16_natural-language-processing

June 25, 2022

CPSC 330
Applied Machine Learning

1 Lecture 16: Introduction to natural language processing

UBC 2022 Summer

Instructor: Mehrdad Oveisi

1.1 Imports

[1]:
```
import os
import re
import string
import sys
import time
from collections import Counter, defaultdict

import IPython
import nltk
import numpy as np
import numpy.random as npr
import pandas as pd
from IPython.display import HTML
from ipywidgets import interactive
from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize, word_tokenize
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
```

1.2 Learning objectives

- Broadly explain what is natural language processing (NLP).
- Name some common NLP applications.

- Explain the general idea of a vector space model.
- Explain the difference between different word representations: term-term co-occurrence matrix representation and Word2Vec representation.
- Describe the reasons and benefits of using pre-trained embeddings.
- Load and use pre-trained word embeddings to find word similarities and analogies.
- Demonstrate biases in embeddings and learn to watch out for such biases in pre-trained embeddings.
- Use word embeddings in text classification and document clustering using `spaCy`.
- Explain the general idea of topic modeling.
- Describe the input and output of topic modeling.
- Carry out basic text preprocessing using `spaCy`.

1.3 What is Natural Language Processing (NLP)?

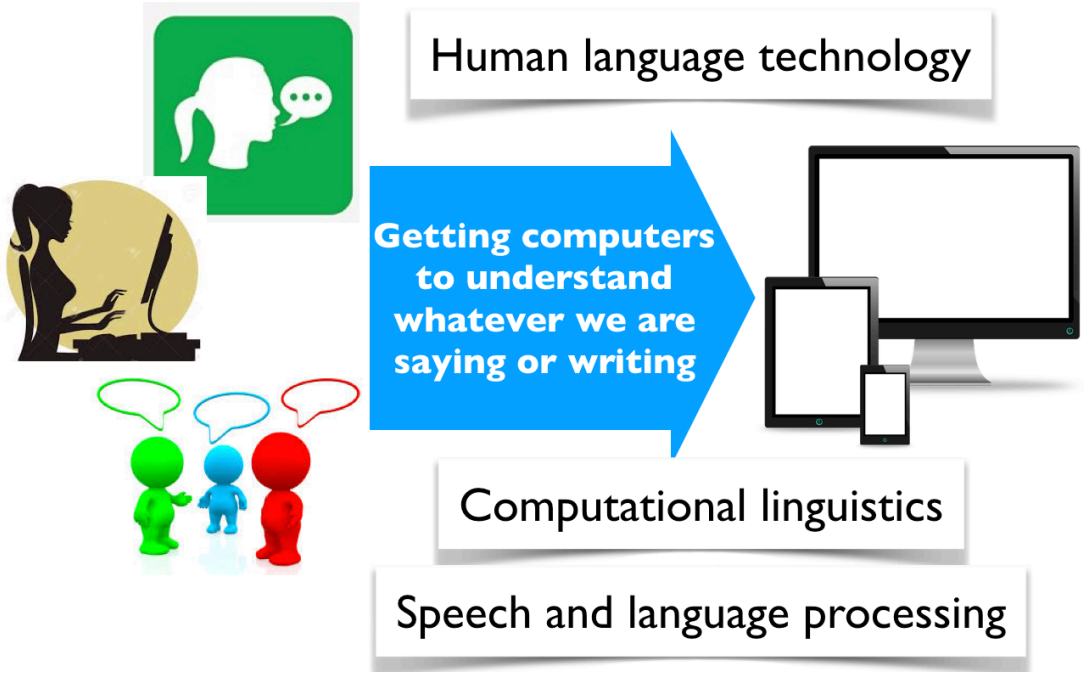
- What should a search engine return when asked the following question?

Who is Panini?

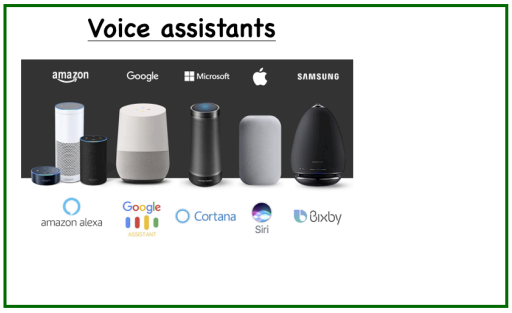
1.3.1 What is Natural Language Processing (NLP)?

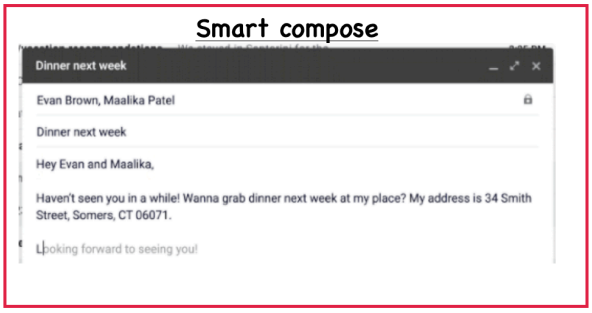
How often do you search everyday?

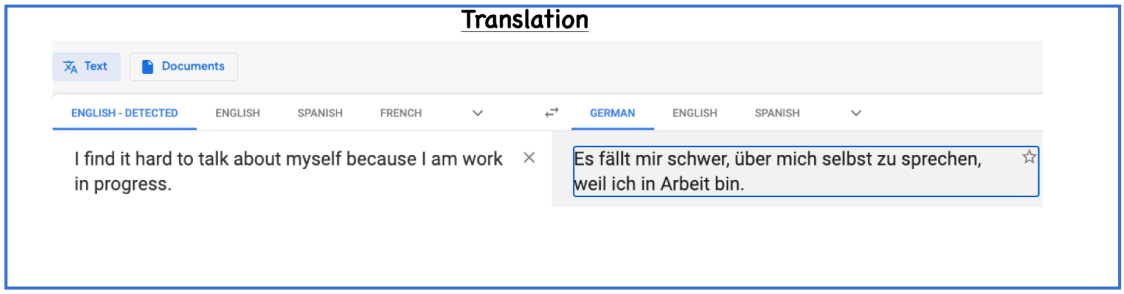
1.3.2 What is Natural Language Processing (NLP)?



1.3.3 Everyday NLP applications







1.3.4 NLP in news

Often you'll NLP in news. Some examples: - How suicide prevention is getting a boost from artificial intelligence - Meet GPT-3. It Has Learned to Code (and Blog and Argue). - How Do You Know a Human Wrote This? - ...

1.3.5 Why is NLP hard?

- Language is complex and subtle.
- Language is ambiguous at different levels.
- Language understanding involves common-sense knowledge and real-world reasoning.
- All the problems related to representation and reasoning in artificial intelligence arise in this domain.

1.3.6 Example: Lexical ambiguity

Who is Panini?

1.3.7 Example: Referential ambiguity

If the **baby** does not thrive on raw **milk**, boil **it**.

it = ?

1.3.8 Ambiguous news headlines

KICKING BABY CONSIDERED TO BE HEALTHY

- **kicking** is used as an adjective or a verb?

MILK DRINKERS ARE TURNING TO POWDER

- **turning** means becoming or take up?

1.3.9 Overall goal

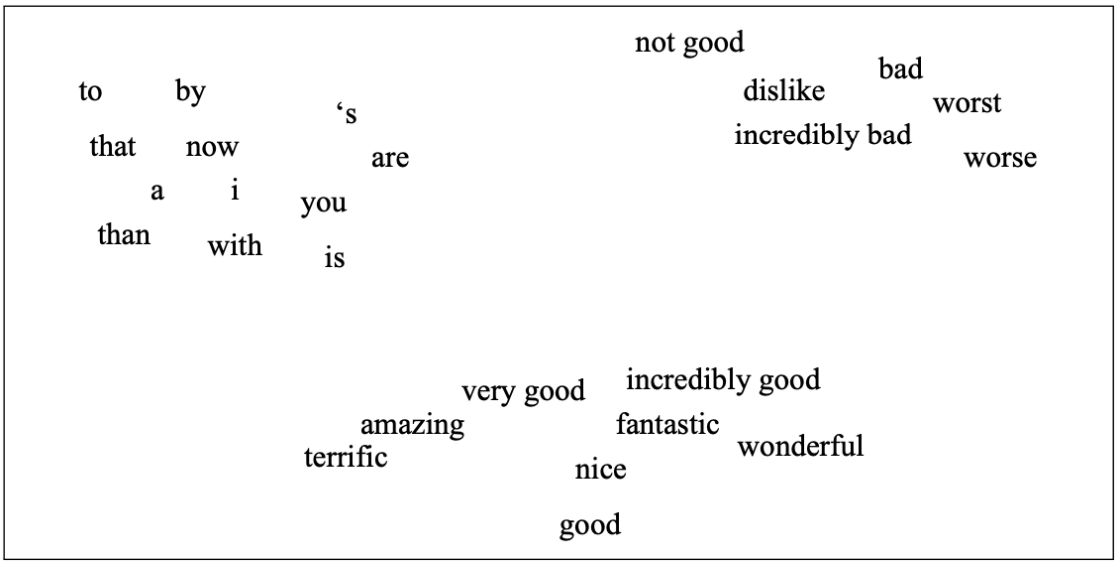
- Give you a quick introduction to this important field in artificial intelligence which extensively uses machine learning.

Today's plan

- Word embeddings
- Topic modeling
- Basic text preprocessing

1.4 Word Embeddings

- The idea is to represent word meaning so that similar words are close together.



(Attribution: Jurafsky and Martin 3rd edition)

1.4.1 Why do we care about word representation?

- So far we have been talking about sentence or document representation.
- Now we are going one step back and talking about **word representation**.
- Although word representation cannot be directly used in text classification tasks such as sentiment analysis using tradition ML models, it's good to know about **word embeddings** because they are so widely used.
- They are quite useful in more advanced machine learning models such as recurrent neural networks.

1.4.2 Word meaning

- A favourite topic of philosophers for centuries.
- An example from legal domain: Are hockey gloves gloves or "articles of plastics"?

Canada (A.G.) v. Igloo Vikski Inc. was a tariff code case that made its way to the SCC (Supreme Court of Canada). The case disputed the definition of hockey gloves as either gloves or as "articles of plastics."

1.4.3 Word meaning: ML and NLP view

- Modeling word meaning that allows us to
  - draw useful inferences to solve meaning-related problems
  - find relationship between words,
    - E.g., which words are similar, which ones have positive or negative connotations

1.5 Word representations

1.5.1 How do we represent words?

- Suppose you are building a question answering system and you are given the following question and three candidate answers.

- What kind of relationship between words would we like our representation to capture in order to arrive at the correct answer?

Question: How tall is Machu Picchu?

```
<p style="font-size:30px"><b>Candidate 1:</b> Machu Picchu is 13.164 degrees south of the equa
```

Candidate 2: The official height of Machu Picchu is 2,430 m.

Candidate 3: Machu Picchu is 80 kilometres (50 miles) northwest of Cusco.

1.5.2 Need a representation that captures *relationships between words.*

- We will be looking at two such representations.
  1. Sparse representation with **term-term co-occurrence matrix**
  2. Dense representation with **Word2Vec**
- Both are based on two ideas: **distributional hypothesis** and **vector space model**.

1.5.3 Distributional hypothesis

You shall know a word by the company it keeps.

Firth, 1957

If A and B have almost identical environments we say that they are synonyms.

Harris, 1954

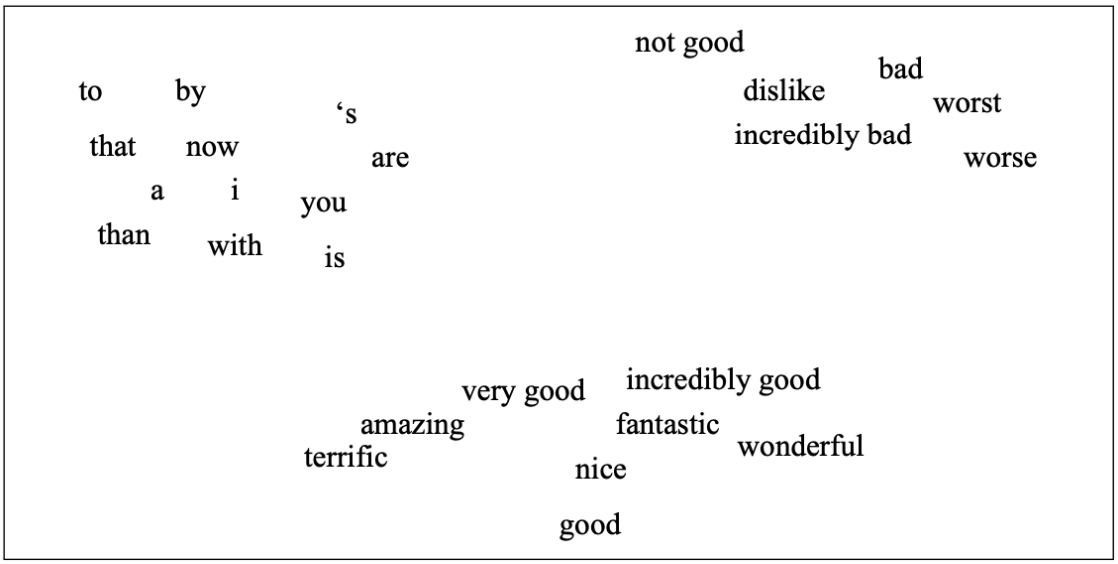
Example:

- Her **child** loves to play in the playground.
- Her **kid** loves to play in the playground.

1.5.4 Vector space model

- Model the meaning of a word by placing it into a vector space.

- A standard way to represent meaning in NLP
- The idea is to create **embeddings of words** so that distances among words in the vector space indicate the relationship between them.



(Attribution: Jurafsky and Martin 3rd edition)

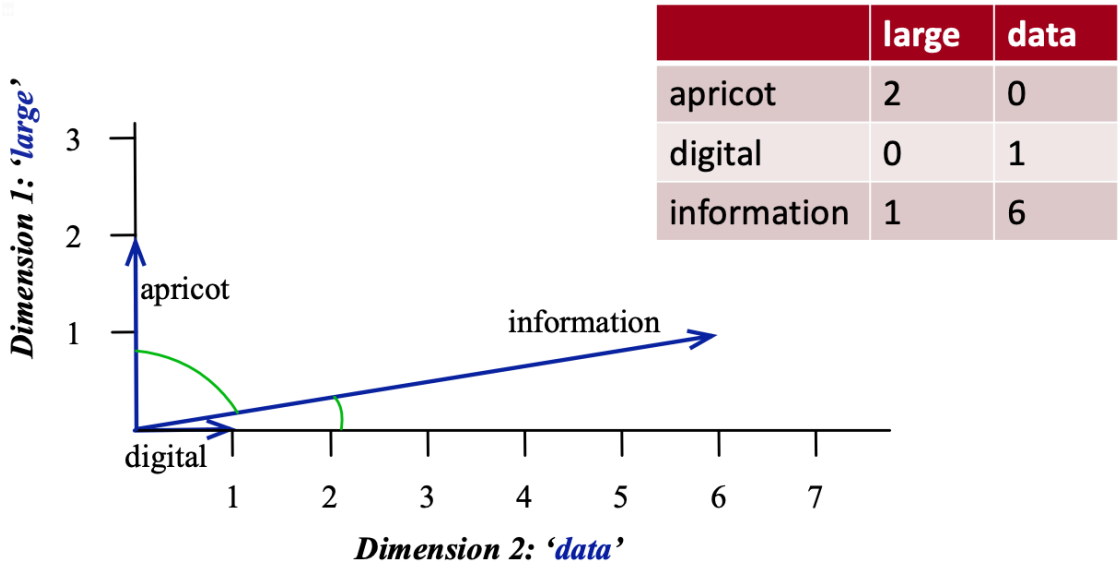
1.5.5 Term-term co-occurrence matrix

- So far we have been talking about documents and we created document-term co-occurrence matrix (e.g., bag-of-words representation of text).
- We can also do this with words. The idea is to go through a corpus of text, keeping a count of all of the words that appear in context of each word (within a window).
- An example:

	large	data
apricot	2	0
digital	0	1
information	1	6

(Credit: Jurafsky and Martin 3rd edition)

1.5.6 Visualizing word vectors and similarity

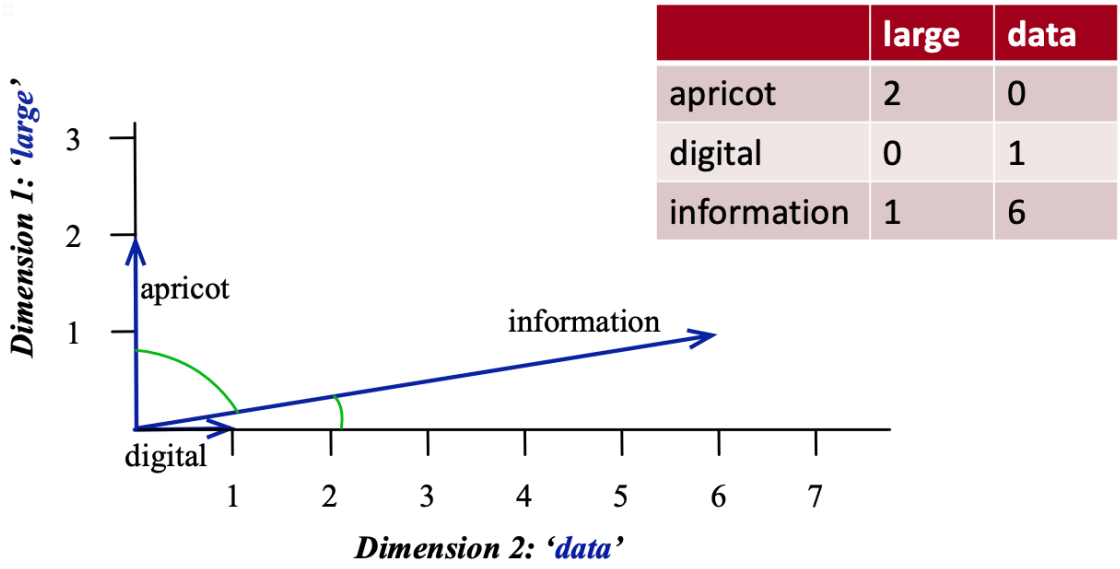


	large	data
apricot	2	0
digital	0	1
information	1	6

(Credit: Jurafsky and Martin 3rd edition)

- The similarity is calculated using dot products between word vectors.
  - Example: $\vec{\text{digital}}.\vec{\text{information}} = 0 \times 1 + 1 \times 6 = 6$
  - Higher the dot product more similar the words.

1.5.7 Visualizing word vectors and similarity



	large	data
apricot	2	0
digital	0	1
information	1	6

(Credit: Jurafsky and Martin 3rd edition)

- The similarity is calculated using dot products between word vectors.
  - Example: $\vec{\text{digital}}.\vec{\text{information}} = 0 \times 1 + 1 \times 6 = 6$
  - Higher the dot product more similar the words.

- We can also calculate a normalized version of dot products.

$$similarity_{cosine}(w_1, w_2) = \frac{w_1.w_2}{\|w_1\|_2 \|w_2\|_2}$$

1.5.8 Let's build term-term co-occurrence matrix for our text.

[2]:
```
# You need to run the following line once
# nltk.download('stopwords')  # You need to run this at least once
```

[3]:
```
sys.path.append("code/.")

from comat import CooccurrenceMatrix
from preprocessing import MyPreprocessor

corpus = [
    "How tall is Machu Picchu?",
    "Machu Picchu is 13.164 degrees south of the equator.",
    "The official height of Machu Picchu is 2,430 m.",
    "Machu Picchu is 80 kilometres (50 miles) northwest of Cusco.",
    "It is 80 kilometres (50 miles) northwest of Cusco, on the crest of the␣
 ↪mountain Machu Picchu, located about 2,430 metres (7,970 feet) above mean␣
 ↪sea level, over 1,000 metres (3,300 ft) lower than Cusco, which has an␣
 ↪elevation of 3,400 metres (11,200 ft).",
]
pp = MyPreprocessor()
pp_corpus = pp.preprocess_corpus(corpus)
cm = CooccurrenceMatrix(pp_corpus)
vocab, comat = cm.fit_transform()
words = [
    key for key, value in sorted(vocab.items(), key=lambda item: (item[1],␣
 ↪item[0]))
]
df = pd.DataFrame(comat.todense(), columns=words, index=words, dtype=np.int8)
df.head()
```

[3]:

	tall	machu	picchu	13.164	degrees	south	equator	official	\
tall	0	1	1	0	0	0	0	0	
machu	1	0	5	1	1	0	0	1	
picchu	1	5	0	1	1	1	0	1	
13.164	0	1	1	0	1	1	1	0	
degrees	0	1	1	1	0	1	1	0	

	height	2,430	…	mean	sea	level	1,000	3,300	ft	lower	\
tall	0	0	…	0	0	0	0	0	0	0	
machu	1	2	…	0	0	0	0	0	0	0	
picchu	1	2	…	0	0	0	0	0	0	0	
13.164	0	0	…	0	0	0	0	0	0	0	

```
degrees       0      0  …     0    0       0      0      0   0       0

          elevation  3,400  11,200
tall              0      0       0
machu             0      0       0
picchu            0      0       0
13.164            0      0       0
degrees           0      0       0

[5 rows x 32 columns]
```

[4]:
```
import warnings
from sklearn.metrics.pairwise import cosine_similarity


def similarity(word1, word2):
    """
    Returns similarity score between word1 and word2
    Arguments
    ---------
    word1 -- (str)
        The first word
    word2 -- (str)
        The second word

    Returns
    --------
    None. Prints the similarity score between word1 and word2.
    """
    vec1 = cm.get_word_vector(word1).todense().flatten()
    vec2 = cm.get_word_vector(word2).todense().flatten()
    v1 = np.squeeze(np.asarray(vec1))
    v2 = np.squeeze(np.asarray(vec2))
    warnings.simplefilter(action="ignore", category=FutureWarning)
    print(
        "The dot product between %s and %s is %0.2f and cosine similarity is %0.
 ↪2f"
        % (word1, word2, np.dot(v1, v2), cosine_similarity(vec1, vec2))
    )
    warnings.simplefilter(action="default", category=FutureWarning)


similarity("tall", "height")
similarity("tall", "official")

### Not very reliable similarity scores because we used only 4 sentences.
```

```
The dot product between tall and height is 2.00 and cosine similarity is 0.71
```

```
The dot product between tall and official is 2.00 and cosine similarity is 0.82
```

- We are able to capture some similarities between words now.
- That said similarities do not make much sense in the toy example above because we're using a tiny corpus.
- To find meaningful patterns of similarities between words, we need a large corpus.
- Let's try a bit larger corpus and check whether the similarities make sense.

[5]:
```
import wikipedia
from nltk.tokenize import sent_tokenize, word_tokenize

corpus = []

queries = ["Machu Picchu", "human stature", "mountain"]

for i in range(len(queries)):
    sents = sent_tokenize(wikipedia.page(queries[i]).content)
    corpus.extend(sents)
print("Number of sentences in the corpus: ", len(sents))
```

```
Number of sentences in the corpus:  129
```

[6]:
```
pp = MyPreprocessor()
pp_corpus = pp.preprocess_corpus(corpus)
cm = CooccurrenceMatrix(pp_corpus)
vocab, comat = cm.fit_transform()
words = [
    key for key, value in sorted(vocab.items(), key=lambda item: (item[1],␣
 ↪item[0]))
]
df = pd.DataFrame(comat.todense(), columns=words, index=words, dtype=np.int8)
df.head()
```

[6]:

	machu	picchu	15th-century	inca	citadel	located	eastern	\
machu	0	93	1	6	2	2	1	
picchu	93	0	1	7	4	1	1	
15th-century	1	1	0	1	1	1	0	
inca	6	7	1	0	1	1	1	
citadel	2	4	1	1	0	1	1	

	cordillera	southern	peru	…	contender	6,600	ranges	\
machu	0	1	5	…	0	0	0	
picchu	0	1	4	…	0	0	0	
15th-century	0	0	0	…	0	0	0	
inca	0	0	2	…	0	0	0	
citadel	1	0	0	…	0	0	0	

	lists	accessible	encyclopædia	britannica	11th	quotations	\

```
machu            0          0          0          0     0          0
picchu           0          0          0          0     0          0
15th-century     0          0          0          0     0          0
inca             0          0          0          0     0          0
citadel          0          0          0          0     0          0

              wikiquote
machu                 0
picchu                0
15th-century          0
inca                  0
citadel               0

[5 rows x 3622 columns]
```

[7]:
```
similarity("tall", "height")
similarity("tall", "official")
```

```
The dot product between tall and height is 28.00 and cosine similarity is 0.10
The dot product between tall and official is 0.00 and cosine similarity is 0.00
```

1.5.9 Sparse vs. dense word vectors

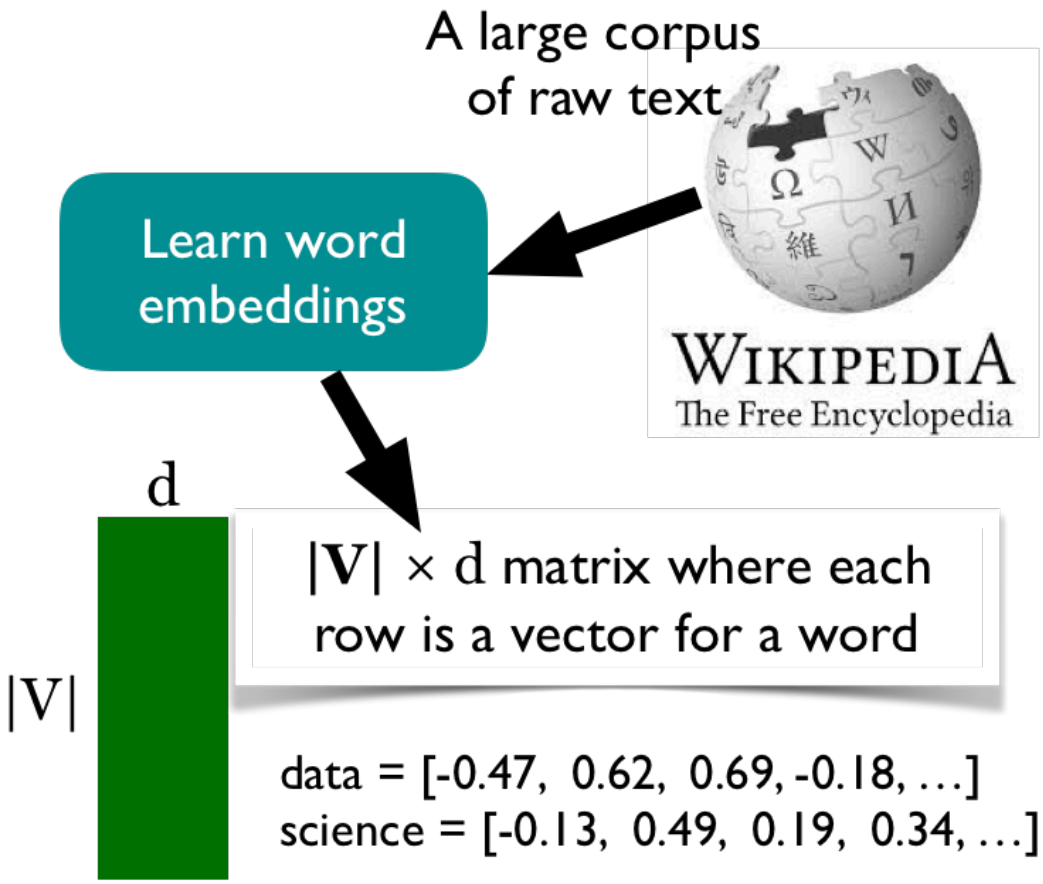
- Term-term co-occurrence matrices are long and sparse.
  - length |V| is usually large (e.g., $> 50{,}000$)
  - most elements are zero
- That is OK because there are efficient ways to deal with sparse matrices.

1.5.10 Alternative

- Learn short (~100 to 1000 dimensions) and dense vectors.
- Short vectors may be easier to train with ML models (less weights to train).
- They may generalize better.
- In practice they work much better!

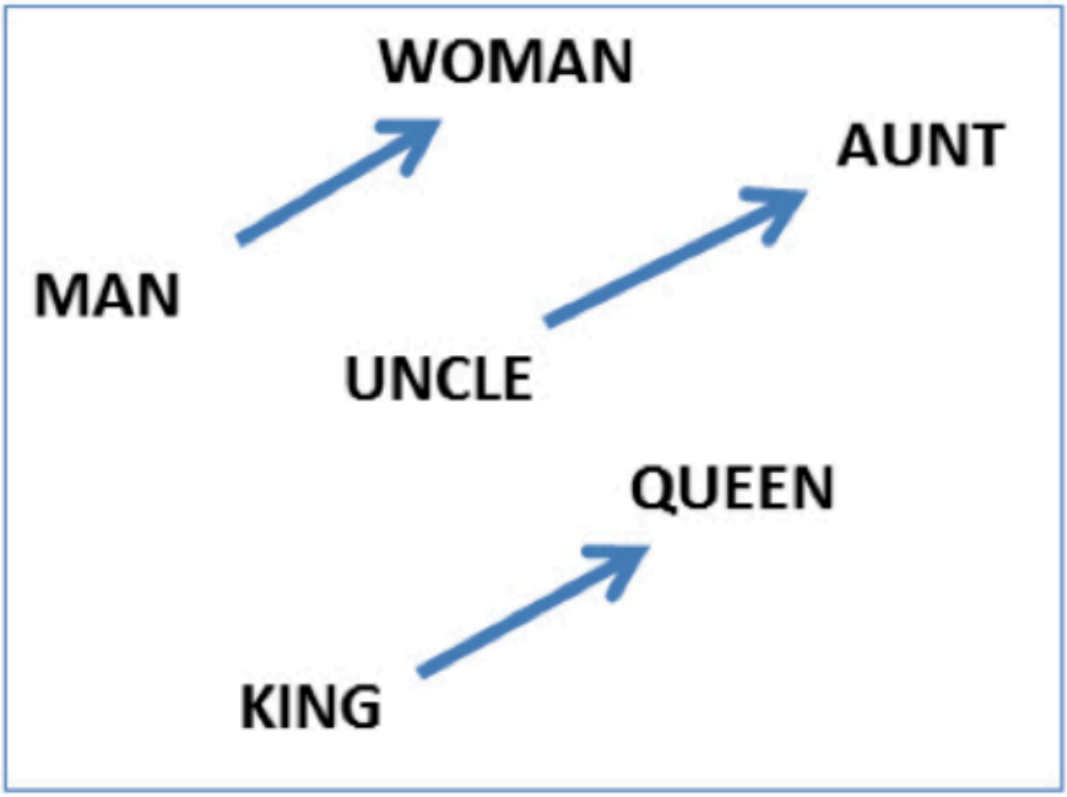
1.5.11 Word2Vec

- A family of algorithms to create dense word embeddings



1.5.12 Success of Word2Vec

- Able to capture complex relationships between words.
- Example: What is the word that is similar to **WOMAN** in the same sense as **KING** is similar to **MAN**?
- Perform a simple algebraic operations with the vector representation of words. $\vec{X} = \vec{\text{KING}} - \vec{\text{MAN}} + \vec{\text{WOMAN}}$
- Search in the vector space for the word closest to $\vec{X}$ measured by cosine distance.



(Credit: Mikolov et al. 2013)

- We can create a dense representation with a library called `gensim`.

```
conda install -n cpsc330 -c anaconda gensim
```

[8]:
```
from gensim.models import Word2Vec

sentences = [["cat", "say", "meow"], ["dog", "say", "woof"]]
model = Word2Vec(sentences, min_count=1)
```

Let's look at the word vector of the word *cat*.

[9]:
```
model.wv["cat"]
```

[9]:
```
array([-0.00713902,  0.00124103, -0.00717672, -0.00224462,  0.0037193 ,
        0.00583312,  0.00119818,  0.00210273, -0.00411039,  0.00722533,
       -0.00630704,  0.00464721, -0.00821997,  0.00203647, -0.00497705,
       -0.00424769, -0.00310899,  0.00565521,  0.0057984 , -0.00497465,
        0.00077333, -0.00849578,  0.00780981,  0.00925729, -0.00274233,
        0.00080022,  0.00074665,  0.00547788, -0.00860608,  0.00058445,
        0.00686942,  0.00223159,  0.00112468, -0.00932216,  0.00848237,
       -0.00626413, -0.00299237,  0.00349379, -0.00077263,  0.00141129,
        0.00178199, -0.0068289 , -0.00972481,  0.00904058,  0.00619805,
       -0.00691293,  0.00340348,  0.00020606,  0.00475374, -0.00711994,
```

```
 0.00402695,  0.00434743,  0.00995737, -0.00447374, -0.00138927,
-0.00731732, -0.00969783, -0.00908026, -0.00102276, -0.00650329,
 0.00484973, -0.00616403,  0.00251919,  0.00073944, -0.00339216,
-0.00097922,  0.00997912,  0.00914589, -0.00446183,  0.00908303,
-0.00564176,  0.00593092, -0.00309722,  0.00343175,  0.00301723,
 0.00690046, -0.00237388,  0.00877504,  0.00758943, -0.00954765,
-0.00800821, -0.0076379 ,  0.00292326, -0.00279472, -0.00692952,
-0.00812826,  0.00830918,  0.00199049, -0.00932802, -0.00479272,
 0.00313674, -0.00471321,  0.00528084, -0.00423344,  0.00264179,
-0.00804569,  0.00620989,  0.00481889,  0.00078719,  0.00301345],
dtype=float32)
```

What's the most similar word to the word *cat*?

[10]:
```
model.wv.most_similar("cat")
```

[10]:
```
[('dog', 0.17018885910511017),
 ('woof', 0.004503009375184774),
 ('say', -0.027750369161367416),
 ('meow', -0.04461709037423134)]
```

This is good. But if you want good and meaningful representations of words you need to train models on a large corpus such as the whole Wikipedia, which is computationally intensive.

So instead of training our own models, we use the **pre-trained embeddings**. These are the word embeddings people have trained embeddings on huge corpora and made them available for us to use.

Let's try out Google news pre-trained word vectors.

[11]:
```
# It'll take a while to run this when you try it out for the first time.
import gensim.downloader as api

google_news_vectors = api.load("word2vec-google-news-300")
```

[12]:
```
print("Size of vocabulary: ", len(google_news_vectors))
```

```
Size of vocabulary:  3000000
```

- `google_news_vectors` above has 300 dimensional word vectors for 3,000,000 unique words from Google news.
- What can we do with these word vectors?

1.5.13 Finding similar words

- Given word w, search in the vector space for the word closest to w as measured by cosine distance.

[13]:
```
google_news_vectors.most_similar("UBC")
```

[13]:
```
[('UVic', 0.7886475920677185),
 ('SFU', 0.7588528394699097),
 ('Simon_Fraser', 0.7356574535369873),
 ('UFV', 0.6880435943603516),
 ('VIU', 0.6778583526611328),
 ('Kwantlen', 0.6771429181098938),
 ('UBCO', 0.6734487414360046),
 ('UPEI', 0.6731126308441162),
 ('UBC_Okanagan', 0.6709133982658386),
 ('Lakehead_University', 0.6622507572174072)]
```

[14]:
```
google_news_vectors.most_similar("information")
```

[14]:
```
[('info', 0.7363681793212891),
 ('infomation', 0.680029571056366),
 ('infor_mation', 0.673384964466095),
 ('informaiton', 0.6639009118080139),
 ('informa_tion', 0.6601256728172302),
 ('informationon', 0.6339334845542908),
 ('informationabout', 0.6320980191230774),
 ('Information', 0.6186580657958984),
 ('informaion', 0.6093292236328125),
 ('details', 0.6063088774681091)]
```

If you want to extract all documents containing information or similar words, you could use this information.

1.5.14 Finding similarity scores between words

[15]:
```
google_news_vectors.similarity("Canada", "hockey")
```

[15]:
```
0.27610135
```

[16]:
```
google_news_vectors.similarity("China", "hockey")
```

[16]:
```
0.060384665
```

[17]:
```
word_pairs = [
    ("height", "tall"),
    ("pineapple", "mango"),
    ("pineapple", "juice"),
    ("sun", "robot"),
    ("GPU", "lion"),
]
for pair in word_pairs:
    print(
        "The similarity between %s and %s is %0.3f"
        % (pair[0], pair[1], google_news_vectors.similarity(pair[0], pair[1]))
```

```
    )
```

```
The similarity between height and tall is 0.473
The similarity between pineapple and mango is 0.668
The similarity between pineapple and juice is 0.418
The similarity between sun and robot is 0.029
The similarity between GPU and lion is 0.002
```

[18]:
```
def analogy(word1, word2, word3, model=google_news_vectors):
    """
    Returns analogy word using the given model.

    Parameters
    --------------
    word1 : (str)
        word1 in the analogy relation
    word2 : (str)
        word2 in the analogy relation
    word3 : (str)
        word3 in the analogy relation
    model :
        word embedding model

    Returns
    ---------------
        pd.dataframe
    """
    print("%s : %s :: %s : ?" % (word1, word2, word3))
    sim_words = model.most_similar(positive=[word3, word2], negative=[word1])
    return pd.DataFrame(sim_words, columns=["Analogy word", "Score"])
```

[19]:
```
analogy("man", "king", "woman")
```

```
man : king :: woman : ?
```

[19]:

	Analogy word	Score
0	queen	0.711819
1	monarch	0.618967
2	princess	0.590243
3	crown_prince	0.549946
4	prince	0.537732
5	kings	0.523684
6	Queen_Consort	0.523595
7	queens	0.518113
8	sultan	0.509859
9	monarchy	0.508741

[20]:
```
analogy("Montreal", "Canadiens", "Vancouver")
```

```
Montreal : Canadiens :: Vancouver : ?
```

[20]:

	Analogy word	Score
0	Canucks	0.821327
1	Vancouver_Canucks	0.750401
2	Calgary_Flames	0.705470
3	Leafs	0.695783
4	Maple_Leafs	0.691617
5	Thrashers	0.687504
6	Avs	0.681716
7	Sabres	0.665307
8	Blackhawks	0.664625
9	Habs	0.661023

[21]:
```
analogy("Toronto", "UofT", "Vancouver")
```

```
Toronto : UofT :: Vancouver : ?
```

[21]:

	Analogy word	Score
0	SFU	0.579245
1	UVic	0.576921
2	UBC	0.571431
3	Simon_Fraser	0.543464
4	Langara_College	0.541347
5	UVIC	0.520495
6	Grant_MacEwan	0.517273
7	UFV	0.514150
8	Ubyssey	0.510421
9	Kwantlen	0.503807

1.5.15 Examples of semantic and syntactic relationships

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter
Adjective to adverb	apparent	apparently	rapid	rapidly
Opposite	possibly	impossibly	ethical	unethical
Comparative	great	greater	tough	tougher
Superlative	easy	easiest	lucky	luckiest
Present Participle	think	thinking	read	reading
Nationality adjective	Switzerland	Swiss	Cambodia	Cambodian
Past tense	walking	walked	swimming	swam
Plural nouns	mouse	mice	dollar	dollars
Plural verbs	work	works	speak	speaks

(Credit: Mikolov 2013)

1.5.16 Implicit biases and stereotypes in word embeddings

- Reflect gender stereotypes present in broader society.
- They may also amplify these stereotypes because of their widespread usage.
- See the paper Man is to Computer Programmer as Woman is to ….

[22]:
```
analogy("man", "computer_programmer", "woman")
```

```
man : computer_programmer :: woman : ?
```

[22]:

	Analogy word	Score
0	homemaker	0.562712
1	housewife	0.510505
2	graphic_designer	0.505180
3	schoolteacher	0.497949
4	businesswoman	0.493489
5	paralegal	0.492551
6	registered_nurse	0.490797
7	saleswoman	0.488163
8	electrical_engineer	0.479773
9	mechanical_engineer	0.475540

Most of the modern embeddings are de-biased.

1.5.17 Other pre-trained embeddings

A number of pre-trained word embeddings are available. The most popular ones are:

- Word2Vec
  - trained on several corpora using the word2vec algorithm
- wikipedia2vec
  - pretrained embeddings for 12 languages
- GloVe
  - trained using the GloVe algorithm
  - published by Stanford University
- fastText pre-trained embeddings for 294 languages
  - trained using the fastText algorithm
  - published by Facebook

Note These pre-trained word vectors are of big size (in several gigabytes).

Here is a list of all pre-trained embeddings available with `gensim`.

[23]:
```
import gensim.downloader

print(*gensim.downloader.info()["models"].keys(), sep=", ")
```

```
fasttext-wiki-news-subwords-300, conceptnet-numberbatch-17-06-300, word2vec-
ruscorpora-300, word2vec-google-news-300, glove-wiki-gigaword-50, glove-wiki-
gigaword-100, glove-wiki-gigaword-200, glove-wiki-gigaword-300, glove-
twitter-25, glove-twitter-50, glove-twitter-100, glove-twitter-200,
__testing_word2vec-matrix-synopsis
```

1.5.18 Word vectors with spaCy

- spaCy also gives you access to word vectors with bigger models: `en_core_web_md` or `en_core_web_lr`
- `spaCy`'s pre-trained embeddings are trained on OntoNotes corpus.
- This corpus has a collection of different styles of texts such as telephone conversations, newswire, newsgroups, broadcast news, broadcast conversation, weblogs, religious texts.
- Let's try it out.

[24]:
```
import spacy

nlp = spacy.load("en_core_web_md")

doc = nlp("pineapple")
doc.vector
```

[24]:
```
array([-6.3358e-01,  1.2266e-01,  4.7232e-01, -2.2974e-01, -2.6307e-01,
        5.6499e-01, -7.2338e-01,  1.6736e-01,  4.2030e-01,  9.3788e-01,
       -1.7248e-01,  2.9126e-01, -7.8257e-01, -1.2844e-01,  3.3886e-01,
       -3.3947e-01, -4.2727e-01,  1.4579e+00, -3.1335e-01, -5.2318e-05,
        6.1973e-01, -3.6482e-02, -1.0523e-01, -1.0039e-01,  2.6267e-01,
       -5.3926e-01,  1.8011e-01, -6.1530e-02,  1.9553e-01, -1.0654e+00,
        6.8364e-02,  1.0889e-02,  6.8862e-01, -3.0893e-02, -2.9747e-01,
        2.7474e-01,  3.5519e-01,  6.7799e-02, -4.5287e-01,  5.9265e-01,
       -1.7061e-01, -2.9119e-01, -1.5082e-01, -3.1661e-01, -3.8160e-01,
        2.8878e-01, -9.4547e-02,  1.9872e-01, -4.1434e-02, -4.5629e-03,
        2.0329e-01,  4.4544e-02, -1.0714e-01,  1.3336e-02,  7.6487e-01,
       -9.7920e-01,  4.1987e-01,  1.2322e-01,  2.2844e-01, -3.8980e-01,
        3.5201e-02,  5.8634e-03,  5.3757e-01, -2.7319e-01, -2.0973e-01,
       -5.6800e-01, -1.3261e-01, -4.0366e-02, -4.3473e-01,  5.9351e-01,
       -4.8095e-01,  4.6200e-01,  3.0850e-01,  1.5563e-02, -2.6255e-01,
       -6.2401e-02,  2.3781e-01,  3.2069e-01,  2.8044e-02, -9.3812e-02,
        4.6437e-01, -3.8001e-01, -3.7434e-01, -3.1303e-02, -3.3077e-01,
       -3.9036e-01,  1.2612e+00,  1.1394e+00, -7.4785e-01,  5.8881e-01,
       -2.9830e-02, -3.4661e-01,  5.9226e-01, -1.1268e-01, -4.8890e-01,
       -1.1186e-01, -7.3733e-02, -4.0556e-01, -5.4455e-01, -6.4546e-03,
```

```
 5.3944e-01, -2.0552e-01, -1.9367e-01, -3.6157e-01, -2.4745e-01,
-1.2224e+00,  4.7183e-01, -3.5296e-01,  8.0333e-01,  4.6336e-02,
-4.3909e-01, -6.4112e-01, -5.2724e-01, -1.6876e-01,  2.0895e-01,
 1.4199e-02, -4.4900e-02, -1.8501e-01,  2.6656e-01,  5.3488e-01,
-5.8822e-02,  1.6143e-01, -2.5204e-01, -3.0524e-01, -4.1517e-01,
 1.8948e-01, -1.1039e-01, -2.6072e-02,  2.1508e-01,  4.5536e-01,
-3.6486e-01, -3.8706e-01,  6.5790e-02,  4.3741e-01, -1.2157e-01,
-1.9559e-01,  8.4984e-02, -2.5018e-02, -1.0655e-01, -1.7552e-01,
-2.1178e+00,  6.2321e-01,  3.7638e-01,  1.8257e-01, -1.0744e-01,
-2.0112e-01, -4.1126e-01,  4.4622e-01,  1.7428e-01, -3.6297e-01,
 4.5627e-01,  2.8229e-01,  8.2129e-01,  2.8097e-01, -9.1766e-01,
-6.8294e-01, -4.6742e-02,  9.2274e-02,  1.2806e-01,  5.2903e-01,
 2.1093e-03,  3.3689e-01,  1.6343e-01,  6.8163e-02, -1.1069e-01,
-7.0630e-01, -1.4520e-01, -2.7527e-01,  3.5671e-02, -1.4704e-01,
 5.3811e-01, -1.7595e-01,  3.4800e-01, -2.3449e-01, -7.5448e-01,
 1.4617e-01, -1.0186e-01,  5.3314e-02,  2.8269e-01, -3.3149e-01,
-1.7500e-01, -3.6936e-01,  1.1284e-01, -4.5708e-02,  6.5629e-02,
-3.2347e-01, -2.3809e-02, -2.6306e-01, -7.6422e-01,  2.3938e-02,
-6.1736e-01, -1.4619e-02,  1.8673e-01,  7.3024e-02,  2.8767e-01,
 2.0519e-01,  1.1381e-01,  1.8318e-01, -4.0793e-01, -2.1453e-01,
-2.3576e-01,  1.8909e-01, -3.0893e-01, -2.8578e-01, -4.0883e-01,
-5.6930e-02,  2.6070e-01,  4.7287e-01,  5.0606e-01,  4.1980e-01,
-8.1708e-01, -1.3931e-01,  5.4377e-01,  4.0280e-01, -4.5356e-01,
-3.4729e-01,  1.1177e-02, -5.9710e-01, -3.3182e-01, -1.3462e-01,
-4.2745e-01, -2.3084e-01,  4.7092e-02,  4.3972e-01,  5.6560e-01,
-1.2565e-01,  5.3948e-02, -1.0517e-01, -4.7234e-01,  5.2615e-01,
-3.2875e-01, -4.0684e-01, -1.3978e-01,  7.8114e-02, -6.4585e-01,
 1.3611e-01,  2.7017e-01, -6.7502e-01,  3.1804e-02, -1.3092e-01,
-2.4324e-01, -1.6927e-02, -2.9533e-01, -2.8841e-01,  3.1248e-01,
 2.6822e-02, -8.9684e-02, -3.2709e-01, -4.5021e-01,  6.5495e-01,
 1.6854e-02, -4.5004e-01,  1.2894e-01, -6.1869e-02, -3.3053e-02,
 5.5370e-01, -3.1155e-01, -1.5421e-01,  3.7474e-01, -1.1047e-01,
 2.4952e-01, -7.5216e-01, -2.6494e-01,  2.2681e-01,  1.9609e-01,
 1.0381e-01, -3.6300e-02,  3.2758e-01, -4.8151e-01, -6.3161e-01,
 5.0505e-01,  1.1183e-01, -1.0353e-01,  1.7729e-01,  3.6105e-01,
-5.6136e-02, -1.4535e-01,  2.3805e-01, -2.3554e-02,  9.8490e-01,
-7.6400e-02,  2.5642e-01,  7.0182e-01,  3.2332e-01, -2.0407e-02,
 6.4603e-01,  6.2263e-01,  8.0323e-02, -2.2618e-02,  5.7249e-01,
 6.3276e-03, -5.4889e-01, -2.2304e-01, -3.4033e-01, -4.3941e-01,
 1.5785e-01, -5.2109e-01, -1.1121e+00,  5.0105e-01,  1.5993e-01],
dtype=float32)
```

1.5.19 Representing documents using word embeddings

- Assuming that we have reasonable representations of words.
- How do we represent meaning of **paragraphs** or **documents**?
- Two simple approaches
  - **Averaging** embeddings

– **Concatenating** embeddings

1.5.20 Averaging embeddings

All empty promises

$(embedding(all) + embedding(empty) + embedding(promise))/3$

1.5.21 Average embeddings with spaCy

- We can do this conveniently with spaCy.
- We need `en_core_web_md` model to access word vectors.
- You can download the model by going to command line and in your course `conda` environment and download `en_core_web_md` as follows.

```
conda activate cpsc330
python -m spacy download en_core_web_md
```

We can access word vectors for individual words in `spaCy` as follows.

[25]:
```
nlp("empty").vector[0:10]
```

[25]:
```
array([-0.76058 , -0.21996 ,  0.095465,  0.10395 , -0.52345 ,  0.060248,
        0.069298, -0.61524 , -0.37134 ,  1.5214  ], dtype=float32)
```

We can get **average embeddings** for a sentence or a document in `spaCy` as follows:

[26]:
```
s = "All empty promises"
doc = nlp(s)
avg_sent_emb = doc.vector
print(avg_sent_emb.shape)
print("Vector for: {}\n{}".format((s), (avg_sent_emb[0:10])))
```

```
(300,)
Vector for: All empty promises
[-0.6622033   0.06859333  0.18747167  0.04704666 -0.239323    0.043686
  0.13470568 -0.00746667 -0.03796467  1.9193001 ]
```

1.5.22 Similarity between documents

- We can also get similarity between documents as follows.
- Note that this is **based on average embeddings of each sentence**.

[27]:
```
doc1 = nlp("Deep learning is very popular these days.")
doc2 = nlp("Machine learning is dominated by neural networks.")
doc3 = nlp("A home-made fresh bread with butter and cheese.")

# Similarity of two documents
print(doc1, "<->", doc2, doc1.similarity(doc2))
print(doc2, "<->", doc3, doc2.similarity(doc3))
```

```
Deep learning is very popular these days. <-> Machine learning is dominated by
neural networks. 0.7346255041188937
Machine learning is dominated by neural networks. <-> A home-made fresh bread
with butter and cheese. 0.4759406337841433
```

- Do these scores make sense?
- There are no common words, but we are still able to identify that doc1 and doc2 are more similar that doc2 and doc3.

- You can use such average embedding representation in text classification tasks.

1.5.23 Airline sentiment analysis using average embedding representation

- Let's try average embedding representation for airline sentiment analysis.
- We used this dataset last week so you should already have it in the data directory. If not you can download it here.

[28]:
```
df = pd.read_csv("data/Airline-Sentiment-2-w-AA.csv", encoding="ISO-8859-1")
```

[29]:
```
from sklearn.model_selection import cross_validate, train_test_split

train_df, test_df = train_test_split(df, test_size=0.2, random_state=123)
X_train, y_train = train_df["text"], train_df["airline_sentiment"]
X_test, y_test = test_df["text"], test_df["airline_sentiment"]
```

[30]:
```
train_df.head()
```

[30]:

	_unit_id	_golden	_unit_state	_trusted_judgments	_last_judgment_at	\
5789	681455792	False	finalized	3	2/25/15 4:21	
8918	681459957	False	finalized	3	2/25/15 9:45	
11688	681462990	False	finalized	3	2/25/15 9:53	
413	681448905	False	finalized	3	2/25/15 10:10	
4135	681454122	False	finalized	3	2/25/15 10:08	

	airline_sentiment	airline_sentiment:confidence	negativereason	\
5789	negative	1.0	Can't Tell	
8918	neutral	1.0	NaN	
11688	negative	1.0	Customer Service Issue	
413	neutral	1.0	NaN	
4135	negative	1.0	Bad Flight	

	negativereason:confidence	airline	airline_sentiment_gold	\
5789	0.6667	Southwest	NaN	
8918	NaN	Delta	NaN	
11688	0.6727	US Airways	NaN	
413	NaN	Virgin America	NaN	
4135	0.3544	United	NaN	

```
                name negativereason_gold  retweet_count  \
5789   mrssuperdimmock                NaN              0
8918           labeles                NaN              0
11688   DropMeAnywhere                NaN              0
413          jsamaudio                NaN              0
4135          CajunSQL                NaN              0

                                                  text tweet_coord  \
5789                     @SouthwestAir link doesn't work         NaN
8918   @JetBlue okayyyy. But I had huge irons on way …         NaN
11688  @USAirways They're all reservations numbers an…  [0.0, 0.0]
413              @VirginAmerica no A's channel this year?         NaN
4135   @united missed it.  Incoming on time, then Sat…         NaN

       tweet_created      tweet_id              tweet_location  \
5789   2/19/15 18:53  5.686040e+17          Lake Arrowhead, CA
8918   2/17/15 10:18  5.677500e+17                         NaN
11688  2/17/15 14:50  5.678190e+17  Here, There and Everywhere
413    2/18/15 12:25  5.681440e+17        St. Francis (Calif.)
4135   2/17/15 14:20  5.678110e+17             Baton Rouge, LA

                  user_timezone
5789   Pacific Time (US & Canada)
8918                          NaN
11688                     Arizona
413    Pacific Time (US & Canada)
4135                          NaN
```

1.5.24 Bag-of-words representation for sentiment analysis

[31]:
```
pipe = make_pipeline(
    CountVectorizer(stop_words="english"), LogisticRegression(max_iter=1000)
)
pipe.named_steps["countvectorizer"].fit(X_train)
X_train_transformed = pipe.named_steps["countvectorizer"].transform(X_train)
print("Data matrix shape:", X_train_transformed.shape)
pipe.fit(X_train, y_train);
```

```
Data matrix shape: (11712, 13064)
```

[32]:
```
print("Train accuracy {:.2f}".format(pipe.score(X_train, y_train)))
print("Test accuracy {:.2f}".format(pipe.score(X_test, y_test)))
```

```
Train accuracy 0.94
Test accuracy 0.80
```

1.5.25 Sentiment analysis with average embedding representation

- Let's see how can we get word vectors using spaCy.

- Let's create average embedding representation for each example.

[33]:
```
X_train_embeddings = pd.DataFrame([text.vector for text in nlp.pipe(X_train)])
X_test_embeddings = pd.DataFrame([text.vector for text in nlp.pipe(X_test)])
```

We have **reduced dimensionality** from 13,064 to 300!

[34]:
```
X_train_embeddings.shape
```

[34]:
```
(11712, 300)
```

[35]:
```
X_train_embeddings.head()
```

[35]:
```
         0         1         2         3         4         5         6    \
0 -0.561952  0.225163 -0.293684 -0.082498 -0.192374  0.050405  0.060202
1 -0.700668  0.124276 -0.259541 -0.260308 -0.129364  0.068210  0.066224
2 -0.720738  0.284142 -0.285948 -0.150022 -0.134630 -0.035959  0.124829
3 -0.602323  0.310292 -0.197665 -0.003409 -0.053958  0.033286  0.190612
4 -0.657547  0.228270 -0.184336 -0.162730 -0.076056  0.042302  0.014390

         7         8         9    …       290       291       292       293  \
0 -0.254210  0.089486  1.773900  … -0.210224 -0.033940 -0.130713 -0.145631
1 -0.307202 -0.047832  1.612876  … -0.172155  0.069523 -0.077437 -0.027307
2 -0.282646  0.044680  2.146762  … -0.308978  0.067948 -0.141636 -0.083790
3 -0.242265  0.060668  1.795106  … -0.165880  0.106567 -0.159539  0.017612
4 -0.260486  0.062895  1.678635  … -0.202215  0.068234  0.006729 -0.011597

        294       295       296       297       298       299
0  0.138794  0.115038 -0.081300 -0.155296  0.053767  0.150895
1  0.097390  0.174096  0.023029  0.083394 -0.034370  0.089504
2  0.118324  0.212624 -0.079882 -0.146916  0.038091  0.046573
3  0.111956  0.153079  0.005538 -0.133347  0.075873 -0.150727
4  0.087860  0.185101  0.031942 -0.112022  0.029528  0.123609

[5 rows x 300 columns]
```

1.5.26 Sentiment classification using average embeddings

- What are the train and test accuracies with average word embedding representation?
- The accuracy is a bit better with **less overfitting**.
- Note that we are using **transfer learning** here.
- The embeddings are trained on a completely different corpus.

[36]:
```
lgr = LogisticRegression(max_iter=1000)
lgr.fit(X_train_embeddings, y_train)
print("Train accuracy {:.2f}".format(lgr.score(X_train_embeddings, y_train)))
print("Test accuracy {:.2f}".format(lgr.score(X_test_embeddings, y_test)))
```

```
Train accuracy 0.74
Test accuracy 0.75
```

1.5.27 Sentiment classification using advanced sentence representations

- Since, representing documents is so essential for text classification tasks, there are more advanced methods for document representation.

[37]:
```
from sentence_transformers import SentenceTransformer

embedder = SentenceTransformer("paraphrase-distilroberta-base-v1")
```

[38]:
```
emb_sents = embedder.encode("all empty promises")
emb_sents.shape
```

[38]:
```
(768,)
```

[39]:
```
emb_train = embedder.encode(train_df["text"].tolist())
emb_train_df = pd.DataFrame(emb_train, index=train_df.index)
emb_train_df
```

[39]:
```
            0         1         2         3         4         5         6    \
5789  -0.120494  0.250262 -0.022795 -0.116368  0.078650  0.037357 -0.251341
8918  -0.182954  0.118281  0.066341 -0.136098  0.094947 -0.121302  0.069233
11688 -0.032988  0.630251 -0.079516  0.148981  0.194709 -0.226264 -0.043630
413   -0.119259  0.172168  0.098698  0.319858  0.415475  0.248360 -0.025923
4135   0.094240  0.360193  0.213747  0.363690  0.275521  0.134936 -0.276319
…           …         …         …         …         …         …         …
5218  -0.204409 -0.145290 -0.064201  0.213572 -0.140226  0.338555 -0.148578
12252  0.108407  0.438293  0.216812 -0.349289  0.422689  0.377760  0.045197
1346   0.068411  0.017590  0.236154  0.221446 -0.103567  0.055510  0.062909
11646 -0.091488 -0.155709  0.032391  0.018313  0.524998  0.563933 -0.080984
3582   0.185626  0.092904  0.097085 -0.174650 -0.193584  0.047294  0.098216

            7         8         9    …       758       759       760  \
5789   0.321429 -0.143984 -0.123487  …  0.199150 -0.150143  0.167077
8918  -0.097500  0.025739 -0.367980  …  0.113612  0.114661  0.049926
11688  0.217398 -0.010716  0.069643  …  0.676791  0.244484  0.051042
413    0.385350  0.066414 -0.334289  … -0.128482 -0.232447 -0.077805
4135   0.009336 -0.021523 -0.258992  …  0.474885  0.242125  0.294532
…           …         …         …  …         …         …         …
5218   0.224516 -0.042963  0.075930  … -0.161949  0.040582  0.003971
12252 -0.034096  0.427570 -0.328272  …  0.257849 -0.032362 -0.275004
1346   0.067424 -0.003504 -0.157758  …  0.007711  0.323297  0.334637
11646  0.097983 -0.535285 -0.377194  …  0.428014 -0.144572  0.045296
3582   0.332670  0.163098 -0.135102  …  0.078530 -0.030176  0.391598

            761       762       763       764       765       766       767
```

```
5789  -0.407670 -0.066161  0.049514  0.019384 -0.357601  0.125995  0.381073
8918   0.256736 -0.118687 -0.190721  0.011985 -0.141883 -0.230142  0.024899
11688  0.064099 -0.146945  0.090878 -0.090060  0.077212 -0.209226  0.308773
413    0.181328  0.123244 -0.143693  0.660456 -0.048714  0.204774  0.163497
4135   0.279013  0.037831  0.089761 -0.548748 -0.049257  0.154525  0.141268
…         …       …        …        …        …        …        …
5218  -0.152549 -0.582908 -0.126527  0.060503 -0.111495 -0.097493  0.199321
12252  0.080452 -0.078975 -0.049972 -0.009762 -0.314754 -0.020774  0.268777
1346   0.367041 -0.068821  0.063667 -0.329991  0.232331 -0.184768 -0.000683
11646 -0.107935 -0.135673 -0.290019 -0.137200 -0.503395 -0.042567 -0.282592
3582   0.073519 -0.454038 -0.244358 -0.790682 -0.607009 -0.255162  0.029779

[11712 rows x 768 columns]
```

[40]:
```
emb_test = embedder.encode(test_df["text"].tolist())
emb_test_df = pd.DataFrame(emb_test, index=test_df.index)
emb_test_df
```

[40]:
```
             0         1         2         3         4         5         6    \
1671  -0.002864  0.217325  0.124350 -0.082548  0.709687 -0.582442  0.257897
10951 -0.141048  0.137934  0.131319  0.194773  0.868205  0.078791 -0.131657
5382  -0.252943  0.527507 -0.065608  0.013467  0.207990  0.003881 -0.066282
3954   0.054318  0.096739  0.113037  0.032039  0.493064 -0.641102  0.078760
11193 -0.065858  0.223270  0.507333  0.266193  0.104695 -0.219555  0.146247
…         …       …        …        …        …        …        …
5861   0.077512  0.322276  0.026697 -0.111392  0.174207  0.235202  0.053888
3627  -0.173311 -0.023604  0.190388 -0.136543 -0.360269 -0.444687  0.056311
12559 -0.124635 -0.101799  0.129061  0.636908  0.681090  0.399300 -0.078322
8123   0.063509  0.332506  0.119605 -0.001363 -0.161801 -0.082302 -0.025883
210    0.015537  0.425568  0.350672  0.113120 -0.128615  0.098113  0.222081

             7         8         9    …       758       759       760  \
1671   0.169356  0.248880 -0.266686  …  0.501767  0.095387  0.340172
10951  0.036244 -0.215749 -0.291946  … -0.056255 -0.056040  0.147340
5382   0.253166  0.021039  0.290956  …  0.180686 -0.042605 -0.173794
3954   0.402187  0.189743 -0.089538  …  0.123879 -0.285019 -0.297771
11193  0.315650 -0.126193 -0.435462  …  0.163994  0.207813 -0.001871
…         …       …        …  …       …        …        …
5861   0.244941  0.181625 -0.226870  …  0.149843  0.311337  0.045975
3627   0.291941 -0.399719 -0.167930  …  0.042209 -0.161904 -0.040535
12559  0.221824 -0.277218 -0.178589  …  0.022364 -0.109274 -0.073540
8123   0.048027  0.126974 -0.159801  …  0.002221 -0.093885  0.430284
210    0.101654  0.224073 -0.341075  …  0.100983 -0.008055  0.202025

             761       762       763       764       765       766       767
1671   0.087452 -0.368359  0.276195  0.238676 -0.219546  0.066603  0.256149
10951  0.189665 -0.357366  0.061799 -0.161922 -0.278955 -0.173722  0.065324
```

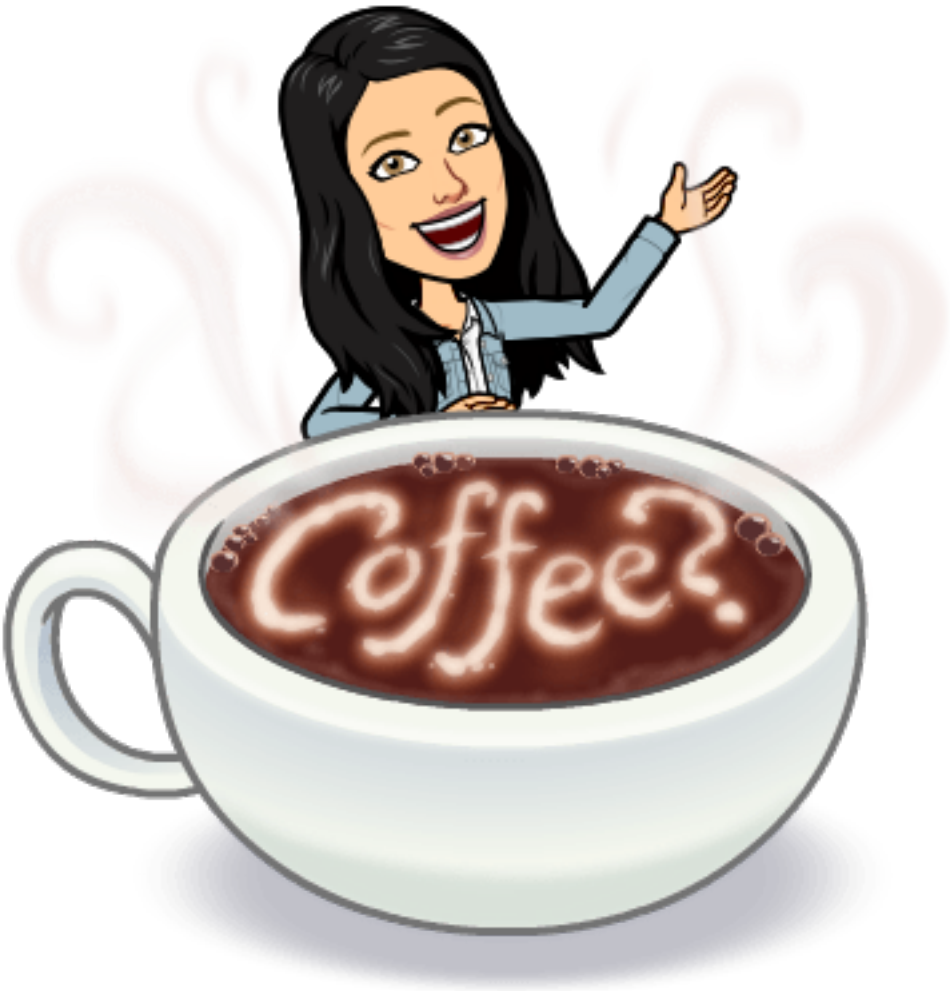
```
5382  -0.079129 -0.169161  0.001317 -0.142593 -0.070816 -0.208826  0.400736
3954   0.557171  0.076169 -0.029826 -0.076094  0.225455  0.002135  0.235430
11193  0.109391 -0.166779 -0.249199 -0.525419 -0.413066  0.119939  0.064297
…           …         …         …         …         …         …         …
5861  -0.572319 -0.068257  0.217745 -0.056510 -0.355174 -0.028610  0.090676
3627  -0.050515 -0.252020 -0.133981  0.155001 -0.154482 -0.060201 -0.126555
12559 -0.153336 -0.123705 -0.238896  0.296447 -0.116797  0.115076 -0.345925
8123  -0.088561  0.321488  0.447437  0.292395 -0.188566 -0.272767  0.126173
210    0.029846 -0.019182  0.107063  0.002301  0.038213 -0.139270 -0.007586

[2928 rows x 768 columns]
```

[41]:
```
lgr = LogisticRegression(max_iter=1000)
lgr.fit(emb_train, y_train)
print("Train accuracy {:.2f}".format(lgr.score(emb_train, y_train)))
print("Test accuracy {:.2f}".format(lgr.score(emb_test, y_test)))
```

```
Train accuracy 0.87
Test accuracy 0.83
```

- Some improvement over bag of words and average embedding representations!
- But much slower …

1.6 Break (5 min)

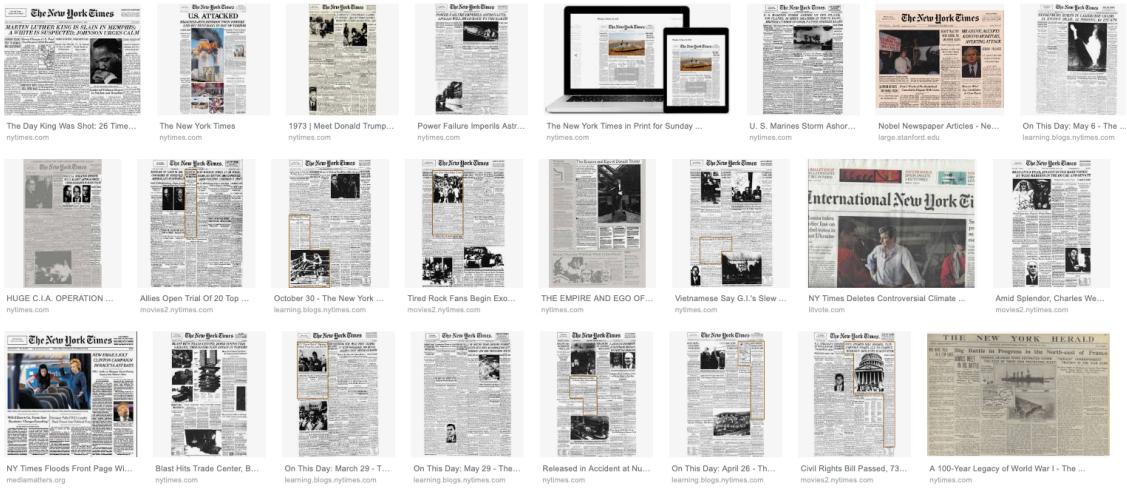


1.7 Topic modeling

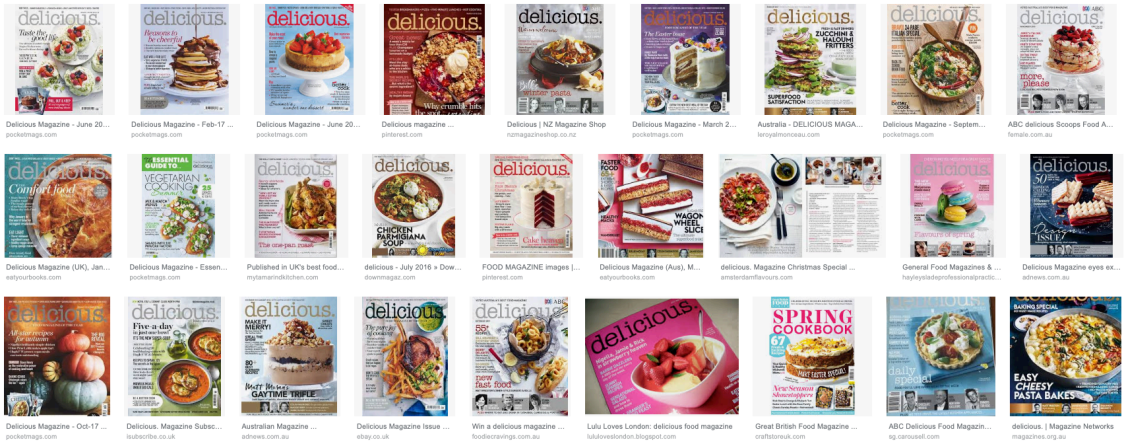
1.7.1 Topic modeling motivation

- Suppose you have a large collection of documents on a variety of topics.

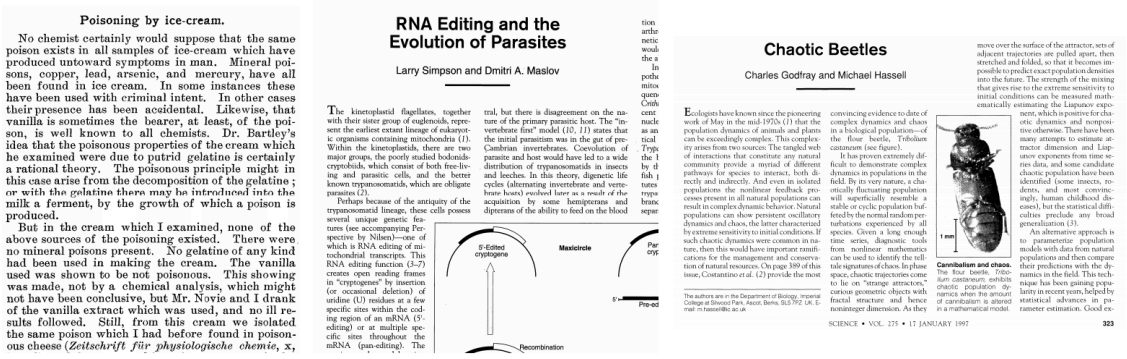
1.7.2 Example: A corpus of news articles



1.7.3 Example: A corpus of food magazines



1.7.4 A corpus of scientific articles



Poisoning by ice-cream.

No chemist certainly would suppose that the same poison exists in all samples of ice-cream which have produced untoward symptoms in man. Mineral poisons, copper, lead, arsenic, and mercury, have all been found in ice cream. In some instances these have been used with criminal intent. In other cases their presence has been accidental. Likewise, that vanilla is sometimes the bearer, at least, of the poison, is well known to all chemists. Dr. Bartley's idea that the poisonous properties of the cream which he examined were due to putrid gelatine is certainly a rational theory. The poisonous principle might in this case arise from the decomposition of the gelatine; or with the gelatine there may be introduced into the milk a ferment, by the growth of which a poison is produced.

But in the cream which I examined, none of the above sources of the poisoning existed. There were no mineral poisons present. No gelatine of any kind had been used in making the cream. The vanilla used was shown to be not poisonous. This showing was made, not by a chemical analysis, which might not have been conclusive, but Mr. Novie and I drank of the vanilla extract which was used, and no ill results followed. Still, from this cream we isolated the same poison which I had before found in poisonous cheese (*Zeitschrift für physiologische chemie*, x,

RNA Editing and the Evolution of Parasites

Larry Simpson and Dmitri A. Maslov

The kinetoplastid flagellates, together with their sister group of euglenoids, represent the earliest extant lineage of eukaryotic organisms containing mitochondria (*1*). Within the kinetoplastids, there are two major groups, the poorly studied bodonids-cryptobiids, which consist of both free-living and parasitic cells, and the better known trypanosomatids, which are obligate parasites (*2*).

Perhaps because of the antiquity of the trypanosomatid lineage, these cells possess several unique genetic features (see accompanying Perspective by Nilsen)—one of which is RNA editing of mitochondrial transcripts. This RNA editing function (*3*–*7*) creates open reading frames in "cryptogenes" by insertion (or occasional deletion) of uridine (U) residues at a few specific sites within the coding region of an mRNA (5′-editing) or at multiple specific sites throughout the mRNA (pan-editing). The

tion
arthr
netic
woul
the a
In
pothe
mitoc
quen
Crith
cent
nucle
as an
tical
Trypa
the l
by t
fish
tutes
trypa
branc
separ

5′-Edited cryptogene

Maxicircle

Pre-ed

Recombination

Chaotic Beetles

Charles Godfray and Michael Hassell

Ecologists have known since the pioneering work of May in the mid-1970s (*1*) that the population dynamics of animals and plants can be exceedingly complex. This complexity arises from two sources: The tangled web of interactions that constitute any natural community provide a myriad of different pathways for species to interact, both directly and indirectly. And even in isolated populations the nonlinear feedback processes present in all natural populations can result in complex dynamic behavior. Natural populations can show persistent oscillatory dynamics and chaos, the latter characterized by extreme sensitivity to initial conditions. If such chaotic dynamics were common in nature, then this would have important ramifications for the management and conservation of natural resources. On page 389 of this issue, Costantino *et al.* (*2*) provide the most convincing evidence to date of complex dynamics and chaos in a biological population—of the flour beetle, *Tribolium castaneum* (see figure).

It has proven extremely difficult to demonstrate complex dynamics in populations in the field. By its very nature, a chaotically fluctuating population will superficially resemble a stable or cyclic population buffeted by the normal random perturbations experienced by all species. Given a long enough time series, diagnostic tools from nonlinear mathematics can be used to identify the telltale signatures of chaos. In phase space, chaotic trajectories come to lie on "strange attractors," curious geometric objects with fractal structure and hence noninteger dimension. As they move over the surface of the attractor, sets of adjacent trajectories are pulled apart, then stretched and folded, so that it becomes impossible to predict exact population densities into the future. The strength of the mixing that gives rise to the extreme sensitivity to initial conditions can be measured mathematically estimating the Liapunov exponent, which is positive for chaotic dynamics and nonpositive otherwise. There have been many attempts to estimate attractor dimension and Liapunov exponents from time series data, and some candidate chaotic population have been identified (some insects, rodents, and most convincingly, human childhood diseases), but the statistical difficulties preclude any broad generalization (*3*).

An alternative approach is to parameterize population models with data from natural populations and then compare their predictions with the dynamics in the field. This technique has been gaining popularity in recent years, helped by statistical advances in parameter estimation. Good ex-

1 mm

Cannibalism and chaos. The flour beetle, *Tribolium castaneum*, exhibits chaotic population dynamics when the amount of cannibalism is altered in a mathematical model.

The authors are in the Department of Biology, Imperial College at Silwood Park, Ascot, Berks. SL5 7PZ, UK. E-mail: m.hassell@ic.ac.uk

SCIENCE • VOL. 275 • 17 JANUARY 1997

323

(Credit: Dave Blei's presentation)

1.7.5 Topic modeling motivation

- Humans are pretty good at reading and understanding a document and answering questions such as
  - What is it about?
  - Which documents is it related to?
- But for a large collection of documents it would take years to read all documents and organize and categorize them so that they are easy to search.
- You need an automated way
  - to get an idea of what's going on in the data or
  - to pull documents related to a certain topic

1.7.6 Topic modeling

- Topic modeling gives you an ability to summarize the major themes in a large collection of documents (corpus).
  - Example: The major themes in a collection of news articles could be
    - **politics**
    - **entertainment**
    - **sports**
    - **technology**
    - …
- A common tool to solve such problems is unsupervised ML methods.
- Given the hyperparameter K, the idea of topic modeling is to describe the data using K "topics"

1.7.7 Topic modeling: Input and output

- Input
  - A large collection of documents
  - A value for the hyperparameter K (e.g., $K = 3$)
- Output
  1. Topic-words association
     - For each topic, what words describe that topic?
  2. Document-topics association
     - For each document, what topics are expressed by the document?

1.7.8 Topic modeling: Example

- Topic-words association
  - For each topic, what words describe that topic?
  - A topic is a mixture of words.

TOPIC 1	TOPIC 2	TOPIC 3
probabilistic, bayesian, markov, stationary, model, hidden, word2vec, pattern, …	fashion, clothes, model, elegant, style, pattern, fresh, designer, creative, …	nutrition, health, butter, soup, bread, baking, apple, fresh, creative, …

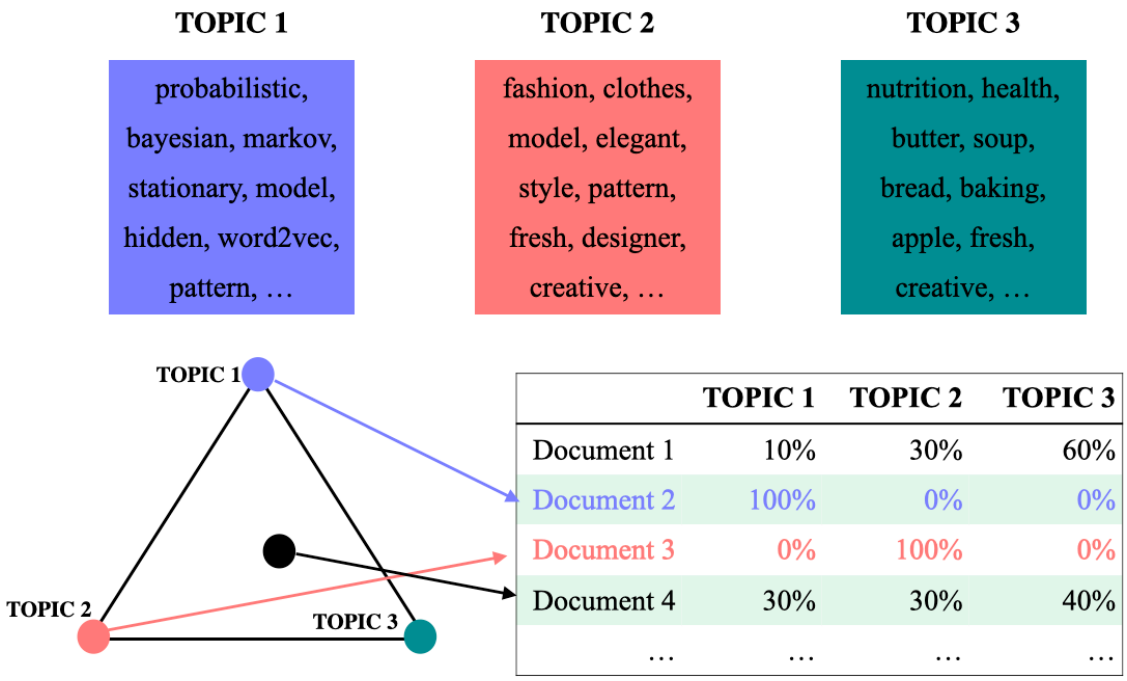
1.7.9 Topic modeling: Example

- Document-topics association
  - For each document, what topics are expressed by the document?
  - A document is a mixture of topics.

	TOPIC 1	TOPIC 2	TOPIC 3
Document 1	10%	30%	60%
Document 2	100%	0%	0%
Document 3	0%	100%	0%
Document 4	30%	30%	40%
…	…	…	…

1.7.10 Topic modeling: Input and output

- Input
  - A large collection of documents
  - A value for the hyperparameter K (e.g., $K = 3$)
- Output
  - For each topic, what words describe that topic?
  - For each document, what topics are expressed by the document?



	TOPIC 1	TOPIC 2	TOPIC 3
Document 1	10%	30%	60%
Document 2	100%	0%	0%
Document 3	0%	100%	0%
Document 4	30%	30%	40%
…	…	…	…

1.7.11 Topic modeling: Some applications

- Topic modeling is a great EDA tool to get a sense of what's going on in a large corpus.
- Some examples
  - If you want to pull documents related to a particular lawsuit.
  - You want to examine people's sentiment towards a particular candidate and/or political party and so you want to pull tweets or Facebook posts related to election.

1.7.12 Topic modeling toy example

[42]:
```
toy_df = pd.read_csv("data/toy_lda_data.csv")
toy_df
```

[42]:

	doc_id	text
0	1	famous fashion model
1	2	fashion model pattern
2	3	fashion model probabilistic topic model confer…
3	4	famous fashion model
4	5	fresh fashion model
5	6	famous fashion model
6	7	famous fashion model
7	8	famous fashion model
8	9	famous fashion model
9	10	creative fashion model
10	11	famous fashion model
11	12	famous fashion model
12	13	fashion model probabilistic topic model confer…
13	14	probabilistic topic model

```
14      15                          probabilistic model pattern
15      16                            probabilistic topic model
16      17                            probabilistic topic model
17      18                            probabilistic topic model
18      19                            probabilistic topic model
19      20                            probabilistic topic model
20      21                            probabilistic topic model
21      22  fashion model probabilistic topic model confer…
22      23                                 apple kiwi nutrition
23      24                                kiwi health nutrition
24      25                                   fresh apple health
25      26                            probabilistic topic model
26      27                            creative health nutrition
27      28                            probabilistic topic model
28      29                            probabilistic topic model
29      30                   hidden markov model probabilistic
30      31                            probabilistic topic model
31      32                            probabilistic topic model
32      33                                 apple kiwi nutrition
33      34                                    apple kiwi health
34      35                                 apple kiwi nutrition
35      36                                    fresh kiwi health
36      37                                 apple kiwi nutrition
37      38                                 apple kiwi nutrition
38      39                                 apple kiwi nutrition
```

[43]:
```python
from gensim import corpora, matutils, models

corpus = [doc.split() for doc in toy_df["text"].tolist()]
# Create a vocabulary for the lda model
dictionary = corpora.Dictionary(corpus)
# Convert our corpus into document-term matrix for Lda
doc_term_matrix = [dictionary.doc2bow(doc) for doc in corpus]
```

[44]:
```python
from gensim.models import LdaModel

# Train an lda model
lda = models.LdaModel(
    corpus=doc_term_matrix,
    id2word=dictionary,
    num_topics=3,
    random_state=123,
    passes=10,
)
```

[45]:
```python
### Examine the topics in our LDA model
lda.print_topics(num_words=4)
```

[45]:
```
[(0,
  '0.303*"model" + 0.296*"probabilistic" + 0.261*"topic" + 0.040*"pattern"'),
 (1, '0.245*"kiwi" + 0.219*"apple" + 0.218*"nutrition" + 0.140*"health"'),
 (2, '0.308*"fashion" + 0.307*"model" + 0.180*"famous" + 0.071*"conference"')]
```

[46]:
```
### Examine the topic distribution for a document
print("Document: ", corpus[0])
df = pd.DataFrame(lda[doc_term_matrix[0]], columns=["topic id", "probability"])
df.sort_values("probability", ascending=False)
```

```
Document:  ['famous', 'fashion', 'model']
```

[46]:

	topic id	probability
2	2	0.828760
0	0	0.087849
1	1	0.083391

You can also visualize the topics using `pyLDAvis`.

```
conda install -n cpsc330 -c anaconda pyLDAvis
```

[47]:
```
# Visualize the topics
import pyLDAvis

pyLDAvis.enable_notebook()
import pyLDAvis.gensim_models as gensimvis

warnings.simplefilter(action="ignore", category=FutureWarning)

vis = gensimvis.prepare(lda, doc_term_matrix, dictionary, sort_topics=False)

warnings.simplefilter(action="default", category=FutureWarning)

vis
```

```
C:\Users\Kenny\miniconda3\envs\cpsc330\lib\site-
packages\past\builtins\misc.py:45: DeprecationWarning: the imp module is
deprecated in favour of importlib and slated for removal in Python 3.12; see the
module's documentation for alternative uses
  from imp import reload
```

[47]:
```
PreparedData(topic_coordinates=              x        y  topics  cluster
Freq
topic
0     -0.187963  0.14888       1        1  40.125872
1      0.312197  0.02174       2        1  27.322216
2     -0.124234 -0.17062       3        1  32.551912, topic_info=
Term       Freq      Total Category  logprob  loglift
1         fashion  13.000000  13.000000  Default  15.0000  15.0000
```

5	probabilistic	15.000000	15.000000	Default	14.0000	14.0000
10	kiwi	9.000000	9.000000	Default	13.0000	13.0000
6	topic	14.000000	14.000000	Default	12.0000	12.0000
9	apple	8.000000	8.000000	Default	11.0000	11.0000
11	nutrition	8.000000	8.000000	Default	10.0000	10.0000
2	model	28.000000	28.000000	Default	9.0000	9.0000
0	famous	8.000000	8.000000	Default	8.0000	8.0000
12	health	5.000000	5.000000	Default	7.0000	7.0000
4	conference	3.000000	3.000000	Default	6.0000	6.0000
7	fresh	3.000000	3.000000	Default	5.0000	5.0000
8	creative	2.000000	2.000000	Default	4.0000	4.0000
3	pattern	2.000000	2.000000	Default	3.0000	3.0000
14	markov	1.000000	1.000000	Default	2.0000	2.0000
13	hidden	1.000000	1.000000	Default	1.0000	1.0000
5	probabilistic	15.092422	15.951447	Topic1	-1.2168	0.8578
6	topic	13.299503	14.180641	Topic1	-1.3433	0.8490
3	pattern	2.026406	2.667343	Topic1	-3.2248	0.6383
14	markov	1.178943	1.781542	Topic1	-3.7664	0.5003
13	hidden	1.178905	1.781544	Topic1	-3.7665	0.5003
2	model	15.434984	28.440566	Topic1	-1.1944	0.3020
8	creative	0.296754	2.698780	Topic1	-5.1459	-1.2945
4	conference	0.338435	3.572499	Topic1	-5.0145	-1.4435
7	fresh	0.297897	3.610627	Topic1	-5.1420	-1.5817
12	health	0.296312	5.447559	Topic1	-5.1474	-1.9984
0	famous	0.297333	8.034160	Topic1	-5.1439	-2.3835
11	nutrition	0.297427	8.178480	Topic1	-5.1436	-2.4009
9	apple	0.296886	8.178506	Topic1	-5.1454	-2.4028
10	kiwi	0.298085	9.088778	Topic1	-5.1414	-2.5043
1	fashion	0.329563	13.387522	Topic1	-5.0410	-2.7912
10	kiwi	8.491084	9.088778	Topic2	-1.4077	1.2294
9	apple	7.582726	8.178506	Topic2	-1.5208	1.2218
11	nutrition	7.581678	8.178480	Topic2	-1.5210	1.2217
12	health	4.852260	5.447559	Topic2	-1.9673	1.1817
7	fresh	2.211254	3.610627	Topic2	-2.7532	0.8071
8	creative	1.223520	2.698780	Topic2	-3.3450	0.5064
13	hidden	0.304395	1.781544	Topic2	-4.7361	-0.4694
14	markov	0.304341	1.781542	Topic2	-4.7363	-0.4696
3	pattern	0.305407	2.667343	Topic2	-4.7328	-0.8697
4	conference	0.304457	3.572499	Topic2	-4.7359	-1.1650
0	famous	0.305362	8.034160	Topic2	-4.7330	-1.9725
1	fashion	0.306387	13.387522	Topic2	-4.7296	-2.4798
6	topic	0.308716	14.180641	Topic2	-4.7221	-2.5297
5	probabilistic	0.307852	15.951447	Topic2	-4.7249	-2.6502
2	model	0.309775	28.440566	Topic2	-4.7186	-3.2223
1	fashion	12.751572	13.387522	Topic3	-1.1762	1.0737
0	famous	7.431464	8.034160	Topic3	-1.7161	1.0444
4	conference	2.929607	3.572499	Topic3	-2.6470	0.9239

```
2           model  12.695808  28.440566   Topic3  -1.1806   0.3158
8        creative   1.178507   2.698780   Topic3  -3.5576   0.2938
7           fresh   1.101477   3.610627   Topic3  -3.6252  -0.0649
14         markov   0.298258   1.781542   Topic3  -4.9316  -0.6649
13         hidden   0.298243   1.781544   Topic3  -4.9317  -0.6650
3         pattern   0.335530   2.667343   Topic3  -4.8139  -0.9508
12         health   0.298987   5.447559   Topic3  -4.9292  -1.7802
6           topic   0.572422  14.180641   Topic3  -4.2797  -2.0874
11      nutrition   0.299375   8.178480   Topic3  -4.9279  -2.1852
9           apple   0.298894   8.178506   Topic3  -4.9295  -2.1868
5   probabilistic   0.551173  15.951447   Topic3  -4.3176  -2.2429
10           kiwi   0.299608   9.088778   Topic3  -4.9271  -2.2900, token_table=
Topic      Freq           Term
term
9         2  0.978174          apple
4         3  0.839748     conference
8         2  0.370538       creative
8         3  0.370538       creative
0         3  0.871280         famous
1         3  0.971053        fashion
7         2  0.553920          fresh
7         3  0.276960          fresh
12        2  0.917842         health
13        1  0.561311         hidden
10        2  0.880206           kiwi
14        1  0.561311         markov
2         1  0.527416          model
2         3  0.457094          model
11        2  0.978177      nutrition
3         1  0.749810        pattern
5         1  0.940354  probabilistic
5         3  0.062690  probabilistic
6         1  0.916743          topic
6         3  0.070519          topic, R=15, lambda_step=0.01, plot_opts={'xlab':
'PC1', 'ylab': 'PC2'}, topic_order=[1, 2, 3])
```

1.8 Topic modeling pipeline

- Preprocess your corpus.
- Train LDA using `Gensim`.
- Interpret your topics.

1.8.1 Data

[48]:
```
import wikipedia

queries = [
    "Artificial Intelligence",
```

```
    "unsupervised learning",
    "Supreme Court of Canada",
    "Peace, Order, and Good Government",
    "Canadian constitutional law",
    "ice hockey",
]
wiki_dict = {"wiki query": [], "text": []}
for i in range(len(queries)):
    wiki_dict["text"].append(wikipedia.page(queries[i]).content)
    wiki_dict["wiki query"].append(queries[i])

wiki_df = pd.DataFrame(wiki_dict)
wiki_df
```

[48]:

	wiki query	text
0	Artificial Intelligence	Artificial intelligence (AI) is intelligence d…
1	unsupervised learning	Supervised learning (SL) is the machine learni…
2	Supreme Court of Canada	The Supreme Court of Canada (SCC; French: Cour…
3	Peace, Order, and Good Government	In many Commonwealth jurisdictions, the phrase…
4	Canadian constitutional law	Canadian constitutional law (French: droit con…
5	ice hockey	Ice hockey (or simply hockey) is a winter team…

1.8.2 Preprocessing the corpus

- **Preprocessing is crucial!**
- Tokenization, converting text to lower case
- Removing punctuation and stopwords
- Discarding words with length $<$ threshold or word frequency $<$ threshold

- Possibly lemmatization: Consider the lemmas instead of inflected forms.
- Depending upon your application, restrict to specific part of speech;
  - For example, only consider nouns, verbs, and adjectives

We'll use spaCy for preprocessing.

[49]:
```
import spacy

warnings.simplefilter(action="ignore", category=DeprecationWarning)

nlp = spacy.load("en_core_web_md", disable=["parser", "ner"])
```

```
warnings.simplefilter(action="default", category=DeprecationWarning)
```

[50]:
```
def preprocess(
    doc,
    min_token_len=2,
    irrelevant_pos=["ADV", "PRON", "CCONJ", "PUNCT", "PART", "DET", "ADP",␣
 ↪"SPACE"],
):
    """
    Given text, min_token_len, and irrelevant_pos carry out preprocessing of␣
 ↪the text
    and return a preprocessed string.

    Parameters
    -------------
    doc : (spaCy doc object)
        the spacy doc object of the text
    min_token_len : (int)
        min_token_length required
    irrelevant_pos : (list)
        a list of irrelevant pos tags

    Returns
    -------------
    (str) the preprocessed text
    """

    clean_text = []

    for token in doc:
        if (
            token.is_stop == False  # Check if it's not a stopword
            and len(token) > min_token_len  # Check if the word meets minimum␣
 ↪threshold
            and token.pos_ not in irrelevant_pos
        ):  # Check if the POS is in the acceptable POS tags
            lemma = token.lemma_  # Take the lemma of the word
            clean_text.append(lemma.lower())
    return " ".join(clean_text)
```

[51]:
```
wiki_df["text_pp"] = [preprocess(text) for text in nlp.pipe(wiki_df["text"])]
```

[52]:
```
wiki_df
```

[52]:
```
                          wiki query  \
0            Artificial Intelligence
```

```
1              unsupervised learning
2            Supreme Court of Canada
3  Peace, Order, and Good Government
4        Canadian constitutional law
5                          ice hockey

                                                 text  \
0  Artificial intelligence (AI) is intelligence d…
1  Supervised learning (SL) is the machine learni…
2  The Supreme Court of Canada (SCC; French: Cour…
3  In many Commonwealth jurisdictions, the phrase…
4  Canadian constitutional law (French: droit con…
5  Ice hockey (or simply hockey) is a winter team…

                                              text_pp
0  artificial intelligence intelligence demonstra…
1  supervised learning machine learn task learn f…
2  supreme court canada scc french cour suprême c…
3  commonwealth jurisdiction phrase peace order g…
4  canadian constitutional law french droit const…
5  ice hockey hockey winter team sport play ice s…
```

1.8.3 Training LDA with gensim

To train an LDA model with gensim, you need

- Document-term matrix
- Dictionary (vocabulary)
- The number of topics (K): `num_topics`
- The number of passes: `passes`

1.8.4 Building a Dictionary

- Let's first create a dictionary using `corpora.Dictionary`.

[53]:
```
corpus = [doc.split() for doc in wiki_df["text_pp"].tolist()]
dictionary = corpora.Dictionary(corpus)  # Create a vocabulary for the lda model
pd.DataFrame(
    dictionary.token2id.keys(), index=dictionary.token2id.values(),␣
 ↪columns=["Word"]
)
```

[53]:
```
      Word
0     0026
1     0030
2     0036
3     0040
4      007
```

```
…          …
4973  zhenskaya
4974
4975
4976
4977

[4978 rows x 1 columns]
```

1.8.5 Gensim's doc2bow

- Now let's convert our corpus into document-term matrix for LDA using `dictionary.doc2bow`.
- For each document, it stores the **frequency of each token** in the document in the format (**token_id**, **frequency**).

[54]:
```
doc_term_matrix = [dictionary.doc2bow(doc) for doc in corpus]
doc_term_matrix[1][:20]
```

[54]:
```
[(292, 4),
 (305, 3),
 (312, 1),
 (318, 3),
 (319, 1),
 (327, 1),
 (329, 3),
 (337, 1),
 (375, 52),
 (385, 1),
 (399, 4),
 (401, 1),
 (402, 1),
 (409, 1),
 (425, 2),
 (427, 4),
 (429, 3),
 (454, 1),
 (465, 1),
 (477, 2)]
```

Now we are ready to train an LDA model.

[55]:
```
from gensim.models import LdaModel

num_topics = 3

lda = models.LdaModel(
    corpus=doc_term_matrix,
    id2word=dictionary,
```

```
    num_topics=num_topics,
    random_state=42,
    passes=10,
)
```

1.8.6 Examine the topics and topic distribution for a document in our LDA model

[56]:
```
lda.print_topics(num_words=4)  # Topics
```

[56]:
```
[(0,
  '0.014*"intelligence" + 0.012*"artificial" + 0.008*"original" +
0.007*"retrieve"'),
 (1, '0.015*"power" + 0.015*"government" + 0.012*"provincial" + 0.011*"law"'),
 (2, '0.024*"hockey" + 0.015*"ice" + 0.012*"team" + 0.012*"player"')]
```

[57]:
```
print("Document: ", wiki_df.iloc[0][0])
print("Topic assignment for document: ", lda[doc_term_matrix[0]])  # Topic␣
 ↪distribution
```

```
Document:  Artificial Intelligence
Topic assignment for document:  [(0, 0.9999209)]
```

1.8.7 Visualize topics

[58]:
```
warnings.simplefilter(action="ignore", category=FutureWarning)

vis = gensimvis.prepare(lda, doc_term_matrix, dictionary, sort_topics=False)

warnings.simplefilter(action="default", category=FutureWarning)

vis
```

[58]:
```
PreparedData(topic_coordinates=              x         y  topics  cluster
Freq
topic
0      0.128437  0.074836       1        1  44.763941
1      0.020105 -0.122908       2        1  10.090208
2     -0.148542  0.048072       3        1  45.145851, topic_info=
Term         Freq       Total Category  logprob  loglift
4462        hockey  234.000000  234.000000  Default  30.0000  30.0000
4477           ice  141.000000  141.000000  Default  29.0000  29.0000
1537  intelligence  135.000000  135.000000  Default  28.0000  28.0000
2164         power   50.000000   50.000000  Default  27.0000  27.0000
1340    government   49.000000   49.000000  Default  26.0000  26.0000
…             …         …         …      …      …        …
2991          year   29.766297   39.516029   Topic3  -5.7736   0.5119
1937      national   32.312503   50.947503   Topic3  -5.6915   0.3399
1330          goal   34.066275   57.925049   Topic3  -5.6387   0.2644
```

```
2981         world   32.420175   53.748741   Topic3  -5.6882   0.2897
1682           law   28.660223   72.578454   Topic3  -5.8115  -0.1339

[188 rows x 6 columns], token_table=      Topic      Freq   Term
term
128       1  0.978942   2007
139       1  0.972191   2015
143       1  0.885248   2016
143       2  0.026826   2016
143       3  0.080477   2016
…     …        …     …
3628      3  0.979329  woman
2981      1  0.390707  world
2981      3  0.595363  world
2991      1  0.227756   year
2991      3  0.759186   year

[260 rows x 3 columns], R=30, lambda_step=0.01, plot_opts={'xlab': 'PC1',
'ylab': 'PC2'}, topic_order=[1, 2, 3])
```

1.8.8 (Optional) Topic modeling with sklearn

- We are using `Gensim` LDA so that we'll be able to use `CoherenceModel` to evaluate topic model later.
- Bit we can also train an LDA model with `sklearn`.

[59]:
```
from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer()
X = vec.fit_transform(wiki_df["text_pp"])
```

[60]:
```
from sklearn.decomposition import LatentDirichletAllocation

n_topics = 3
lda = LatentDirichletAllocation(
    n_components=n_topics, learning_method="batch", max_iter=10, random_state=0
)
document_topics = lda.fit_transform(X)
```

[61]:
```
print("lda.components_.shape: {}".format(lda.components_.shape))
```

```
lda.components_.shape: (3, 4954)
```

[62]:
```
sorting = np.argsort(lda.components_, axis=1)[:, ::-1]
feature_names = np.array(vec.get_feature_names_out())
```

[63]:
```
import mglearn
```

```
warnings.simplefilter(action="ignore", category=DeprecationWarning)

mglearn.tools.print_topics(
    topics=range(3),
    feature_names=feature_names,
    sorting=sorting,
    topics_per_chunk=5,
    n_words=10,
)

warnings.simplefilter(action="default", category=DeprecationWarning)
```

```
topic 0       topic 1       topic 2
--------      --------      --------
hockey        intelligence  court
ice           artificial    law
team          learning      provincial
player        algorithm     federal
league        original      government
play          retrieve      power
game          machine       canada
penalty       archive       justice
puck          human         supreme
nhl           displaystyle  case
```

```
C:\Users\Kenny\miniconda3\envs\cpsc330\lib\site-packages\mglearn\plot_pca.py:7:
DeprecationWarning: The 'cachedir' parameter has been deprecated in version 0.12
and will be removed in version 0.14.
You provided "cachedir='cache'", use "location='cache'" instead.
  memory = Memory(cachedir="cache")
C:\Users\Kenny\miniconda3\envs\cpsc330\lib\site-packages\mglearn\plot_nmf.py:7:
DeprecationWarning: The 'cachedir' parameter has been deprecated in version 0.12
and will be removed in version 0.14.
You provided "cachedir='cache'", use "location='cache'" instead.
  memory = Memory(cachedir="cache")
```

1.9 Basic text preprocessing

1.9.1 Introduction

- Why do we need preprocessing?
  - Text data is unstructured and messy.
  - We need to "normalize" it before we do anything interesting with it.
- Example:
  - **Lemma**: Same stem, same part-of-speech, roughly the same meaning
    - Vancouver's → Vancouver
    - computers → computer

    - rising → rise, rose, rises

1.9.2 Tokenization

- Sentence segmentation
  - Split text into sentences
- Word tokenization
  - Split sentences into words

1.9.3 Tokenization: sentence segmentation

MDS is a Master's program at UBC in British Columbia. MDS teaching team is truly multicultural!! Dr. George did his Ph.D. in Scotland. Dr. Timbers, Dr. Ostblom, Dr. Rodríguez-Arelis, and Dr. Kolhatkar did theirs in Canada. Dr. Gelbart did his PhD in the U.S.

- How many sentences are there in this text?

[64]:
```
### Let's do sentence segmentation on "."
text = (
    "MDS is a Master's program at UBC in British Columbia. "
    "MDS teaching team is truly multicultural!! "
    "Dr. George did his Ph.D. in Scotland. "
    "Dr. Timbers, Dr. Ostblom, Dr. Rodríguez-Arelis, and Dr. Kolhatkar did␣
 ↪theirs in Canada. "
    "Dr. Gelbart did his PhD in the U.S."
)

print(text.split("."))
```

```
["MDS is a Master's program at UBC in British Columbia", ' MDS teaching team is
truly multicultural!! Dr', ' George did his Ph', 'D', ' in Scotland', ' Dr', '
Timbers, Dr', ' Ostblom, Dr', ' Rodríguez-Arelis, and Dr', ' Kolhatkar did
theirs in Canada', ' Dr', ' Gelbart did his PhD in the U', 'S', '']
```

1.9.4 Sentence segmentation

- In English, period (.) is quite ambiguous. (In Chinese, it is unambiguous.)
  - Abbreviations like Dr., U.S., Inc.
  - Numbers like 60.44%, 0.98
- ! and ? are relatively ambiguous.
- How about writing regular expressions?
- A common way is using off-the-shelf models for sentence segmentation.

[65]:
```
### Let's try to do sentence segmentation using nltk
from nltk.tokenize import sent_tokenize

sent_tokenized = sent_tokenize(text)
print(sent_tokenized)
```

```
["MDS is a Master's program at UBC in British Columbia.", 'MDS teaching team is
truly multicultural!!', 'Dr. George did his Ph.D. in Scotland.', 'Dr. Timbers,
Dr. Ostblom, Dr. Rodríguez-Arelis, and Dr. Kolhatkar did theirs in Canada.',
'Dr. Gelbart did his PhD in the U.S.']
```

1.9.5 Word tokenization

MDS is a Master's program at UBC in British Columbia.

- How many words are there in this sentence?
- Is whitespace a sufficient condition for a word boundary?

1.9.6 Word tokenization

MDS is a Master's program at UBC in British Columbia.

- What's our definition of a word?
  - Should British Columbia be one word or two words?
  - Should punctuation be considered a separate word?
  - What about the punctuations in `U.S.`?
  - What do we do with words like `Master's`?
- This process of identifying word boundaries is referred to as **tokenization**.
- You can use regex but better to do it with off-the-shelf ML models.

[66]:
```
### Let's do word segmentation on white spaces
print("Splitting on whitespace: ", [sent.split() for sent in sent_tokenized])

### Let's try to do word segmentation using nltk
from nltk.tokenize import word_tokenize

word_tokenized = [word_tokenize(sent) for sent in sent_tokenized]
# This is similar to the input format of word2vec algorithm
print("\n\n\nTokenized: ", word_tokenized)
```

```
Splitting on whitespace:  [['MDS', 'is', 'a', "Master's", 'program', 'at',
'UBC', 'in', 'British', 'Columbia.'], ['MDS', 'teaching', 'team', 'is', 'truly',
'multicultural!!'], ['Dr.', 'George', 'did', 'his', 'Ph.D.', 'in', 'Scotland.'],
['Dr.', 'Timbers,', 'Dr.', 'Ostblom,', 'Dr.', 'Rodríguez-Arelis,', 'and', 'Dr.',
'Kolhatkar', 'did', 'theirs', 'in', 'Canada.'], ['Dr.', 'Gelbart', 'did', 'his',
'PhD', 'in', 'the', 'U.S.']]
```

```
Tokenized:  [['MDS', 'is', 'a', 'Master', "'s", 'program', 'at', 'UBC', 'in',
'British', 'Columbia', '.'], ['MDS', 'teaching', 'team', 'is', 'truly',
'multicultural', '!', '!'], ['Dr.', 'George', 'did', 'his', 'Ph.D.', 'in',
'Scotland', '.'], ['Dr.', 'Timbers', ',', 'Dr.', 'Ostblom', ',', 'Dr.',
'Rodríguez-Arelis', ',', 'and', 'Dr.', 'Kolhatkar', 'did', 'theirs', 'in',
```

```
'Canada', '.'], ['Dr.', 'Gelbart', 'did', 'his', 'PhD', 'in', 'the', 'U.S',
'.']]
```

1.9.7 Word segmentation

For some languages you need much more sophisticated tokenizers. - For languages such as Chinese, there are no spaces between words. - jieba is a popular tokenizer for Chinese. - German doesn't separate compound words. * Example: *rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz* * (the law for the delegation of monitoring beef labeling)

1.9.8 Types and tokens

- Usually in NLP, we talk about
  - **Type** an element in the vocabulary
  - **Token** an instance of that type in running text

1.9.9 Exercise for you

UBC is located in the beautiful province of British Columbia. It's very close to the U.S. border. You'll get to the USA border in about 45 mins by car.

- Consider the example above.
  - How many types? (task dependent)
  - How many tokens?

1.9.10 Other commonly used preprocessing steps

- Punctuation and stopword removal
- Stemming and lemmatization

1.9.11 Punctuation and stopword removal

- The most frequently occurring words in English are not very useful in many NLP tasks.
  - Example: *the* , *is* , *a* , and punctuation
- Probably not very informative in many tasks

[67]:
```
# Let's use `nltk.stopwords`.
# Add punctuations to the list.
stop_words = list(set(stopwords.words("english")))
import string

punctuation = string.punctuation
stop_words += list(punctuation)
# stop_words.extend(['``','`','br','"',""", "''", "'s"])
print(stop_words)
```

```
['will', "wouldn't", 'are', 'down', 'she', 'does', 'during', 'few', 'into', 't',
'you', 'them', 'hadn', 'have', 'the', 'for', 'didn', 'up', 'shouldn', 'through',
'theirs', 'own', 'and', 'it', 've', 'other', 'where', 'is', 'on', 'll',
"haven't", 'itself', 'd', 'him', "she's", "mightn't", 'weren', 'as', 'doing',
```

```
'they', "shan't", 'if', 'by', 'just', 'needn', 'your', "should've", 'further',
'myself', 'once', "hadn't", "mustn't", 'too', 'between', "you've", 'wasn',
'against', 'these', 'm', 'over', 'hasn', 'any', 'here', "aren't", 'be', 'aren',
'ourselves', 'out', 'above', "couldn't", 'had', "wasn't", 'himself', 'now',
"shouldn't", "don't", 'after', 'shan', 'those', 'why', "you're", 'until',
'under', 'same', "didn't", 'isn', 'very', 'what', 'an', 'both', 'herself',
'from', "needn't", 'before', 'should', 'but', 'ain', 'more', 'has', 'their',
'all', 'each', 'to', 'nor', 'his', 'were', 'been', 'a', 'this', 'wouldn',
'doesn', 'below', "it's", 'couldn', 'so', 'ma', 'because', 'having', "hasn't",
"you'll", 'yourself', 'with', 'did', 'mustn', 'while', 'haven', 'who', 's',
'themselves', 'her', 'mightn', 'some', 'can', "that'll", 'won', 'am', 'no',
'which', 'of', 'whom', 'its', 'there', 'at', 'such', 'then', 'o', 'only', 're',
'don', 'being', 'ours', 'our', 'yourselves', 'that', 'again', 'not', 'y',
"isn't", 'yours', 'when', 'or', 'how', "won't", 'than', "you'd", "weren't",
'hers', 'about', 'me', 'in', "doesn't", 'i', 'he', 'my', 'we', 'most', 'off',
'do', 'was', '!', '"', '#', '$', '%', '&', "'", '(', ')', '*', '+', ',', '-',
'.', '/', ':', ';', '<', '=', '>', '?', '@', '[', '\\', ']', '^', '_', '`', '{',
'|', '}', '~']
```

[68]:
```
### Get rid of stop words
preprocessed = []
for sent in word_tokenized:
    for token in sent:
        token = token.lower()
        if token not in stop_words:
            preprocessed.append(token)
print(preprocessed)
```

```
['mds', 'master', "'s", 'program', 'ubc', 'british', 'columbia', 'mds',
'teaching', 'team', 'truly', 'multicultural', 'dr.', 'george', 'ph.d.',
'scotland', 'dr.', 'timbers', 'dr.', 'ostblom', 'dr.', 'rodríguez-arelis',
'dr.', 'kolhatkar', 'canada', 'dr.', 'gelbart', 'phd', 'u.s']
```

1.9.12 Lemmatization

- For many NLP tasks (e.g., web search) we want to ignore morphological differences between words
  - Example: If your search term is "studying for ML quiz" you might want to include pages containing "tips to study for an ML quiz" or "here is how I studied for my ML quiz"
- Lemmatization converts inflected forms into the base form.

[71]:
```
import nltk

nltk.download("wordnet")  # You need to run this at least once
nltk.download('omw-1.4')  # You need to run this at least once
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\Kenny\AppData\Roaming\nltk_data…
[nltk_data]   Unzipping corpora\wordnet.zip.
```

```
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     C:\Users\Kenny\AppData\Roaming\nltk_data…
[nltk_data]   Unzipping corpora\omw-1.4.zip.
```

[71]:
```
True
```

[70]:
```
# nltk has a lemmatizer
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
print("Lemma of studying: ", lemmatizer.lemmatize("studying", "v"))
print("Lemma of studied: ", lemmatizer.lemmatize("studied", "v"))
```

```
---------------------------------------------------------------------------
LookupError                               Traceback (most recent call last)
File ~\miniconda3\envs\cpsc330\lib\site-packages\nltk\corpus\util.py:84, in␣
 ↪LazyCorpusLoader.__load(self)
     83 try:
---> 84     root = nltk.data.find(f"{self.subdir}/{zip_name}")
     85 except LookupError:

File ~\miniconda3\envs\cpsc330\lib\site-packages\nltk\data.py:583, in␣
 ↪find(resource_name, paths)
    582 resource_not_found = f"\n{sep}\n{msg}\n{sep}\n"
--> 583 raise LookupError(resource_not_found)

LookupError:
**********************************************************************
  Resource wordnet not found.
  Please use the NLTK Downloader to obtain the resource:

  >>> import nltk
  >>> nltk.download('wordnet')

  For more information see: https://www.nltk.org/data.html

  Attempted to load corpora/wordnet.zip/wordnet/

  Searched in:
    - 'C:\\Users\\Kenny/nltk_data'
    - 'C:\\Users\\Kenny\\miniconda3\\envs\\cpsc330\\nltk_data'
    - 'C:\\Users\\Kenny\\miniconda3\\envs\\cpsc330\\share\\nltk_data'
    - 'C:\\Users\\Kenny\\miniconda3\\envs\\cpsc330\\lib\\nltk_data'
    - 'C:\\Users\\Kenny\\AppData\\Roaming\\nltk_data'
    - 'C:\\nltk_data'
    - 'D:\\nltk_data'
    - 'E:\\nltk_data'
```

```
**********************************************************************


During handling of the above exception, another exception occurred:

LookupError                               Traceback (most recent call last)
Input In [70], in <cell line: 5>()
      2 from nltk.stem import WordNetLemmatizer
      4 lemmatizer = WordNetLemmatizer()
----> 5 print("Lemma of studying: ", lemmatizer.lemmatize("studying", "v"))
      6 print("Lemma of studied: ", lemmatizer.lemmatize("studied", "v"))

File ~\miniconda3\envs\cpsc330\lib\site-packages\nltk\stem\wordnet.py:45, in␣
 ↪WordNetLemmatizer.lemmatize(self, word, pos)
     33 def lemmatize(self, word: str, pos: str = "n") -> str:
     34     """Lemmatize `word` using WordNet's built-in morphy function.
     35     Returns the input word unchanged if it cannot be found in WordNet.
     36
   (…)
     43     :return: The lemma of `word`, for the given `pos`.
     44     """
---> 45     lemmas = wn._morphy(word, pos)
     46     return min(lemmas, key=len) if lemmas else word

File ~\miniconda3\envs\cpsc330\lib\site-packages\nltk\corpus\util.py:121, in␣
 ↪LazyCorpusLoader.__getattr__(self, attr)
    118 if attr == "__bases__":
    119     raise AttributeError("LazyCorpusLoader object has no attribute␣
 ↪'__bases__'")
--> 121 self.__load()
    122 # This looks circular, but its not, since __load() changes our
    123 # __class__ to something new:
    124 return getattr(self, attr)

File ~\miniconda3\envs\cpsc330\lib\site-packages\nltk\corpus\util.py:86, in␣
 ↪LazyCorpusLoader.__load(self)
     84             root = nltk.data.find(f"{self.subdir}/{zip_name}")
     85         except LookupError:
---> 86             raise e
     88 # Load the corpus.
     89 corpus = self.__reader_cls(root, *self.__args, **self.__kwargs)

File ~\miniconda3\envs\cpsc330\lib\site-packages\nltk\corpus\util.py:81, in␣
 ↪LazyCorpusLoader.__load(self)
     79 else:
     80     try:
---> 81         root = nltk.data.find(f"{self.subdir}/{self.__name}")
     82     except LookupError as e:
```

```
     83         try:

File ~\miniconda3\envs\cpsc330\lib\site-packages\nltk\data.py:583, in␣
 ↪find(resource_name, paths)
    581 sep = "*" * 70
    582 resource_not_found = f"\n{sep}\n{msg}\n{sep}\n"
--> 583 raise LookupError(resource_not_found)

LookupError:
**********************************************************************
  Resource wordnet not found.
  Please use the NLTK Downloader to obtain the resource:

  >>> import nltk
  >>> nltk.download('wordnet')

  For more information see: https://www.nltk.org/data.html

  Attempted to load corpora/wordnet

  Searched in:
    - 'C:\\Users\\Kenny/nltk_data'
    - 'C:\\Users\\Kenny\\miniconda3\\envs\\cpsc330\\nltk_data'
    - 'C:\\Users\\Kenny\\miniconda3\\envs\\cpsc330\\share\\nltk_data'
    - 'C:\\Users\\Kenny\\miniconda3\\envs\\cpsc330\\lib\\nltk_data'
    - 'C:\\Users\\Kenny\\AppData\\Roaming\\nltk_data'
    - 'C:\\nltk_data'
    - 'D:\\nltk_data'
    - 'E:\\nltk_data'
**********************************************************************
```

1.9.13 Stemming

- Has a similar purpose but it is a crude chopping of affixes
  - *automates, automatic, automation* all reduced to *automat.*
- Usually these reduced forms (stems) are not actual words themselves.

- A popular stemming algorithm for English is PorterStemmer.
- Beware that it can be aggressive sometimes.

[ ]:
```
from nltk.stem.porter import PorterStemmer

text = (
    "UBC is located in the beautiful province of British Columbia... "
    "It's very close to the U.S. border."
```

```
)
ps = PorterStemmer()
tokenized = word_tokenize(text)
stemmed = [ps.stem(token) for token in tokenized]
print("Before stemming: ", text)
print("\n\nAfter stemming: ", " ".join(stemmed))
```

1.9.14 Other tools for preprocessing

- We used Natural Language Processing Toolkit (nltk) above
- Many available tools

- spaCy

1.9.15 spaCy

- Industrial strength NLP library.
- Lightweight, fast, and convenient to use.
- spaCy does many things that we did above in one line of code!
- Also has multi-lingual support.

[ ]:
```
import spacy


warnings.simplefilter(action="ignore", category=DeprecationWarning)

# Load the model
nlp = spacy.load("en_core_web_md")

warnings.simplefilter(action="default", category=DeprecationWarning)

text = (
    "MDS is a Master's program at UBC in British Columbia. "
    "MDS teaching team is truly multicultural!! "
    "Dr. George did his Ph.D. in Scotland. "
    "Dr. Timbers, Dr. Ostblom, Dr. Rodríguez-Arelis, and Dr. Kolhatkar did␣
 ↪theirs in Canada. "
    "Dr. Gelbart did his PhD in the U.S."
)

doc = nlp(text)
```

[ ]:
```
# Accessing tokens
tokens = [token for token in doc]
print("\nTokens: ", tokens)

# Accessing lemma
lemmas = [token.lemma_ for token in doc]
```

```
print("\nLemmas: ", lemmas)

# Accessing pos
pos = [token.pos_ for token in doc]
print("\nPOS: ", pos)
```

1.9.16 Other typical NLP tasks

In order to understand text, we usually are interested in extracting information from text. Some common tasks in NLP pipeline are: - Part of speech tagging - Assigning part-of-speech tags to all words in a sentence. - Named entity recognition - Labelling named "real-world" objects, like persons, companies or locations.
- Coreference resolution - Deciding whether two strings (e.g., UBC vs University of British Columbia) refer to the same entity - Dependency parsing - Representing grammatical structure of a sentence

1.9.17 Extracting named-entities using spaCy

[ ]:
```
from spacy import displacy


warnings.simplefilter(action="ignore", category=DeprecationWarning)

doc = nlp(
    "University of British Columbia "
    "is located in the beautiful "
    "province of British Columbia."
)
displacy.render(doc, style="ent")

warnings.simplefilter(action="default", category=DeprecationWarning)

# Text and label of named entity span
print("Named entities:\n", [(ent.text, ent.label_) for ent in doc.ents])
print("\nORG means: ", spacy.explain("ORG"))
print("GPE means: ", spacy.explain("GPE"))
```

1.9.18 Dependency parsing using spaCy

[ ]:
```
warnings.simplefilter(action="ignore", category=DeprecationWarning)

doc = nlp("I like cats")
displacy.render(doc, style="dep")

warnings.simplefilter(action="default", category=DeprecationWarning)
```

1.9.19 Many other things possible

- A powerful tool
- All my Capstone groups last year used this tool.
- You can build your own rule-based searches.
- You can also access word vectors using spaCy with bigger models. (Currently we are using `en_core_web_md` model.)

1.10 Bug workaround!

[72]:
```
# "Do you have pyLDAvis installed in your virtual
#  environment? Apparently there is a strong evidence
#  that it is an interaction with pyLDAvis."
# https://stackoverflow.com/a/69171847/14881731

from IPython.display import HTML
css_str = '<style> \
.jp-Button path { fill: #616161;} \
text.terms { fill: #616161;} \
.jp-icon-warn0 path {fill: var(--jp-warn-color0);} \
.bp3-button-text path { fill: var(--jp-inverse-layout-color3);} \
.jp-icon-brand0 path { fill: var(--jp-brand-color0);} \
text.terms { fill: #616161;} \
</style>'
display(HTML(css_str ))
```

```
<IPython.core.display.HTML object>
```

[ ]:
```

```