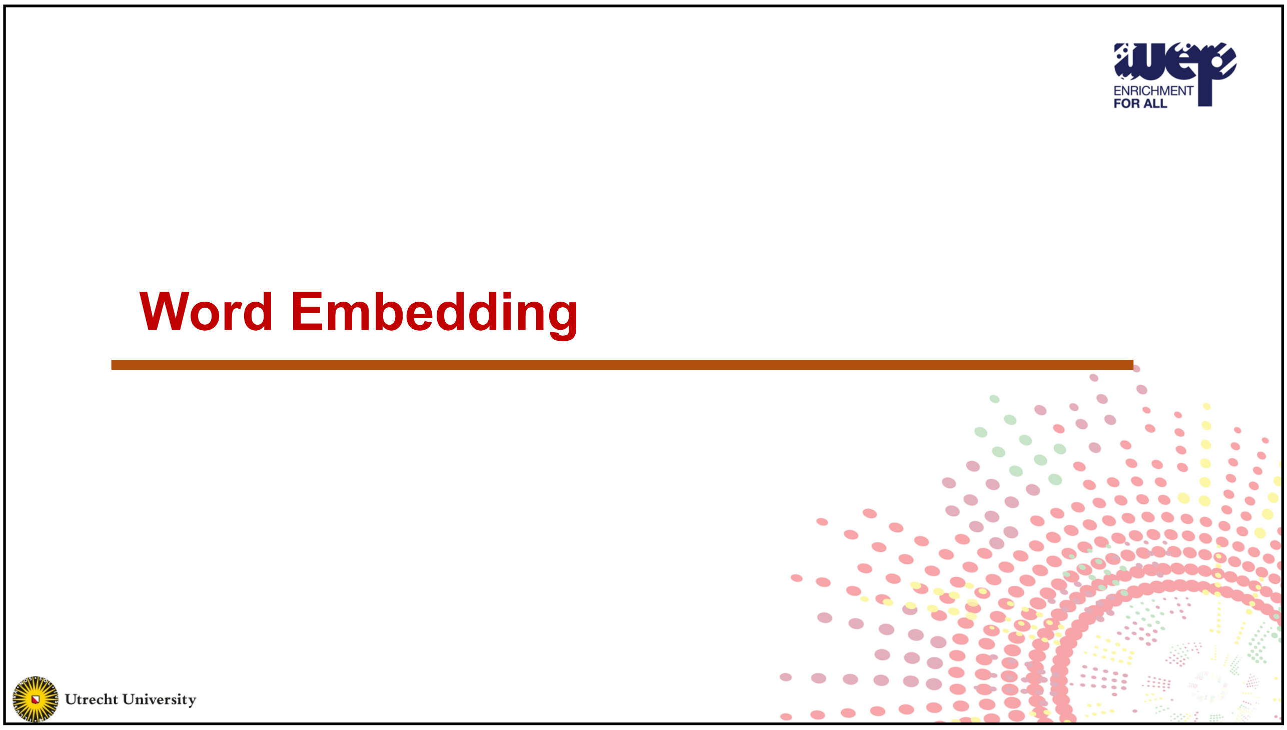


Coding Time



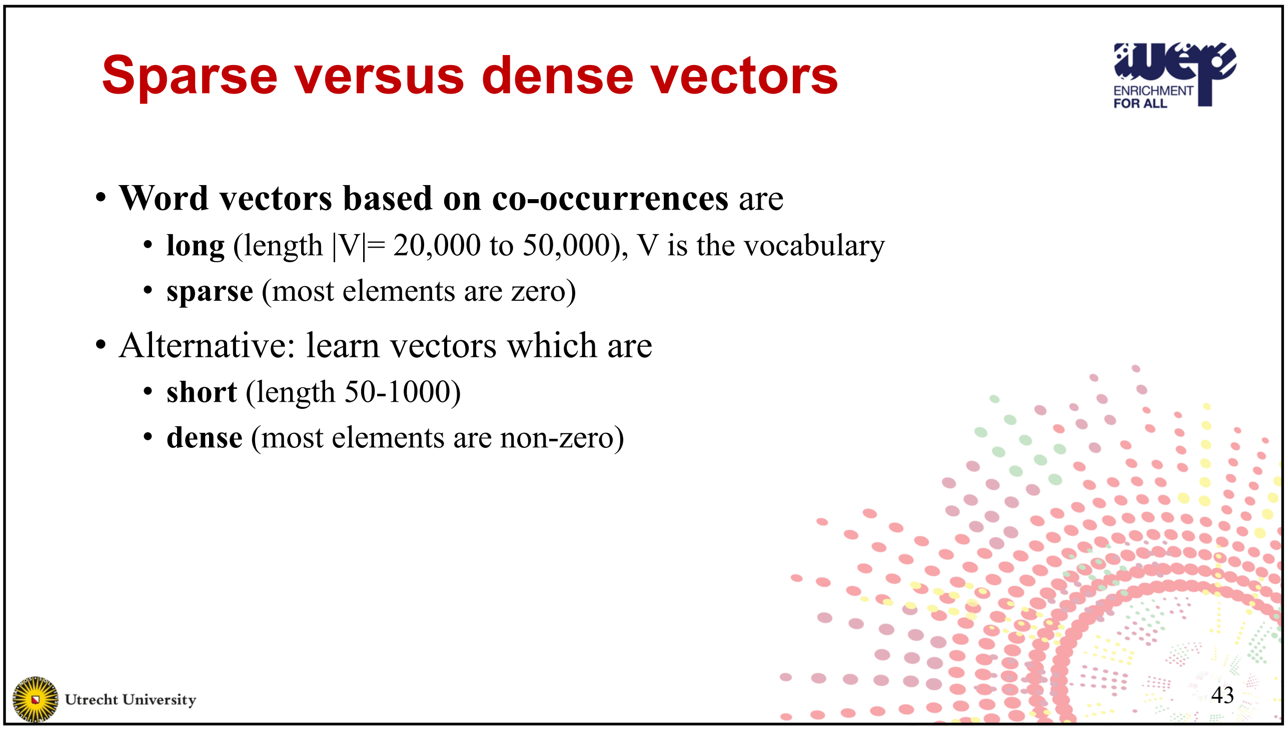
Coffee Break

41



Word Embedding

42



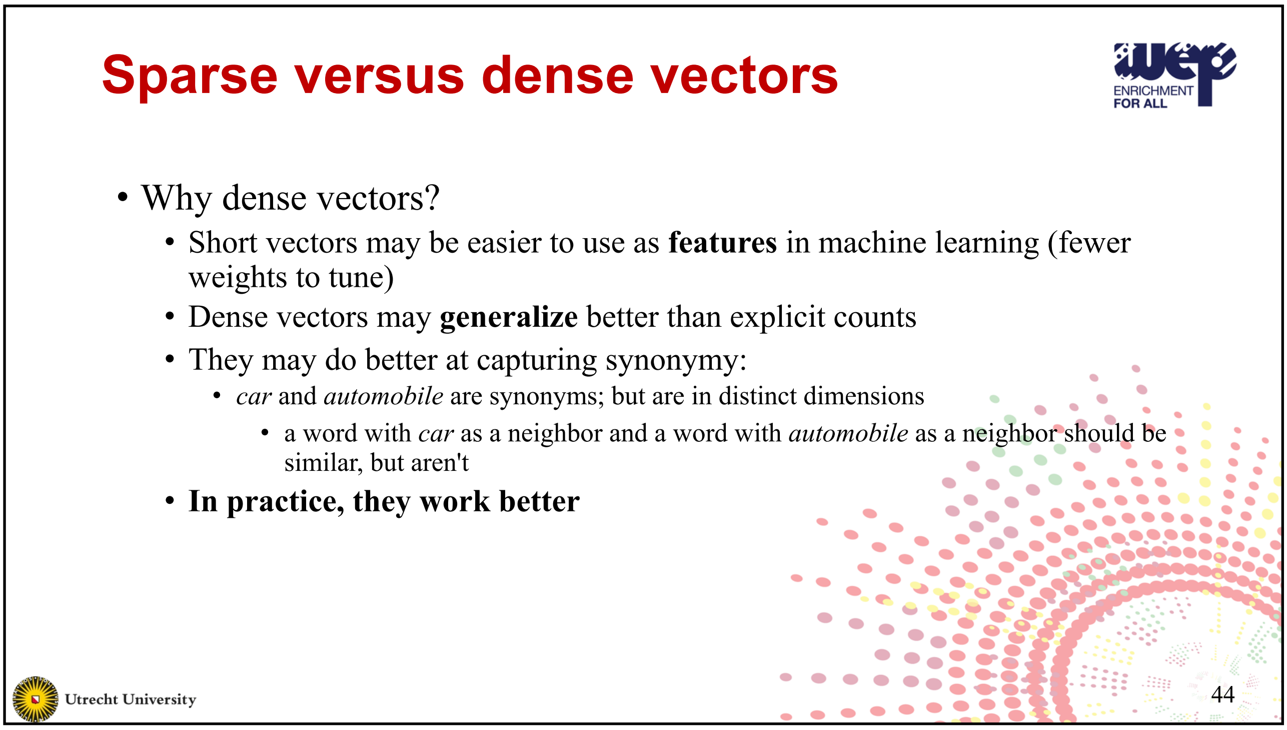
Sparse versus dense vectors

- **Word vectors based on co-occurrences** are
  - **long** (length |V|= 20,000 to 50,000), V is the vocabulary
  - **sparse** (most elements are zero)
- Alternative: learn vectors which are
  - **short** (length 50-1000)
  - **dense** (most elements are non-zero)

Utrecht University

43

43



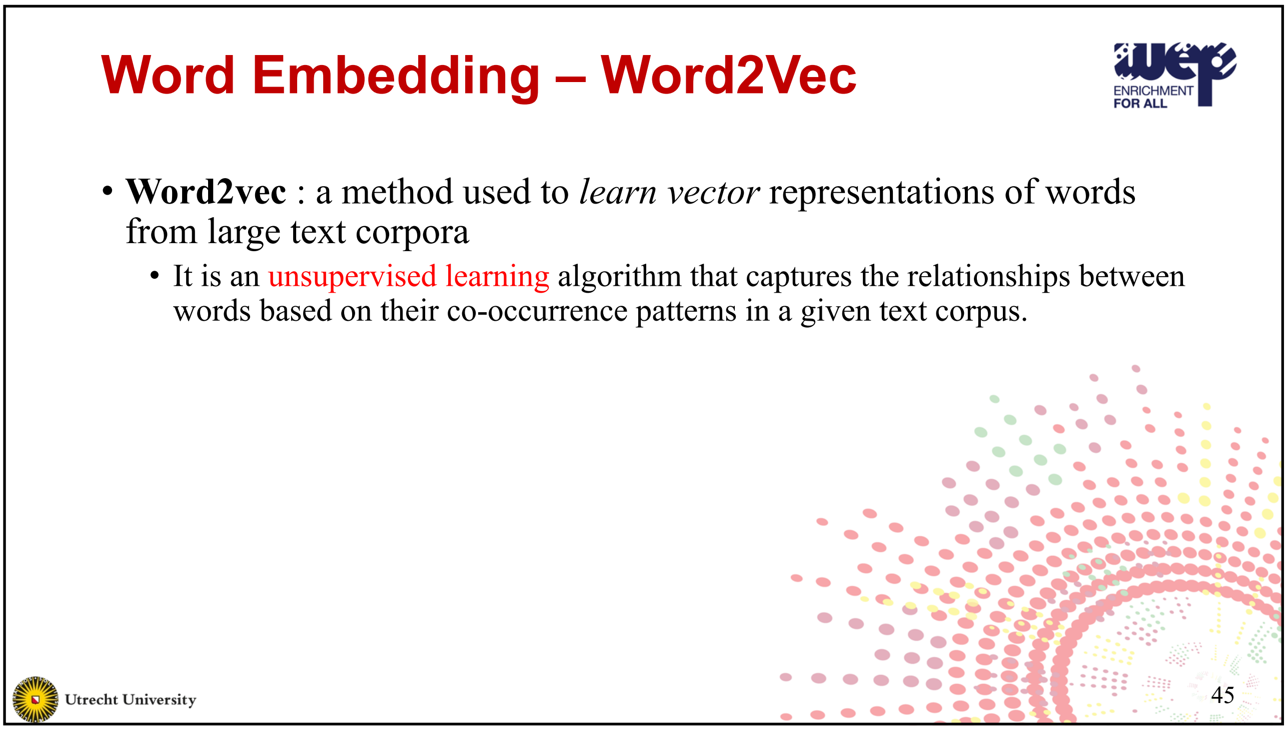
Sparse versus dense vectors

- Why dense vectors?
  - Short vectors may be easier to use as **features** in machine learning (fewer weights to tune)
  - Dense vectors may **generalize** better than explicit counts
  - They may do better at capturing synonymy:
    - *car* and *automobile* are synonyms; but are in distinct dimensions
      - a word with *car* as a neighbor and a word with *automobile* as a neighbor should be similar, but aren't
  - **In practice, they work better**

Utrecht University

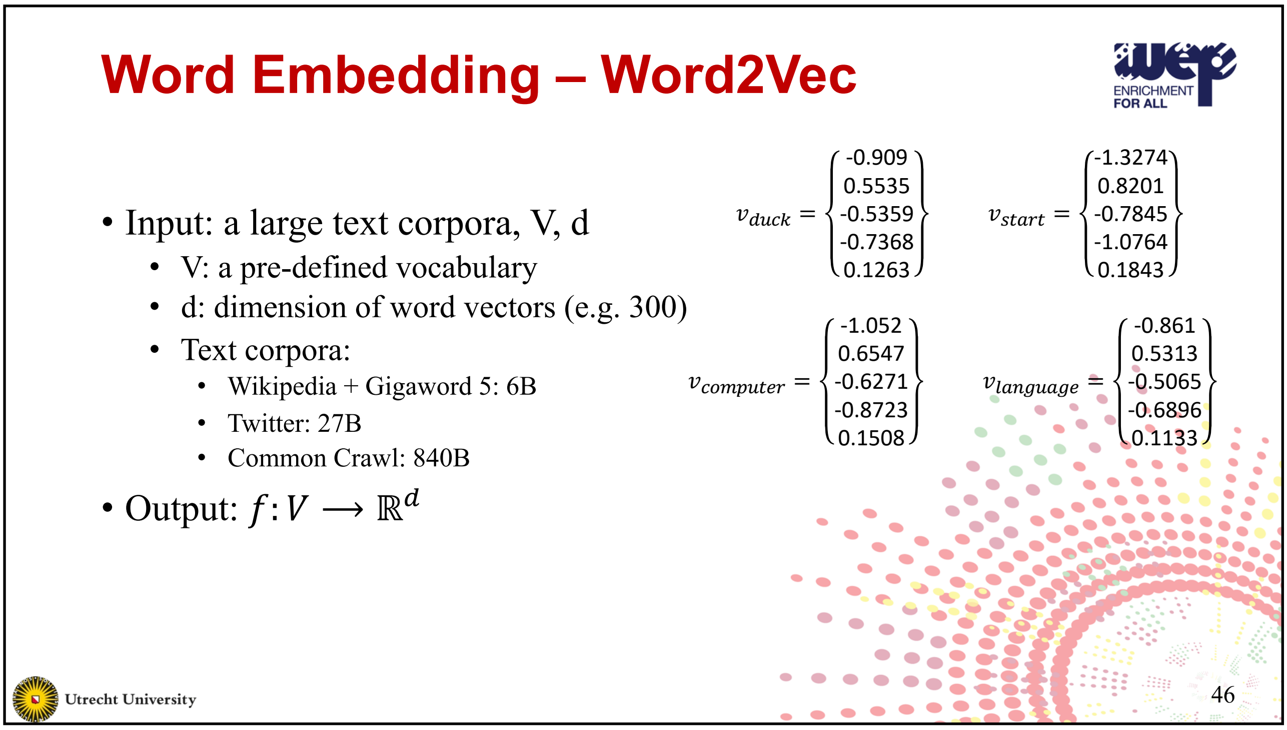
44

44



Word Embedding – Word2Vec

- **Word2vec** : a method used to *learn vector* representations of words from large text corpora
  - It is an unsupervised learning algorithm that captures the relationships between words based on their co-occurrence patterns in a given text corpus.



Word Embedding – Word2Vec

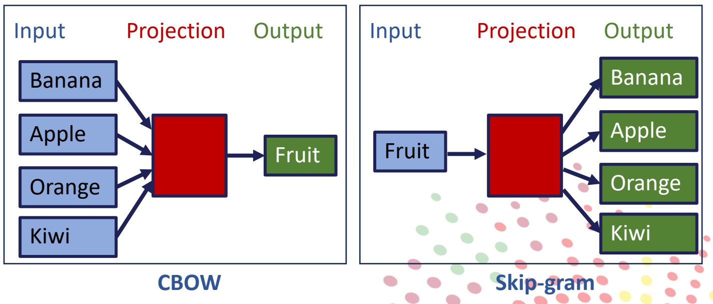
- Input: a large text corpora, V, d
  - V: a pre-defined vocabulary
  - d: dimension of word vectors (e.g. 300)
  - Text corpora:
    - Wikipedia + Gigaword 5: 6B
    - Twitter: 27B
    - Common Crawl: 840B
- Output: $f: V \longrightarrow \mathbb{R}^d$

$$v_{duck} = \begin{Bmatrix} -0.909 \\ 0.5535 \\ -0.5359 \\ -0.7368 \\ 0.1263 \end{Bmatrix} \quad v_{start} = \begin{Bmatrix} -1.3274 \\ 0.8201 \\ -0.7845 \\ -1.0764 \\ 0.1843 \end{Bmatrix}$$

$$v_{computer} = \begin{Bmatrix} -1.052 \\ 0.6547 \\ -0.6271 \\ -0.8723 \\ 0.1508 \end{Bmatrix} \quad v_{language} = \begin{Bmatrix} -0.861 \\ 0.5313 \\ -0.5065 \\ -0.6896 \\ 0.1133 \end{Bmatrix}$$

Word2Vec – Two Models

- Word2vec can be used in various ways:
  - The CBOW (continuous bag-of-words) predicts the current word based on the context or nearby words
  - The Skip-gram predicts surrounding words given the current word



Application – Sentiment Analysis

Sentiment Analysis

- Try to extract and identify positive/negative valence from a text.
- Basic idea:

  Sentiment = Total no. positive words − Total no. negative words

  - Use ‘sentiment dictionaries’ (lexicons) to assess a score (positive/negative) or emotion to each term;
- In Python: transformers, nltk, TextBlob;
- More advanced methods: use classification to predict sentiment from text (e.g. tf-idf).

Sentiment Analysis with Transformers

- Measure the sentiment of the sentences

  "KAUST was established to become world class university"

  And

  "KAU is an old university"

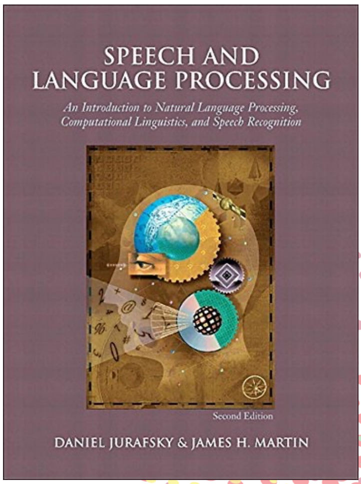
```
from sklearn.feature_extraction.text import CountVectorizer
D1 = 'the cat sits on the bed’
from transformers import pipeline
sentiment_pipeline = pipeline("sentiment-analysis")
s1 = "KAUST was established to become world class university”
s2 = "KAU is an old university"
data = [s1, s2]
sentiment_pipeline(data)
```

Reading Material for Interested Students

- Speech and Language Processing

  Ch 2, 6

This is a link to the draft of the 3-rd edition



SPEECH AND LANGUAGE PROCESSING

An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition

Second Edition

DANIEL JURAFSKY & JAMES H. MARTIN

Thanks to:

Mel Chekol

Daniel Oberski

Dong Nguyen

Utrecht University

51

51

Thank You

Utrecht University

52

Utrecht University

DISCLAIMER

The information in this presentation has been compiled with the utmost care, but no rights can be derived from its contents.

53