HSS 611 - Week 14: Text-as-data

2023-11-27

Agenda

- Pattern matching with regex
  - Introduction to key approaches
- Text normalization
  - Tokenization
  - Stopwords
  - Lemmatization/stemming
- Representing and comparing texts
  - Count vectors
  - TF-IDF
  - Cosine similarity
- Word embeddings
  - Vector semantics
  - Pretrained embeddings

Pattern matching with regex

Matching substrings

- Finding and counting substrings

```
print("strawberry".find("berry"))
print("strawberry".count("berry"))
```

```
5
1
```

Pattern matching with regex

Matching substrings

- Finding and counting substrings
- `find()` returns -1 for non-match

```
print("strawberry".find("better"))
print("strawberry".count("better"))
```

```
-1
0
```

Pattern matching with regex

Matching substrings

- Returns the first match

```
print("berryberrystrawberry".find("berry"))
```

0

Pattern matching with regex

Matching substrings

- List of 80 fruits

```
import pandas as pd
url = 'https://raw.githubusercontent.com/taegyoon-kim/programming_dhcss_23fw/ma
fruit = pd.read_csv(url, header = None)[0].to_list()
print(len(fruit))
print(fruit[0])
```

```
80
apple
```

Pattern matching with regex

Matching substrings

- Use list comprehension to extract matches

```
[i for i in fruit if i.find("berry") > -1]
```

```
['bilberry',
 'blackberry',
 'blueberry',
 'boysenberry',
 'cloudberry',
 'cranberry',
 'elderberry',
 'goji berry',
 'gooseberry',
 'huckleberry',
 'mulberry',
 'raspberry',
 'salal berry',
 'strawberry']
```

Pattern matching with regex

What is regex?

- A sequence of characters that forms a flexible search pattern
- Can be used to check if a string contains the specified search pattern

```
import re
mo = re.search(r'berry', 'strawberry.')
print(type(mo))
```

```
<class 're.Match'>
```

Pattern matching with regex

Extraction into a list

```
print(mo)
print(mo.group())

berries = [i for i in fruit if re.search(r'berry',i)]
print(berries)

c = re.compile(r'berry') # *compile* the pattern into an object
berries = [i for i in fruit if c.search(i)]
```

```
<re.Match object; span=(5, 10), match='berry'>
berry
['bilberry', 'blackberry', 'blueberry', 'boysenberry', 'cloudberry', 'cranberry
```

Pattern matching with regex

Multiple matches

```
mo_m = re.search(r'berry',"berryberrystrawberry")
print(mo_m.group()) # returns the first match

mo_m2 = re.findall(r'berry',"berryberrystrawberry")
print(mo_m2) # this is a list of strings
```

```
berry
['berry', 'berry', 'berry']
```

Pattern matching with regex

Square brackets for “or”

- Square brackets for “or”: matches “any one of” the characters in `[]`.
- Read data first

```
url = 'https://raw.githubusercontent.com/taegyoon-kim/programming_dhcss_23fw/ma
sentences = pd.read_csv(url, header = None, sep = '@')[0].to_list()
print(len(sentences))
print(sentences[0])
```

```
720
The birch canoe slid on the smooth planks.
```

Pattern matching with regex

Square brackets for “or”

- For `beat`, `heat`, `peat`

```
c = re.compile(r' [bhp]eat ')
l_mo = [i for i in sentences if c.search(i)]
for i in l_mo:
  print(i)
```

```
The heart beat strongly and with firm strokes.
Burn peat after the logs give out.
Feel the heat of the weak dying flame.
A speedy man can beat this track mark.
Even the worst will beat his low score.
It takes heat to bring out the odor.
```

Pattern matching with regex

Square brackets for "or"

- Use - to indicate a range of contiguous characters

```
c = re.compile(r' [b-p]eat ')
l_mo = [i for i in sentences if c.search(i)]
for i in l_mo:
  print(i)
```

```
The heart beat strongly and with firm strokes.
Burn peat after the logs give out.
Feel the heat of the weak dying flame.
A speedy man can beat this track mark.
Even the worst will beat his low score.
Pack the records in a neat thin case.
It takes heat to bring out the odor.
A clean neck means a neat collar.
```

Pattern matching with regex

Square brackets for “or”

- Match anything but one of the characters in the square brackets

```
c = re.compile(r' [^bhp]eat ')
l_mo = [i for i in sentences if c.search(i)]
for i in l_mo:
  print(i)
```

```
Pack the records in a neat thin case.
A clean neck means a neat collar.
```

Pattern matching with regex

"Or" over multi-character patterns

- We can use | operator
- Parentheses can be used to indicate parts in the pattern

```
c = re.compile(r'(black|blue|red)(currant|berry)')
l_mo = [i for i in fruit if c.search(i)]
l_mo
```

```
['blackberry', 'blackcurrant', 'blueberry', 'redcurrant']
```

Pattern matching with regex

The group() attribute

```
c = re.compile(r'(black|blue|red)(currant|berry)')
mo = c.search('blackberry')
print(mo.group(0))
print(mo.group(1))
print(mo.group(2))
```

```
blackberry
black
berry
```

Pattern matching with regex

Special characters and the backslash

- Let's create a random example string

```
eg_str = 'Example STRING, with numbers (12, 15 and also 10.2)?! Wow, two sentences.'
```

Pattern matching with regex

Special characters and the backslash

- There are several characters that have a special meaning in regex, and (may) have to be escaped in order to match the literal character
- They include `^`, `$`, `.`, `*`, `+`, |, `!`, `?`, `(`, `)`, `[`, `]`, {, }, <, and >

Pattern matching with regex

Special characters and the backslash

- For example, `.` means "any character but a newline"

```
allchars = re.findall(r'.', eg_str)
print(allchars)
```

```
['E', 'x', 'a', 'm', 'p', 'l', 'e', ' ', 'S', 'T', 'R', 'I
```

```
allperiods = re.findall(r'\.', eg_str)
print(allperiods)
```

```
['.', '.']
```

Pattern matching with regex

Special characters and the backslash

- For example, `.` means "any character but a newline"

```
matches = re.findall(r'a.', eg_str)
print(matches)
matches = re.findall(r'a\.', eg_str)
print(matches)
```

```
['am', 'an', 'al']
[]
```

Pattern matching with regex

Class shorthands

- "\w" (any alphanumeric character), "\s" (any space character), and "\d" (any numeric digit)
- The capitalized versions of these are used to mean "anything but" that class

```
# any alphanumeric character
matches = re.findall(r'\w',eg_str)
print(matches)

# any non-alphanumeric character
matches = re.findall(r'\W',eg_str)
print(matches)
```

```
['E', 'x', 'a', 'm', 'p', 'l', 'e', 'S', 'T', 'R', 'I', 'N', 'G', 'w', 'i', 't', 'h', 'n', 'u', 'm', 'b',
[' ', ',', ' ', ' ', ' ', '(', ',', ' ', ' ', ' ', ' ', '.', ')', '?', '!', ' ', ',', ' ', ' ', '.']
```

Pattern matching with regex

Class shorthands

```
# any whitespace character
matches = re.findall(r'\s', eg_str)
print(matches)

# any non-whitespace character
matches = re.findall(r'\S', eg_str)
print(matches)
```

```
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['E', 'x', 'a', 'm', 'p', 'l', 'e', 'S', 'T', 'R', 'I', 'N', 'G', ',', 'w', 'i', 't', 'h', 'n', 'u', 'm',
```

Pattern matching with regex

Class shorthands

```
# any digit character
matches = re.findall(r'\d', eg_str)
print(matches)

# any non-digit character
matches = re.findall(r'\D', eg_str)
print(matches)
```

```
['1', '2', '1', '5', '1', '0', '2']
['E', 'x', 'a', 'm', 'p', 'l', 'e', ' ', 'S', 'T', 'R', 'I', 'N', 'G', ',', ' ', 'w', 'i', 't', 'h', ' ',
```

Pattern matching with regex

Quantifiers: * (zero or more of the previous)

```
# any string of zero or more digits
matches = re.findall('\d*', eg_str)
print(matches)
```

```
['', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',
```

Pattern matching with regex

Quantifiers: + (one or more of the previous)

```
matches = re.findall('\d+', eg_str) # any string of zero or more digits
print(matches)
```

```
['12', '15', '10', '2']
```

Pattern matching with regex

Quantifiers: {n} {n,m} and {n,}

- {n} = “exactly n” of the previous
- {n,m} = “between n and m” of the previous
- {n,} = “n or more” of the previous

Pattern matching with regex

Quantifiers: {n} {n,m} and {n,}

```
# 3 x's
matches = re.findall(r'x{3}','x xx xxx xxxx xxxxx')
print(matches)

# 3 or 4 x's
matches = re.findall(r'x{3,4}','x xx xxx xxxx xxxxx')
print(matches)

# 3 or more x's
matches = re.findall(r'x{3,}','x xx xxx xxxx xxxxx')
print(matches)
```

```
['xxx', 'xxx', 'xxx']
['xxx', 'xxxx', 'xxxx']
['xxx', 'xxxx', 'xxxxx']
```

Pattern matching with regex

Quantifiers: ? (zero or one of the previous)

```
c = re.compile(r' [bp]?eat ')
matches = [i for i in sentences if c.search(i)]
matches
```

```
['The heart beat strongly and with firm strokes.',
 'Burn peat after the logs give out.',
 'A speedy man can beat this track mark.',
 'Even the worst will beat his low score.',
 'Quench your thirst, then eat the crackers.']
```

Pattern matching with regex

Quantifiers

- `.+` is a greedy quantifier that matches one or more of any character as many times as possible
- `.+?` is a non-greedy quantifier that matches as few characters as possible

```
matches = re.findall(r'\(.+\)',
'(First bracketed statement) Other text (Second bracketed statement)')
print(matches)
```

```
['(First bracketed statement) Other text (Second bracketed statement)']
```

Pattern matching with regex

Quantifiers

- `.+` is a greedy quantifier that matches one or more of any character as many times as possible
- `.+?` is a non-greedy quantifier that matches as few characters as possible

```
matches = re.findall(r'\(.+?\)',
'(First bracketed statement) Other text (Second bracketed statement)')
print(matches)
```

```
['(First bracketed statement)', '(Second bracketed statement)']
```

Pattern matching with regex

Quantifiers

- `.+` is a greedy quantifier that matches one or more of any character as many times as possible
- `.+?` is a non-greedy quantifier that matches as few characters as possible

```
matches = re.findall(r'x.+x','x xx xxx xxxx xxxxx')
print(matches)

matches = re.findall(r'x.+?x','x xx xxx xxxx xxxxx')
print(matches)
```

```
['x xx xxx xxxx xxxxx']
['x x', 'x x', 'xx x', 'xxx', 'xxx']
```

Text normalization

Caveat

- The objective of text normalization is to clean up text and transform it into meaningful units of analysis
- A specific combination of techniques to be employed is highly dependent on your research question
- You always need to ask yourself what impact your decisions would have on subsequent analyses

Text normalization

Caveat

PA

Text Preprocessing For Unsupervised Learning: Why It Matters, When It Misleads, And What To Do About It

Matthew J. Denny[1] and Arthur Spirling[2]

[1] *203 Pond Lab, Pennsylvania State University, University Park, PA 16802, USA. Email: mdenny@psu.edu*
[2] *Office 405, 19 West 4th St., New York University, New York, NY 10012, USA. Email: arthur.spirling@nyu.edu*

Abstract

Despite the popularity of unsupervised techniques for political science text-as-data research, the importance and implications of preprocessing decisions in this domain have received scant systematic attention. Yet, as we show, such decisions have profound effects on the results of real models for real data. We argue that substantive theory is typically too vague to be of use for feature selection, and that the supervised literature is not necessarily a helpful source of advice. To aid researchers working in unsupervised settings, we introduce a statistical procedure and software that examines the sensitivity of findings under alternate preprocessing regimes. This approach complements a researcher's substantive understanding of a problem by providing a characterization of the variability changes in preprocessing choices may induce when analyzing a particular dataset. In making scholars aware of the degree to which their results are likely to be sensitive to their preprocessing decisions, it aids replication efforts.

Keywords: statistical analysis of texts, unsupervised learning, descriptive statistics

Text normalization

Tokenization

- A token is an instance of a sequence of characters in a document that are grouped together as a useful semantic unit
- Tokenization is the task of segmenting running text into tokens
- The very start of downstream NLP tasks
- NLTK (Natural Language ToolKit), among many others, offers useful tools for tokenization

Import NLTK and Punkt Tokenizer

- There are a wide range of tokenizers
- They differ in various aspects
  - E.g., handling punctuation, special characters, or numbers
- See the official document of Punkt Tokenizer

```
import nltk
#nltk.download('punkt') # import Punkt Tokenizer
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
```

Text normalization

```
text = """Altman announced a few weeks ago at OpenAI’s first-ever developer day that the company would mak
print(text)
```

```
Altman announced a few weeks ago at OpenAI’s first-ever developer day that the company would make tools av
OpenAI has also worked with Microsoft to roll out ChatGPT-like technology across Microsoft’s products.
OpenAI and iPhone designer Jony Ive had also reportedly been in talks to raise $1 billion from Japanese co
```

Text normalization

```
sentences = sent_tokenize(text) # a.k.a sentence segmentation
for s in sentences:
  print(f"{s}\n")
```

```
Altman announced a few weeks ago at OpenAI’s first-ever developer day that the company would make tools av

OpenAI has also worked with Microsoft to roll out ChatGPT-like technology across Microsoft’s products.

OpenAI and iPhone designer Jony Ive had also reportedly been in talks to raise $1 billion from Japanese co
```

Text normalization

```
tokenized = word_tokenize(sentences[1])
for i in tokenized:
  print(i)
```

```
OpenAI
has
also
worked
with
Microsoft
to
roll
out
ChatGPT-like
technology
across
Microsoft
’
s
products
.
```

Text normalization

Stop words

- Stop words are the words that are filtered out (due to insignificance)
- For the previous example, the words like ‘the’ or ‘an’ are not quite meaningful
- Again, however, it depends upon the nature of the task you are working on
- E.g., topic modeling vs. language modeling

Stop words

```
from nltk.corpus import stopwords
#nltk.download('stopwords')
print(len(stopwords.words('english')))
print(stopwords.words('english')[0:10])
```

```
179
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're"]
```

Text normalization

Stop words

```
tokens = word_tokenize(sentences[1])

tokens_no_sw = [t for t in tokens if not t.lower() in stopwords.words('english')]

print(tokens)
print(tokens_no_sw)
```

```
['OpenAI', 'has', 'also', 'worked', 'with', 'Microsoft', 'to', 'roll', 'out', 'ChatGPT-like', 'technology
['OpenAI', 'also', 'worked', 'Microsoft', 'roll', 'ChatGPT-like', 'technology', 'across', 'Microsoft', ''
```

Text normalization

Lemmatization

- Lemmatization is identifying the base form of a word, irrespective of its surface variations
  - ‘am’, ‘are’, and ‘is’ share the same lemma ‘be’
  - ‘dog’, ‘dogs’, and ‘doggy’ share the same lemma ‘dog’
- Built on morphological parsing (splitting words into morphemes)

Lemmatization

```
from nltk.stem import WordNetLemmatizer
#nltk.download('wordnet')

tokens = ['cats', 'doing', 'lives', 'has',
'going', 'legislate', 'asocial', 'flew', 'friendly', 'loved']

wordnet_lemmatizer = WordNetLemmatizer()
tokens_lemma = [wordnet_lemmatizer.lemmatize(w) for w in tokens]

print(tokens)
print(tokens_lemma)
```

```
['cats', 'doing', 'lives', 'has', 'going', 'legislate', 'asocial', 'flew', 'friendly', 'loved']
['cat', 'doing', 'life', 'ha', 'going', 'legislate', 'asocial', 'flew', 'friendly', 'loved']
```

Text normalization

Stemming

- Stemming is a simpler but cruder method (lemmatization algorithms are be complex)
- Consists of chopping off word-final affixes
- There are various stemming algorithms as well

Stemming

```
from nltk.stem import PorterStemmer

tokens_stem_p = []

ps = PorterStemmer()
for w in tokens:
    root = ps.stem(w)
    tokens_stem_p.append(root)

print(tokens)
print(tokens_stem_p)
```

```
['cats', 'doing', 'lives', 'has', 'going', 'legislate', 'asocial', 'flew', 'friendly', 'loved']
['cat', 'do', 'live', 'ha', 'go', 'legisl', 'asoci', 'flew', 'friendli', 'love']
```

Text normalization

Stemming

```
from nltk.stem import LancasterStemmer

tokens_stem_l = []

ls = LancasterStemmer()
for w in tokens:
    root = ls.stem(w)
    tokens_stem_l.append(root)

print(tokens)
print(tokens_stem_p)
print(tokens_stem_l)
```

```
['cats', 'doing', 'lives', 'has', 'going', 'legislate', 'asocial', 'flew', 'friendly', 'loved']
['cat', 'do', 'live', 'ha', 'go', 'legisl', 'asoci', 'flew', 'friendli', 'love']
['cat', 'doing', 'liv', 'has', 'going', 'legisl', 'asoc', 'flew', 'friend', 'lov']
```

Text normalization

No single answer to how we should go about this

- It is a good practice to understand what happends under the hood at each step of the process (tokenization -> stop words -> lemmatization/stemming)
- One approach is not only think in advance deductively which combination would be most suitable but also run robustness analyses built on different combinations
- Another (less ideal but easier) approach is to (critically) follow "the norm"

Representing and comparing texts

The BoW model (the bag-of-words model)

- The BoW model is a model of text represented as an unordered collection of words
- Converting running text to BoW data in the form of a DTM (document-term matrix)

Representing and comparing texts

DTM (document-term matrix)

- Count vectors

Document D1	*The child makes the dog happy* the: 2, dog: 1, makes: 1, child: 1, happy: 1
Document D2	*The dog makes the child happy* the: 2, child: 1, makes: 1, dog: 1, happy: 1

	child	dog	happy	makes	the	BoW Vector representations
D1	1	1	1	1	2	**[1,1,1,1,2]**
D2	1	1	1	1	2	**[1,1,1,1,2]**

Representing and comparing texts

Count vectors

- Most easily done in Python with `CountVectorizer` from the ML library `scikit-learn`
- We will use the example of U.S. presidents' inaugural speeches
- https://raw.githubusercontent.com/taegyoon-kim/programming_dhcss_23fw/main/week_14/inaugural_speech_us.csv

```
from sklearn.feature_extraction.text import CountVectorizer

url = 'https://raw.githubusercontent.com/taegyoon-kim/programming_dhcss_23fw/main/week_14/inaugural_speech
inaugural_df = pd.read_csv(url)

print(len(inaugural_df)) # 58 speeches/documents
print(inaugural_df.head()) # the first five
```

```
58
          docnames                                               text
0  1789-Washington  Fellow-Citizens of the Senate and of the House...
1  1793-Washington  Fellow citizens, I am again called upon by the...
2       1797-Adams  When it was first perceived, in early times, t...
3   1801-Jefferson  Friends and Fellow Citizens:    Called upon to...
4   1805-Jefferson  Proceeding, fellow citizens, to that qualifica...
```

Representing and comparing texts

Count vectors

```
vectorizer = CountVectorizer() # many text normalization decisions here
```

- No lemmatization/stemming included!
- `stop_words = None`
- `lowercase = True`
- `max_df`, `min_df`, `ngram_range`, etc.
- See here for a complete set of arguments

Representing and comparing texts

`fit_tranform`

- **fit** means learning the vocabulary of the corpus
- **transform** means creating a matrix and populating with counts

```
dtm = vectorizer.fit_transform(inaugural_df['text'])
dtm # 58 * 9046 (vocabulary)
```

```
<58x9046 sparse matrix of type '<class 'numpy.int64'>'
    with 43638 stored elements in Compressed Sparse Row format>
```

Representing and comparing texts

Shape and sparsity

```
print(dtm.shape)
print(dtm.size) # non-zero elements

import numpy as np
1 - (float(dtm.size) / np.prod(dtm.shape))
```

```
(58, 9046)
43638
```

```
0.9168274032340451
```

Representing and comparing texts

Inside DTM

```
arr_dtm = dtm.toarray()
print(arr_dtm)
```

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

Representing and comparing texts

Inside DTM

```
vocab = vectorizer.get_feature_names_out()
print(vocab[0:30])
print(vocab[-10:])
```

```
['000' '100' '120' '125' '13' '14th' '15th' '16' '1774' '1776' '1778'
 '1780' '1787' '1789' '1790' '1800' '1801' '1812' '1815' '1816' '1817'
 '1818' '1826' '1850' '1861' '1868' '1873' '1880' '1886' '1890']
['your' 'yours' 'yourself' 'yourselves' 'youth' 'youthful' 'zeal'
 'zealous' 'zealously' 'zone']
```

Representing and comparing texts

Inside DTM

```
df_dtm = pd.DataFrame(arr_dtm, columns = vocab)
df_dtm.head()
```

	000	100	120	125	13	14th	15th	16	1774	1776	...	your	yours	yourself	y
0	0	0	0	0	0	1	0	0	0	0	...	9	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	1	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	1	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	7	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	4	0	0	0

Representing and comparing texts

The most common tokens

```
sum_words = arr_dtm.sum(axis = 0) # 1D array (length = 9,046)
words_freq = [(word, sum_words[idx]) for word, idx in vectorizer.vocabulary_.items()]
words_freq = sorted(words_freq, key = lambda x: x[1], reverse = True)

words_freq[0:10]
```

```
[('the', 9821),
 ('of', 6889),
 ('and', 5207),
 ('to', 4423),
 ('in', 2726),
 ('our', 2146),
 ('that', 1748),
 ('we', 1740),
 ('be', 1452),
 ('is', 1430)]
```

Representing and comparing texts

Without stop words

```
vectorizer_nostop = CountVectorizer(stop_words = 'english')
dtm_nostop = vectorizer_nostop.fit_transform(inaugural_df['text'])
dtm_nostop
```

```
<58x8771 sparse matrix of type '<class 'numpy.int64'>'
    with 36009 stored elements in Compressed Sparse Row format>
```

Representing and comparing texts

Without stop words

```
arr_dtm_nostop = dtm_nostop.toarray()
arr_dtm_nostop
```

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]])
```

Representing and comparing texts

Without stop words

```
sum_words_nostop = arr_dtm_nostop.sum(axis = 0) # 1D array (length = 8,771)
words_freq_nostop = [(word, sum_words_nostop[idx]) for word, idx in vectorizer_nostop.vocabulary_.items()]
words_freq_nostop = sorted(words_freq_nostop, key = lambda x: x[1], reverse = True)

words_freq_nostop[0:10]
```

```
[('government', 591),
 ('people', 566),
 ('great', 338),
 ('world', 337),
 ('states', 324),
 ('nation', 311),
 ('shall', 310),
 ('country', 303),
 ('peace', 254),
 ('new', 252)]
```

Representing and comparing texts

TF (Term Frequency) - IDF (Inverse Document Frequency) vectors

- Counter vectors consider the frequencies of words
- However, some words are too frequent: *the*, *a*, *an*, etc.
- We want to weight how unique a word is in the corpus

Representing and comparing texts

TF-IDF vectors

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

TF-IDF

Term x within document y

$tf_{x,y}$ = frequency of x in y

df_x = number of documents containing x

N = total number of documents

Representing and comparing texts

TF-IDF vectors

```
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = ['can you fly', 'can you sleep']
tf_idf_vectorizer = TfidfVectorizer()
tf_idf_matrix = tf_idf_vectorizer.fit_transform(corpus)
print(tf_idf_vectorizer.get_feature_names_out())
print(tf_idf_matrix.toarray())
```

```
['can' 'fly' 'sleep' 'you']
[[0.50154891 0.70490949 0.         0.50154891]
 [0.50154891 0.         0.70490949 0.50154891]]
```

Representing and comparing texts

Let's use news titles data

```
import pandas as pd

url = 'https://raw.githubusercontent.com/taegyoon-kim/programming_dhcss_23fw/main/week_14/news_title.csv'
news_df = pd.read_csv(url, sep = ';')
news_df = news_df.sample(1000) # random sample of 1,000
```

Representing and comparing texts

TF-IDF vectors

```
news_df.head()
```

	No	News Title	Category
20678	20679	LG G Watch hands-on (video)	Technology
8178	8179	Dear Robin Thicke, leave it now you're embarra...	Entertainment
24335	24336	Barbara Walters says she wants to 'lounge in b...	Entertainment
47546	47547	Alibaba to buy SingPost stake for USD249m	Business
62607	62608	Microsoft scrambles to fix bug in Internet Exp...	Technology

Representing and comparing texts

TF-IDF vectors

```
from sklearn.feature_extraction.text import TfidfVectorizer

tf_idf_vectorizer = TfidfVectorizer()
tf_idf_matrix = tf_idf_vectorizer.fit_transform(news_df['News Title'])

feature_names = tf_idf_vectorizer.get_feature_names_out()

tf_idf_df = pd.DataFrame(
  tf_idf_matrix.toarray(),
  columns = feature_names,
  index = news_df['No'])
```

Representing and comparing texts

TF-IDF vectors

```
tf_idf_df
```

	00	04	09	10	100	1000	10c	10news	11	115	…	zynga	œvalidatesâ
No													
20679	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	…	0.0	0.0
8179	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	…	0.0	0.0
24336	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	…	0.0	0.0
47547	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	…	0.0	0.0
62608	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	…	0.0	0.0
…	…	…	…	…	…	…	…	…	…	…	…	…	…
9639	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	…	0.0	0.0
20813	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	…	0.0	0.0
21486	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	…	0.0	0.0
36111	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	…	0.0	0.0
1780	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	…	0.0	0.0

Representing and comparing texts

Cosine similarity

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2} \sqrt{\sum\limits_{i=1}^{n} B_i^2}}$$

- It ranges from -1 (opposite) to 0 (orthogonal) to 1 (identical)

Representing and comparing texts

Creating a cosine similarity matrix

```
from sklearn.metrics.pairwise import cosine_similarity

cos_sim_mat = cosine_similarity(tf_idf_df, tf_idf_df)
cos_sim_df = pd.DataFrame(
  cos_sim_mat,
  columns = news_df['No'],
  index = news_df['No'])
cos_sim_df.head()
```

No No	20679	8179	24336	47547	62608	30361	50778	10315
20679	1.000000	0.0	0.042651	0.000000	0.000000	0.0	0.0	0.05419
8179	0.000000	1.0	0.000000	0.000000	0.000000	0.0	0.0	0.00000
24336	0.042651	0.0	1.000000	0.020687	0.058588	0.0	0.0	0.02991
47547	0.000000	0.0	0.020687	1.000000	0.025350	0.0	0.0	0.03769
62608	0.000000	0.0	0.058588	0.025350	1.000000	0.0	0.0	0.00000

Representing and comparing texts

Representing the matrix as an edge list

```
edges = []
for i in range(len(cos_sim_df)):
  for j in range(i+1, len(cos_sim_df)):
    weight = cos_sim_df.iloc[i, j]
    edges.append((cos_sim_df.index[i], cos_sim_df.columns[j], weight))

edges_df = pd.DataFrame(edges, columns = ["source", "target", "weight"])
edges_df.head()
```

	source	target	weight
0	20679	8179	0.000000
1	20679	24336	0.042651
2	20679	47547	0.000000
3	20679	62608	0.000000
4	20679	30361	0.000000

Representing and comparing texts

Add category labels

```
edges_df_m1 = pd.merge(
  edges_df, news_df[['No', 'Category']],
  left_on = 'source',
  right_on = 'No',
  how = 'left')

edges_df_m2 = pd.merge(
  edges_df_m1, news_df[['No', 'Category']],
  left_on = 'target',
  right_on = 'No',
  how = 'left')

edges_df_m2['comb'] = edges_df_m2['Category_x'] + '-' + edges_df_m2['Category_y']

edges_df_m2['comb']
```

```
0         Technology-Entertainment
1         Technology-Entertainment
2              Technology-Business
3            Technology-Technology
4         Technology-Entertainment
                    ...
499495    Entertainment-Technology
499496    Entertainment-Technology
499497       Technology-Technology
499498       Technology-Technology
499499       Technology-Technology
Name: comb, Length: 499500, dtype: object
```

Representing and comparing texts

Average cosine similarity

```
edges_df_m2.groupby('comb')['weight'].agg('mean').sort_values(ascending = False)
```

```
comb
Medical-Medical                0.012561
Technology-Technology          0.011952
Entertainment-Entertainment    0.010260
Business-Business              0.008361
Technology-Business            0.007051
Technology-Medical             0.006931
Technology-Entertainment       0.006814
Entertainment-Technology       0.006655
Entertainment-Medical          0.006523
Business-Technology            0.006264
Medical-Technology             0.006230
Medical-Entertainment          0.006214
Business-Medical               0.005985
Entertainment-Business         0.005908
Medical-Business               0.005513
Business-Entertainment         0.005236
Name: weight, dtype: float64
```

Word embeddings

Vector semantics

- Vector semantics starts with learning representations of the meaning of words
- Distributional hypothesis: words that occur in similar contexts tend to have similar meanings (from the 1950s in linguists' work such as Harris 1954)
- The very beginning of vector semantics dates back at least to Osgood et al. (1957) where words were represented in a 3-dimensional vector space (valence, arousal, dominance)

	Valence	Arousal	Dominance
music	7.67	5.57	6.5
heartbreak	2.45	5.65	3.58

Word embeddings

What are word embeddings?

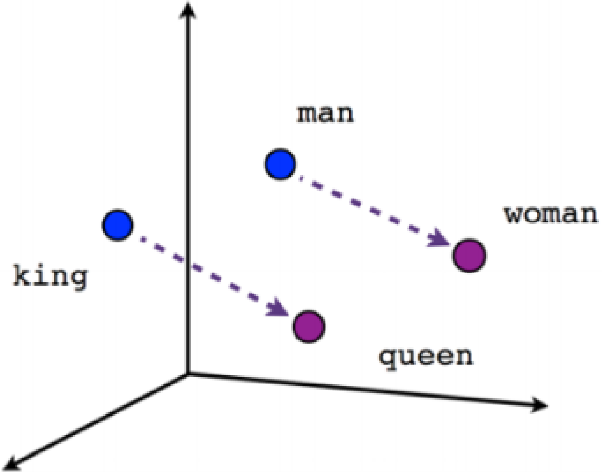
- Word embeddings refer to short dense vectors for representing words (Word2vec, BERT, etc.)
- They are used to position a word as a point in a multidimensional semantic space

Word embeddings

What are word embeddings?

- A paradigm shift from the traditional BoW approach
- Employ neural networks to learn word representations from a large corpus
- Explicitly considers context (not “contextual embeddings” like BERT though)

Word embeddings



Word embeddings

What are word embeddings?

- Many useful applications
  - Discover the relationships between words (compute their similarity)
  - Track changes in meaning
  - Other applications like text classification or clustering
- Note that dimensions are not interpretable

Word embeddings

Word2Vec

- A set of algorithms (CBOW and SGNS) for learning word representations from a corpus

Word embeddings

Pre-trained embeddings

- Word2Vec: https://code.google.com/archive/p/word2vec/
- GloVe: https://nlp.stanford.edu/projects/glove/
- FastText: https://fasttext.cc/docs/en/english-vectors.html