

Word Semantics

From theory to practice

Outline

- Introduction to word semantics
- Semantic relations
- WordNet
- Word Embedding
- Using Word Embedding in Python

Lemmas and senses

lemma

mouse (N)

1. any of numerous small rodents...

2. a hand-operated device that controls a cursor...

sense

Modified from the online thesaurus WordNet

A **sense** or “**concept**” is the meaning component of a word
Lemmas can be **polysemous** (have multiple senses)

Relations between senses:

Synonymy

Synonyms have the same meaning in some or all contexts.

- filbert / hazelnut
- couch / sofa
- big / large
- automobile / car
- vomit / throw up
- water / H₂O

Relations between senses:

Synonymy

Note that there are probably no examples of perfect synonymy.

- Even if many aspects of meaning are identical
- Still may differ based on politeness, slang, register, genre, etc.

Relation: Synonymy?

water/H₂O

"H₂O" in a surfing guide?

big/large

my big sister != my large sister

The Linguistic Principle of Contrast

Difference in form →
difference in meaning

Relation: Similarity

Words with similar meanings. Not synonyms, but sharing some element of meaning

car, bicycle

cow, horse

Ask humans how similar two words are

word1	word2	similarity
vanish	disappear	9.8
behave	obey	7.3
belief	impression	5.95
muscle	bone	3.65
modest	flexible	0.98
hole	agreement	0.3

SimLex-999 dataset (Hill et al., 2015)

Relation: Word relatedness

Also called "word association"

Words can be related in any way,
perhaps via a semantic frame or field

- coffee, tea: **similar**
- coffee, cup: **related**, not similar

Semantic field

Words that

- cover a particular semantic domain
- bear structured relations with each other.

hospitals

surgeon, scalpel, nurse, anaesthetic, hospital

restaurants

waiter, menu, plate, food, menu, chef

houses

door, roof, kitchen, family, bed

Relation: Antonymy

Senses that are opposites with respect to only one feature of meaning

Otherwise, they are very similar!

dark/light short/long fast/slow rise/
fall

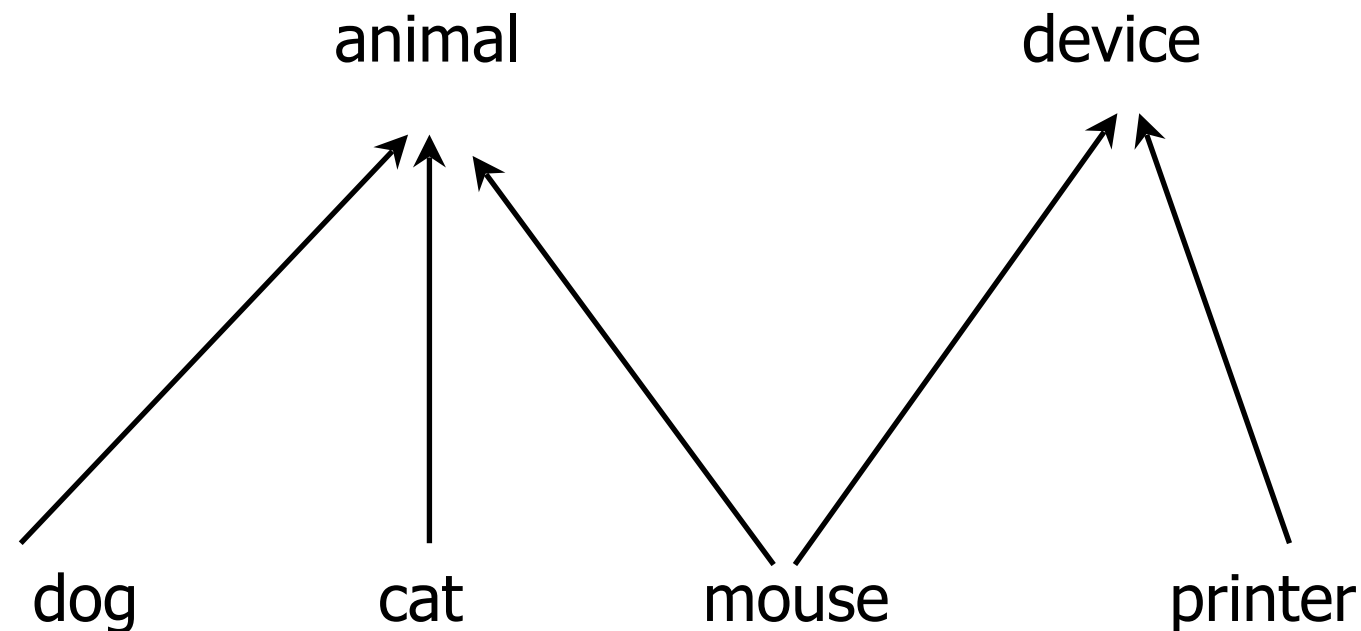
hot/cold up/down in/out

More formally: antonyms can

- define a binary opposition or be at opposite ends of a scale
 - long/short, fast/slow
- Be *reversives*:
 - rise/fall, up/down

Relation: Hyponymy

- An hyponym is a word whose meaning contains the entire meaning of another, known as the superordinate.



Meronymy/Holonymy

- A word **w1** is a meronym of another word **w2** (the holonym) if the relation **is-part-of** holds between the meaning of **w1** and **w2**.
 - Meronymy is transitive and asymmetric
 - A meronym can have many holonyms
- Meronyms are distinguishing features that hyponyms can inherit.
 - " Ex. If "beak" and "wing" are meronyms of "bird", and if "canary" is a hyponym of "bird", then (by inheritance), "beak" and "wing" must be meronyms of "canary".
- Limited transitivity:
 - " Ex. "A house has a door" and "a door has a handle", then "a house has a handle" (?)

Connotation (sentiment)

- Words have **affective** meanings
 - Positive connotations (*happy*)
 - Negative connotations (*sad*)
- Connotations can be subtle:
 - Positive connotation: *copy, replica, reproduction*
 - Negative connotation: *fake, knockoff, forgery*
- Evaluation (sentiment!)
 - Positive evaluation (*great, love*)
 - Negative evaluation (*terrible, hate*)

So far

Concepts or word senses

- Have a complex many-to-many association with **words** (homonymy, multiple senses)

Have relations with each other

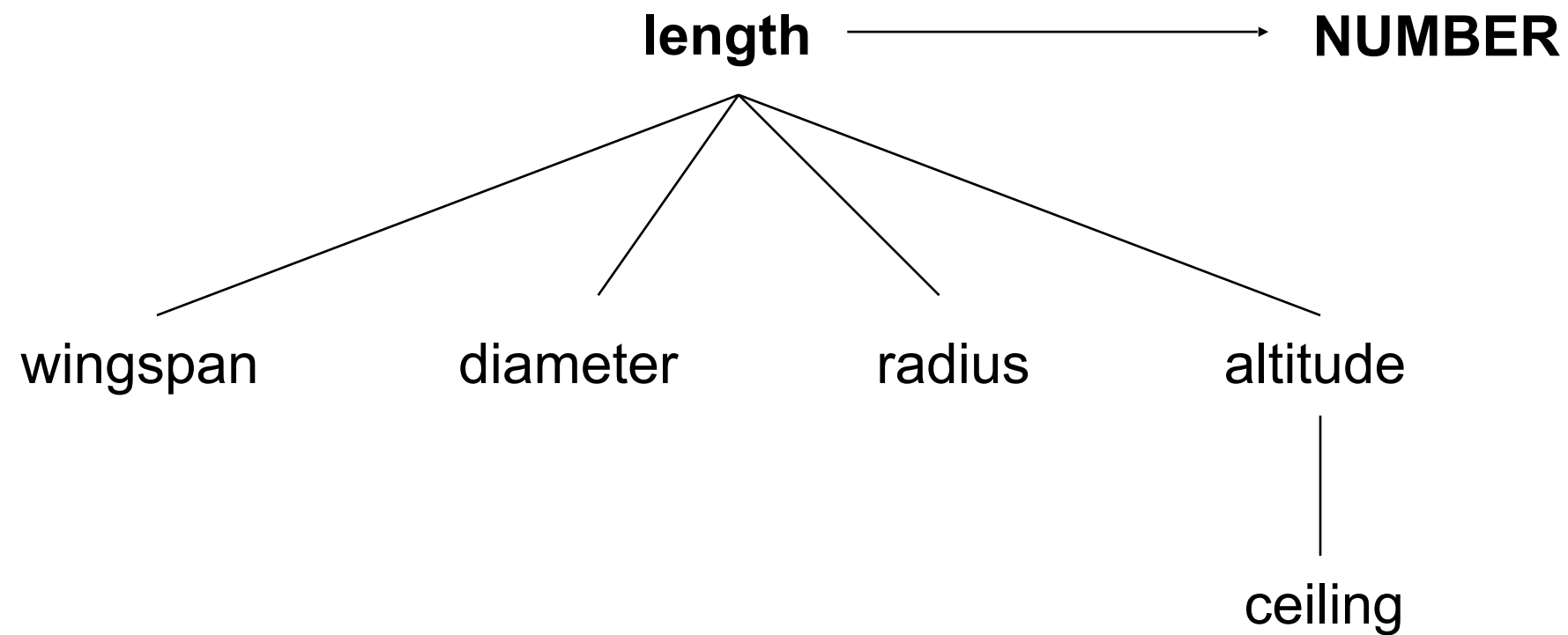
- Synonymy
- Antonymy
- Similarity
- Relatedness
- Connotation

Word Relationship analysis with WordNet

WordNet

- <https://wordnet.princeton.edu>
- A dictionary based on psycholinguistic principles
- A lexical database based on conceptual lookup
 - Organizing concepts in a semantic network.
- Organize lexical information in terms of word meaning, rather than word form
 - Wordnet can also be used as a thesaurus.
- Interfaces for:
 - Java, Prolog, Lisp, Python, Perl, C#
- Recently available also for languages different from English (including Italian)

Using WordNet



Words in WordNet

- A Word is a conventional association between:
 - A lexicalised concept, and
 - A word form that plays a syntactic role.
 - A practical way of organizing lexicalised concepts that words can express.
- Lexical Matrix: a contingency matrix between:
 - Word forms (the columns)
 - Word meanings (the rows).
 - An entry in the matrix indicates that the form in that column can be used to express (in the appropriate context) the meaning in that row.

Lexical Matrix

Word Meanings	Word Forms				
M1	F1	F2	F3	...	F _n
M2	E1,1	E1,2			
M3		E2,2			
...			E3,3	...	
M _m					E _{m,n}

Synonymy

Polysemy

Concepts in WordNet

Hypothesis:

- A synonym is often sufficient to identify the concept.

Differential approach

- Word meaning can be represented by a list of the word forms that can be used to express it: the **synset**.

WordNet in Python

Available in NLTK

```
from nltk.corpus import wordnet  
syn=wordnet.synsets("Run")  
print(syn)
```

Result

```
[Synset('book.n.01'), Synset('book.n.02'),  
Synset('record.n.05'), Synset('script.n.01'),  
Synset('ledger.n.01'), Synset('book.n.06'),  
Synset('book.n.07'), Synset('koran.n.01'),  
Synset('bible.n.01'), Synset('book.n.10'),  
Synset('book.n.11'), Synset('book.v.01'),  
Synset('reserve.v.04'), Synset('book.v.03'),  
Synset('book.v.04')]
```

- The word “book” has multiple synsets
- Each one is composed of:
 - A name (book itself, or a synonym)
 - A part of speech tag
 - A numeric id if the same name has multiple meanings

Analyzing a synset

```
synset = wordnet.synsets("Travel")
print('Word and Type : ' + synset[0].name())
print('Part-of-speech:' + synset[0].pos())
print('Synonym of Travel is: ' + synset[0].lemmas()[0].name())
print('The meaning of the word : ' + synset[0].definition())
print('Example of Travel : ' + str(synset[0].examples()))
print('Hypernyms:' + str(synset[0].hypernyms()))
print('Hyponyms:' + str(synset[0].hyponyms()))
```

Result

Word and Type : travel.n.01

Part-of-speech:n

Synonym of Travel is: travel

The meaning of the word : the act of going from one place to another

Example of Travel : ['he enjoyed selling but he hated the travel']

Hypernyms:[Synset('motion.n.06')]

Hyponyms:[Synset('air_travel.n.01'),
Synset('circumnavigation.n.01'), Synset('commutation.n.01'),
Synset('crossing.n.01'), Synset('driving.n.02'),
Synset('journey.n.01'), Synset('junketing.n.01'),
Synset('on_the_road.n.01'), Synset('peregrination.n.01'),
Synset('riding.n.02'), Synset('stage.n.06'),
Synset('staging.n.03'), Synset('traversal.n.02'),
Synset('walk.n.04'), Synset('wandering.n.01'),
Synset('water_travel.n.01'), Synset('wayfaring.n.01')]

Getting all synonyms and antonyms

```
syn = list()
ant = list()
for synset in wordnet.synsets("Travel"):
    for lemma in synset.lemmas():
        syn.append(lemma.name())      #add the synonyms
        if lemma.antonyms():         #When antonyms are available, add them into the list
            ant.append(lemma.antonyms()[0].name())
print('Synonyms: ' + str(syn))
print('Antonyms: ' + str(ant))
```

Result

Synonyms: ['travel', 'traveling',
'travelling', 'change_of_location',
'travel', 'locomotion', 'travel', 'travel',
'go', 'move', 'locomote', 'travel',
'journey', 'travel', 'trip', 'jaunt',
'travel', 'journey', 'travel', 'travel',
'move_around']

Antonyms: ['stay_in_place']

Synset Distance

```
w1 = wordnet.synset( 'ship.n.01' )  
w2 = wordnet.synset( 'boat.n.01' )  
print(w1.wup_similarity(w2))
```

Result is 0.9090909090909091

Note: you need to get synsets before computing the distance

Word Embeddings

Computational models of word meaning

Can we build a theory of how to represent word meaning, that accounts for at least some of the desiderata?

We'll introduce **vector semantics**

The standard model in language processing!

Handles many of our goals!

Idea 1: Defining meaning by linguistic distribution

Let's define the meaning of a word by its distribution in language use, meaning its neighboring words or grammatical environments.

Idea 2: Meaning as a point in space (Osgood et al. 1957)

3 affective dimensions for a word

- **valence**: pleasantness
- **arousal**: intensity of emotion
- **dominance**: the degree of control exerted

NRC VAD Lexicon
(Mohammad 2018)

	Word	Score	Word	Score
Valence	love	1.000	toxic	0.008
	happy	1.000	nightmare	0.005
Arousal	elated	0.960	mellow	0.069
	frenzy	0.965	napping	0.046
Dominance	powerful	0.991	weak	0.045
	leadershi	0.983	empty	0.081

Hence the connotation of a word is a vector in 3-space

Idea 1: Defining meaning by linguistic distribution

Idea 2: Meaning as a point in multidimensional space

Defining meaning as a point in space based on distribution

Each word = a vector (not just "good" or " w_{45} ")

Similar words are "**nearby in semantic space**"

We build this space automatically by seeing which words are **nearby in text**



We define meaning of a word as a vector

Called an "embedding" because it's
embedded into a space (see textbook)

The standard way to represent meaning in
NLP

**Every modern NLP algorithm uses embeddings
as the representation of word meaning**

Fine-grained model of meaning for similarity

We'll discuss 2 kinds of embeddings

tf-idf (we've seen it already)

- Information Retrieval workhorse!
- A common baseline model
- **Sparse** vectors
- Words are represented by (a simple function of) the **counts** of nearby words

Word2vec

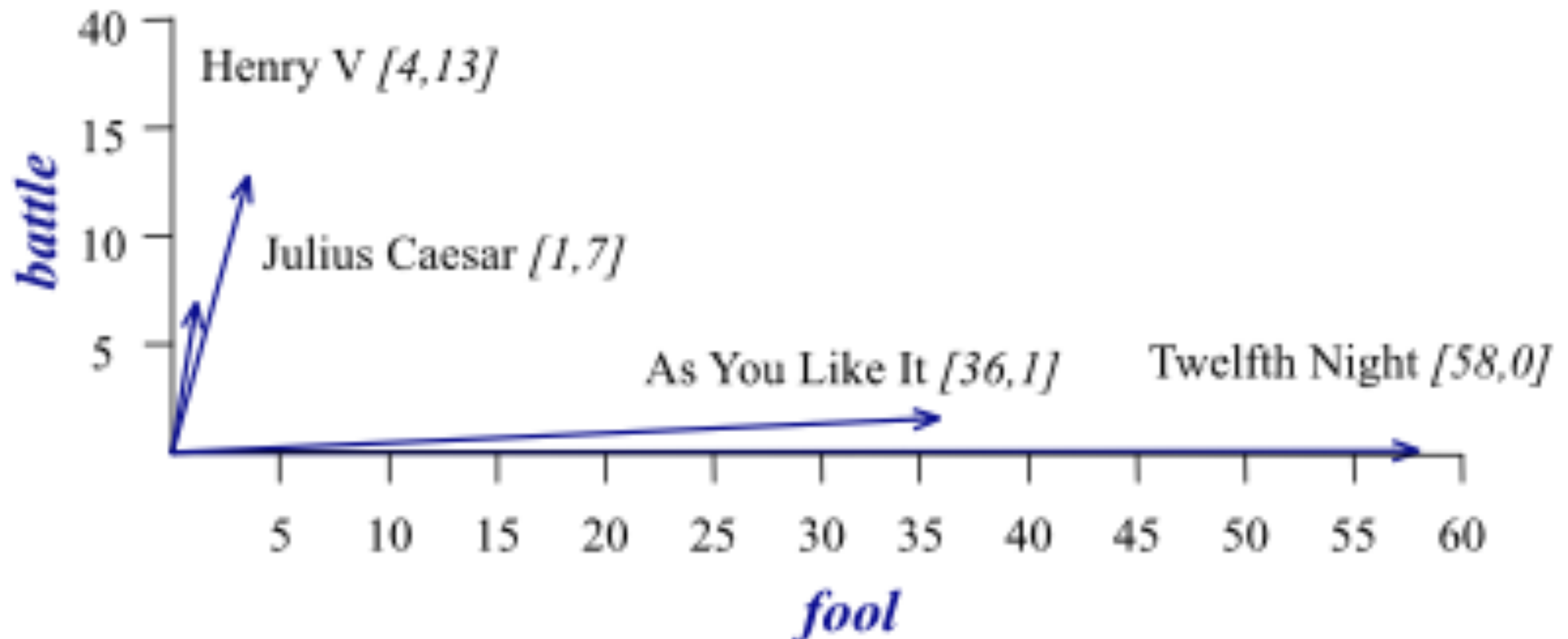
- **Dense** vectors
- Representation is created by training a classifier to **predict** whether a word is likely to appear nearby
- Later we'll discuss extensions called **contextual embeddings**

Term-document matrix

Each document is represented by a vector of words

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Visualizing document vectors



Idea for word meaning: Words can be vectors too!!!

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

battle is "the kind of word that occurs in Julius Caesar and Henry V"

fool is "the kind of word that occurs in comedies, especially Twelfth Night"

More common: word-word matrix (or "term-context matrix")

Two **words** are similar in meaning if their context vectors are similar

is traditionally followed by **cherry** pie, a traditional dessert
often mixed, such as **strawberry** rhubarb pie. Apple pie
computer peripherals and personal **digital** assistants. These devices usually
a computer. This includes **information** available on the internet

	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...

Vector Semantics & Embeddings

Sparse versus dense vectors

tf-idf (or PMI) vectors are

- **long** (length $|V| = 20,000$ to $50,000$)
- **sparse** (most elements are zero)

Alternative: learn vectors which are

- **short** (length 50-1000)
- **dense** (most elements are non-zero)

Sparse versus dense vectors

Why dense vectors?

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

Common methods for getting short dense vectors

“Neural Language Model”-inspired models

- Word2vec (skipgram, CBOW), GloVe

Singular Value Decomposition (SVD)

- A special case of this is LSA – Latent Semantic Analysis (which we have seen already)

Alternative to these "static embeddings":

- Contextual Embeddings (ELMo, BERT)
- Compute distinct embeddings for a word in its context
- Separate embeddings for each token of a word

Simple static embeddings you can download!

Those have been pre-trained on existing large datasets (e.g., Google news, Tweets, Wikipedia)

Word2vec (Mikolov et al)

<https://code.google.com/archive/p/word2vec/>

GloVe (Pennington, Socher, Manning)

<http://nlp.stanford.edu/projects/glove/>

FastText (Facebook Research)

<https://fasttext.cc>

<https://research.fb.com/downloads/fasttext/>

Word2vec

Popular embedding method

Very fast to train

Code available on the web

Idea: **predict** rather than **count**

Word2vec provides various options. We'll do:

skip-gram with negative sampling (SGNS)

Word2vec

Instead of **counting** how often each word w occurs near "*apricot*"

- Train a classifier on a binary **prediction** task:
 - Is w likely to show up near "*apricot*"?

We don't actually care about this task

- But we'll take the learned classifier weights as the word embeddings

Big idea: **self-supervision**:

- A word c that occurs near *apricot* in the corpus acts as the gold "correct answer" for supervised learning
- No need for human labels
- Bengio et al. (2003); Collobert et al. (2011)

Approach: predict if candidate word c is a "neighbor"

1. Treat the target word t and a neighboring context word c as **positive examples**.
2. Randomly sample other words in the lexicon to get negative examples
3. Use logistic regression to train a classifier to distinguish those two cases
4. Use the learned weights as the embeddings

Skip-Gram Training Data

Assume a ± 2 word window, given training sentence:

...lemon, a [tablespoon of apricot jam, a] pinch...

c1 c2 c3 c4

Skip-Gram Classifier

(assuming a +/- 2 word window)

...lemon, a [tablespoon of apricot jam, a] pinch...

c1 c2 [target] c3 c4

Goal: train a classifier that is given a candidate (word, context) pair

(apricot, jam)

(apricot, aardvark)

...

And assigns each pair a probability:

$$P(+|w, c)$$

$$P(-|w, c) = 1 - P(+|w, c)$$

Similarity is computed from dot product

Remember: two vectors are similar if they have a high dot product

- Cosine is just a normalized dot product

So:

- $\text{Similarity}(w, c) \propto w \cdot c$

We'll need to normalize to get a probability

- (cosine isn't a probability either)

Skip-gram classifier: summary

A probabilistic classifier, given

- a test target word w
- its context window of L words $c_{1:L}$


Estimates probability that w occurs in this window based on similarity of w (embeddings) to $c_{1:L}$ (embeddings).

To compute this, we just need embeddings for all the words.

Skip-Gram Training data

...lemon, a [tablespoon of apricot jam, a] pinch...

c1 c2 [target] c3 c4



Positive Examples

t	c
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

Skip-Gram Training data

...lemon, a [tablespoon of apricot jam, a] pinch...

c1 c2 [target] c3 c4



Positive Examples


t	c
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

For each positive example we'll grab k negative examples, sampling by frequency

Skip-Gram Training data

...lemon, a [tablespoon of apricot jam, a] pinch...

c1 c2 [target] c3 c4



Positive Examples

t	c
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

Negative Examples

t	c	t	c
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

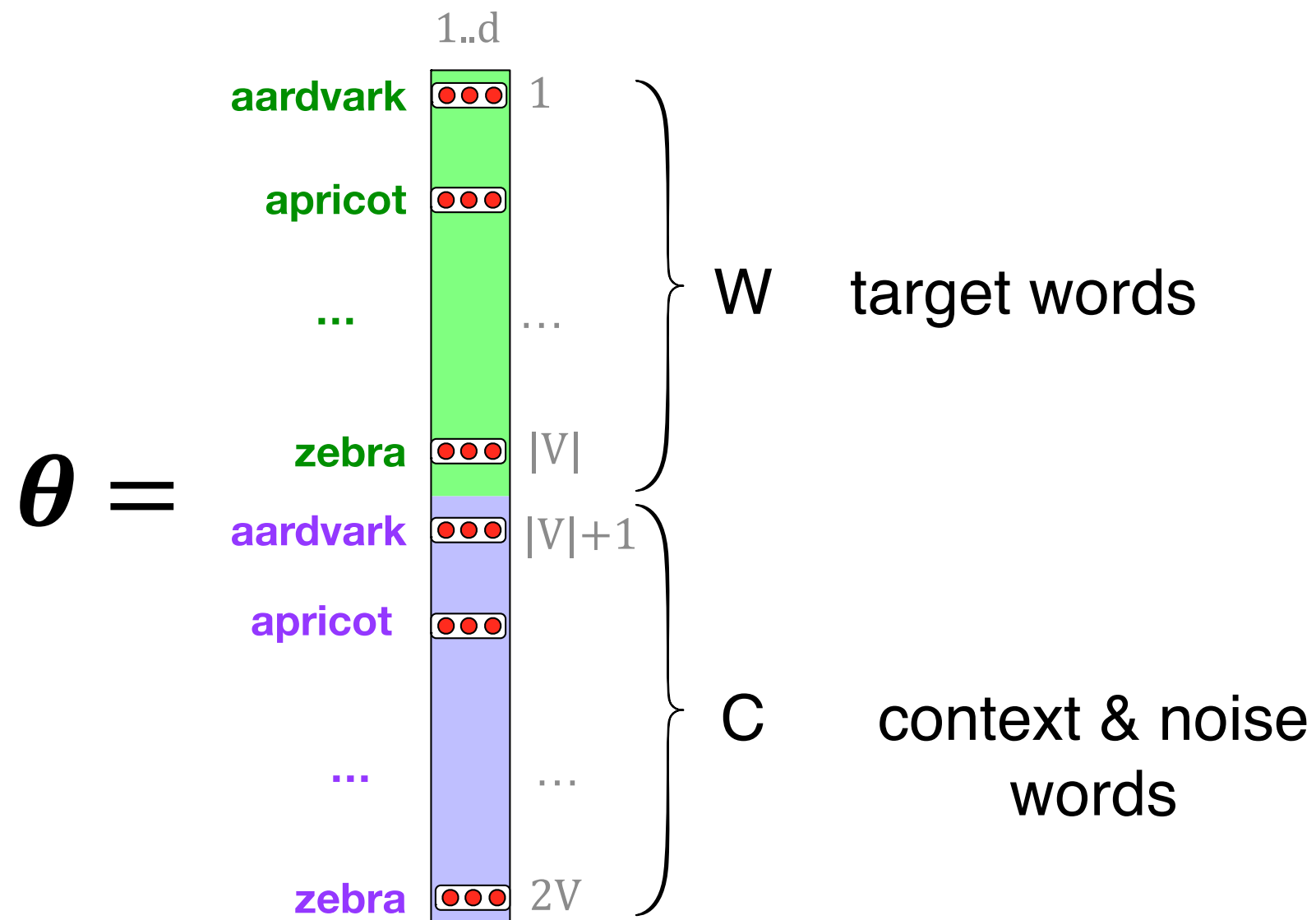
Word2vec: how to learn vectors

Given the set of positive and negative training instances, and an initial set of embedding vectors

The goal of learning is to adjust those word vectors such that we:

- **Maximize** the similarity of the **target word, context word** pairs (w, c_{pos}) drawn from the positive data
- **Minimize** the similarity of the (w, c_{neg}) pairs drawn from the negative data.

These embeddings we'll need: a set for w , a set for c



Summary: How to learn word2vec (skip-gram) embeddings

Start with V random d -dimensional vectors as initial embeddings

Train a classifier based on embedding similarity

- Take a corpus and take pairs of words that co-occur as positive examples
- Take pairs of words that don't co-occur as negative examples
- Train the classifier to distinguish these by slowly adjusting all the embeddings to improve the classifier performance
- Throw away the classifier code and keep the embeddings.

The kinds of neighbors depend on window size

Small windows ($C = \pm 2$) : nearest words are syntactically similar words in same taxonomy

- *Hogwarts* nearest neighbors are other fictional schools
- *Sunnydale, Evernight, Blandings*

Large windows ($C = \pm 5$) : nearest words are related words in same semantic field

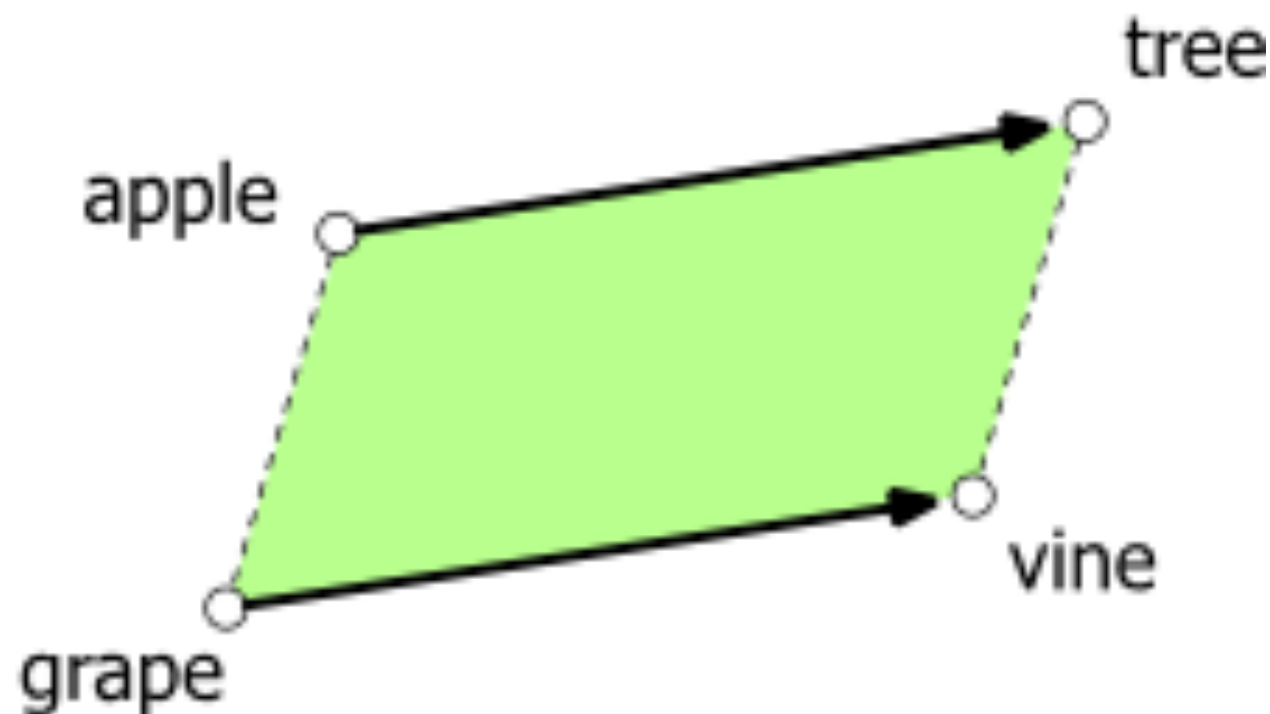
- *Hogwarts* nearest neighbors are Harry Potter world:
- *Dumbledore, half-blood, Malfoy*

Analogical relations

The classic parallelogram model of analogical reasoning
(Rumelhart and Abrahamson 1973)

To solve: "*apple is to tree as grape is to _____*"

Add $\overrightarrow{\text{tree} - \text{apple}}$ to $\overrightarrow{\text{grape}}$ to get *vine*



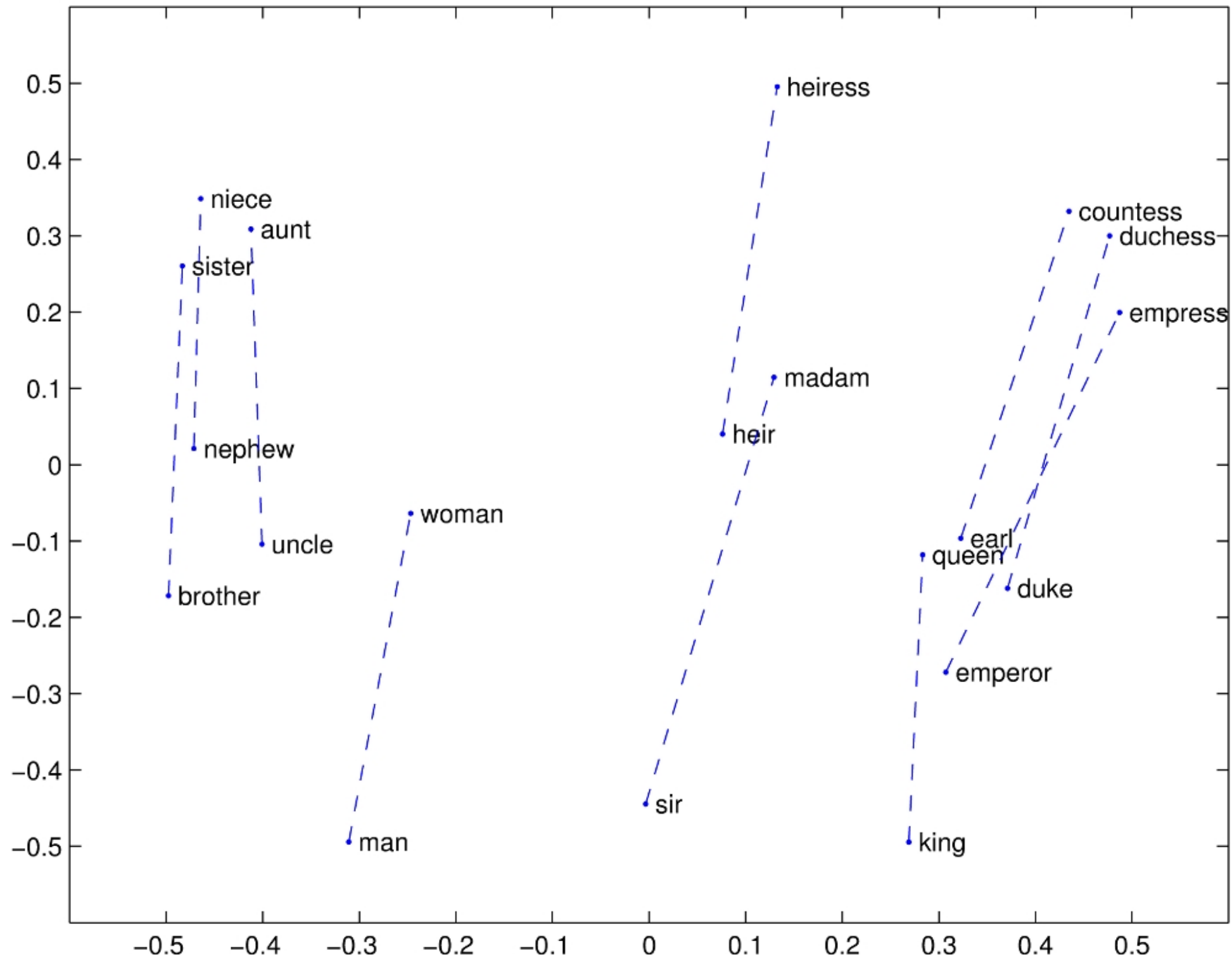
Analogical relations via parallelogram

The parallelogram method can solve analogies with both sparse and dense embeddings (Turney and Littman 2005, Mikolov et al. 2013b)

$\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ is close to $\overrightarrow{\text{queen}}$

$\overrightarrow{\text{Paris}} - \overrightarrow{\text{France}} + \overrightarrow{\text{Italy}}$ is close to $\overrightarrow{\text{Rome}}$

Structure in GloVe Embedding Space



Embeddings reflect cultural bias!

Ask “Paris : France :: Tokyo : x”

x = Japan

Ask “father : doctor :: mother : x”

x = nurse

Ask “man : computer programmer :: woman : x”

x = homemaker

Algorithms that use embeddings as part of e.g., hiring searches for programmers, might lead to bias in hiring

Word Embedding in Python

Word Embeddings in Python

Available in Gensim

- You can train them, or you can load pretrained embeddings
- Can be used to infer relations between words
- Can be used to train machine learning classifiers

Keras/TensorFlow, or pretrained models in HuggingFace

- To train neural networks
- Different models (word2vec, GloVE, but also alternative embeddings such as BERT)

Gensim example

```
from gensim.models import Word2Vec
# define training data
sentences = [['this', 'is', 'the', 'first', 'sentence', 'for', 'word2vec'],
              ['this', 'is', 'the', 'second', 'sentence'],
              ['yet', 'another', 'sentence'],
              ['one', 'more', 'sentence'],
              ['and', 'the', 'final', 'sentence']]

# train model
model = Word2Vec(sentences, min_count=1, vector_size=50)

word="another"
print("another:", model.wv[word])
print("Model length:", len(model.wv))

print("Most similar words:", model.wv.most_similar(positive=word, topn=3))

# save model
model.save('model.bin')

# load model
new_model = Word2Vec.load('model.bin')
print(new_model)
```

Results

```
another: [-0.01427803  0.00248206 -0.01435343 -0.00448924  0.00743861
0.01166625
 0.00239636  0.00420546 -0.00822078  0.01445066 -0.01261408  0.00929443
-0.01643995  0.00407293 -0.0099541  -0.00849538 -0.00621797  0.01131042
 0.0115968  -0.0099493  0.00154666 -0.01699156  0.01561961  0.01851458
-0.00548466  0.00160045  0.0014933  0.01095577 -0.01721216  0.00116891
 0.01373884  0.00446319  0.00224935 -0.01864431  0.01696473 -0.01252826
-0.00598475  0.00698757 -0.00154526  0.00282258  0.00356398 -0.0136578
-0.01944963  0.01808117  0.01239611 -0.01382586  0.00680696  0.00041213
 0.00950749 -0.01423989]
```

Model length: 14

Most similar words: [('second', 0.2373521625995636), ('one', 0.1845843493938446), ('is', 0.13940517604351044)]

Word2Vec(vocab=14, vector_size=50, alpha=0.025)

Discussion - I

- The `Word2Vec` constructor creates a word embedding model from the sentences
- By using `model.wv[word]` we can access the vector of a word
- `model.wv.most_similar(positive=word,topn=3)` returns the closest words if “another” is considered as a positive example
- Once you have created a model, you can store it with the `save` method, and then load it again with `Word2Vec.load`

Discussion - II

- That set of sentences was clearly not enough to train a model
- The training requires a careful calibration of the Word2vec parameters (e.g., the window size, the vector size, etc.)
- In the following, we will mainly use pre-trained models

Using pretrained models

```
import gensim.downloader
print(list(gensim.downloader.info()['models'].keys()))
```

- Result:
['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']

Using pretrained models - discussion

- We have models from different training data, and with vectors having different lengths
- A longer vector normally gives more accurate results, but also means a more expensive computation

Let's use one...

```
import gensim.downloader
glove_vectors = gensim.downloader.load('glove-wiki-gigaword-50')

print(len(glove_vectors[0]))

print(glove_vectors['twitter'])

print(glove_vectors.most_similar('twitter'))

print(glove_vectors.most_similar(positive=['doctor', 'woman'], negative='man'))
```

Discussion

- `gensim.downloader.load` downloads a pretrained model
- `glove_vectors['twitter']` lets you access to the vector for the word 'twitter' (note the structure is slightly different from Word2Vec where you should access `model.wv` instead)
- `most_similar` returns the most similar words. Positive words contribute positively towards the similarity, negative words negatively.

Result

- 50

- | | | | | | |
|-----------|-----------|-----------|-----------|----------|----------|
| 0.55473 | 0.14251 | 1.577 | 0.44222 | -0.40965 | -0.24373 |
| -1.2366 | -0.64589 | 0.31804 | 0.48623 | -0.20947 | 0.019861 |
| -0.28046 | -0.64705 | 0.87607 | -0.28965 | -1.1877 | -0.22703 |
| 0.73132 | 0.064986 | 0.34437 | -0.044798 | 0.85787 | 1.0463 |
| 1.3781 | -0.21831 | 0.45545 | -0.36639 | -0.32279 | -0.34018 |
| 1.5663 | -0.028824 | 0.0062708 | -0.62084 | -1.3351 | 0.082663 |
| -0.085856 | -0.67657 | -1.1872 | -0.40016 | 1.1583 | -0.50842 |
| -1.8528 | 0.49679 | 0.94368 | -0.97676 | 0.30505 | 0.15514 |
| 0.26331 | -0.10485 | | | | |

- [('facebook', 0.9333045482635498), ('myspace', 0.8801369667053223), ('youtube', 0.8430657982826233), ('blog', 0.8262057304382324), ('blogs', 0.8064824342727661), ('blogging', 0.7970671057701111), ('tumblr', 0.7901089787483215), ('email', 0.778261125087738), ('tweets', 0.7604537010192871), ('e-mail', 0.7538726925849915)]

- [('nurse', 0.8404642939567566), ('child', 0.7663259506225586), ('pregnant', 0.7570130228996277), ('mother', 0.7517457604408264), ('patient', 0.7516663074493408), ('physician', 0.7507280707359314), ('dentist', 0.7360343933105469), ('therapist', 0.7342537045478821), ('parents', 0.7286345958709717), ('surgeon', 0.7165213227272034)]

**You now see what AI
bias means!!!!**

Using Word Embedding in Machine Learning

Problems

- Up to now, we represented documents as vectors
- However, now a word is a vector, hence a document would be a matrix
- How we should proceed?

Approach 1: Embedding Tensor

- Where appropriate, each word (in the ordering it appears in the document) is represented as a vector
- We no longer treat documents as bag of independent words (ordering is taken into account)
- No need for stemming, you may or may not want to perform stop word removal (experiment!)
- Hence, each document is a $m \times n$ matrix where
 - m is the embedding size
 - n is the document length
- Since documents may have varying lengths, we consider n as the maximum

Embedding tensor

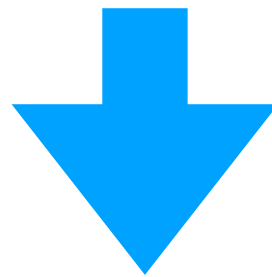
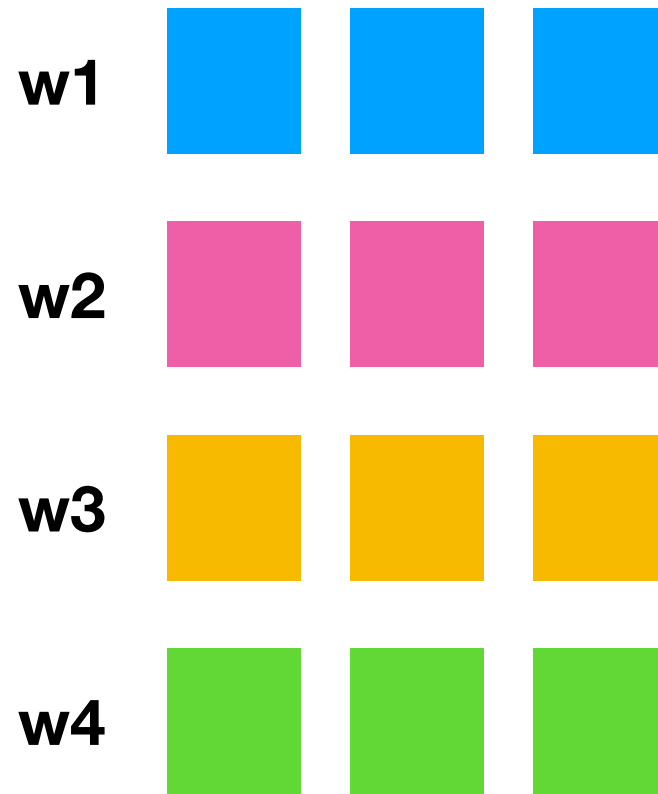
- Let's assume we have an embedding size=3 and a document composed of 4 words

- $\text{doc}=(w1, w2, w3, w4)$

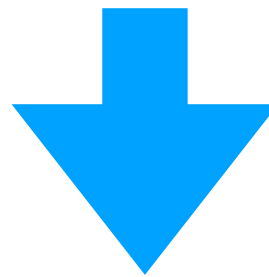
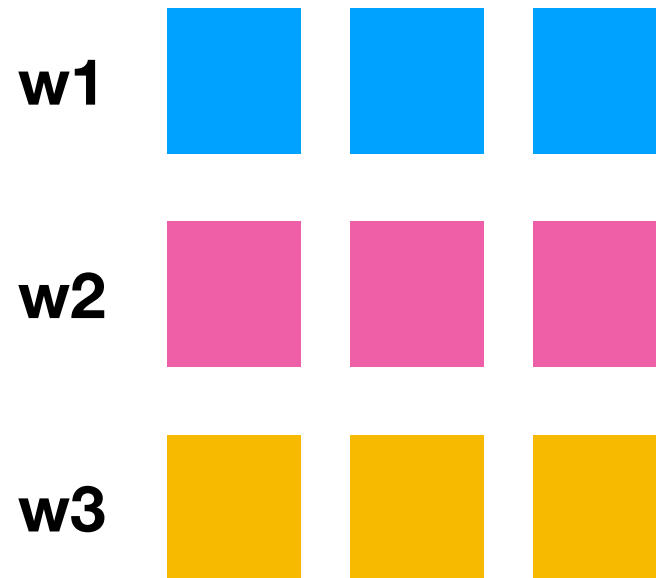
- embedding=

w1			
w2			
w3			
w4			

Flattening the Embedding



Flattening the Embedding



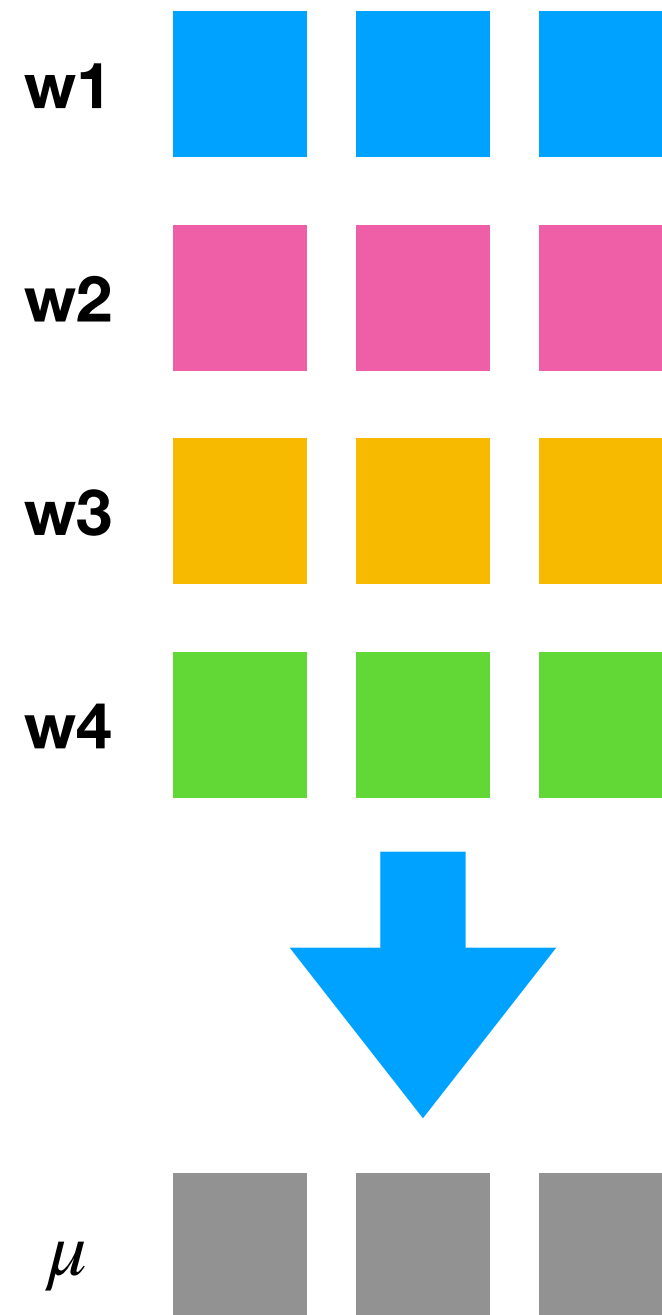
Flattening: Discussion

- Will create a huge number of input variables
- However, much more precise, as it does not aggregate
- Doable if you have a large dataset and enough computational power
- Often used with neural networks

Creating a document embedding

- The document embedding is obtained by averaging the embeddings of each word belonging to the document
- The vector is a centroid of the word vectors belonging to the document
- Hence, the number of features will be equal to the embedding size, and very reduced
- Particularly useful when you have limited data, you want to use algorithms like NaiveBayes or SVM, and you want to reduce the computation

Document embedding



Example

- In this example we will use word embeddings with SVM to classify spam
- Note: the application to NaiveBayes is a little bit more complicated, as it requires to normalize the embedding values between 0 and 1
 - This can be done using the [MinMaxScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html):
 - <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>

Text processing

```
import gensim
import numpy as np
import pandas as pd
from nltk.corpus import stopwords
import nltk
from sklearn.preprocessing import Normalizer
```

```
def transformText(text):
    # Convert text to lowercase
    text = text.lower()
    # Strip multiple whitespaces
    text = gensim.corpora.textcorpus.strip_multiple_whitespaces(text)
    # Removing all the stopwords
    stops = set(stopwords.words("english"))
    filtered_words = [word for word in text.split()]
    # Preprocessed text after stop words removal
    text = " ".join(filtered_words)
    # Remove the punctuation
    text = gensim.parsing.preprocessing.strip_punctuation(text)
    # Strip all the numerics
    text = gensim.parsing.preprocessing.strip_numeric(text)
    # Strip multiple whitespaces
    text = gensim.corpora.textcorpus.strip_multiple_whitespaces(text)
    return nltk.word_tokenize(text)
```

Notes

- Stop word removal and stemming not required
- Stemming would produce stems that are not in the pre-trained embedding
- However you can experiment with stop word removal

Embedding transformer

```
from sklearn.base import BaseEstimator, TransformerMixin

class MeanEmbeddingVectorizer(BaseEstimator, TransformerMixin):
    def __init__(self, model):
        self.model = model
        # if a text is empty we should return a vector of zeros
        # with the same dimensionality as all the other vectors
        self.dim = len(self.model[0])

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        newDocs = []
        for doc in X:
            vecs = []
            i = 0
            for w in doc:
                i += 1
                if w in self.model:
                    vecs.append(self.model[w])
                else:
                    vecs.append(np.asarray(np.zeros(self.dim)))
            if i > 0:
                n = np.asarray(np.mean(vecs, axis=0))
            else:
                n = np.asarray(np.zeros(self.dim))
            newDocs.append(np.asarray(n))
        return np.array(newDocs)
```

Discussion

- The constructor `__init__()` stores the pretrained model
- The transform method iterates for each word in the document and:
 - If the word is in the pre-trained embedding, accumulates its vector
 - if not, it adds a zeroes vector
- Then, if the set of obtained vectors is not empty, creates an average vector, otherwise it create a vector of zeroes

Loads the pre-trained embedding model and creates the pipeline

```
import gensim.downloader
print(list(gensim.downloader.info()['models'].keys()))
glove_vectors = gensim.downloader.load('glove-wiki-gigaword-100')

from sklearn.pipeline import Pipeline
from sklearn import svm

clf = Pipeline(
    [
        ("emb", MeanEmbeddingVectorizer(glove_vectors)),
        ("clf", svm.SVC(kernel="rbf", C=1000, gamma=0.001)),
    ]
)
```

Note

- If needed the pipeline could be calibrated, too

Creates the training and test set...

[illegible]

Training and prediction

```
clf.fit(X_train, y_train)
```

```
#performing the actual prediction  
predicted = clf.predict(X_test)
```

```
from sklearn import metrics  
print(pd.crosstab(y_test, predicted))  
print(metrics.classification_report(y_test, predicted))
```