Behind the Secrets of Large Language Models

Lecture 4

Word embeddings & Tokenization

Michael Färber, Simon Razniewski

ScaDS.AI
DRESDEN LEIPZIG

TECHNISCHE UNIVERSITÄT DRESDEN

Acknowledgment/License

- Shared under CC BY 4.0 with permission by
  - First part based on “Speech and Language Processing” (3rd ed. draft)
    Dan Jurafsky and James H. Martin, Lec. 6,
    https://web.stanford.edu/~jurafsky/slp3/
  - Second part based on „Advanced NLP“, Mohit Iyyer
    https://people.cs.umass.edu/~miyyer/cs685/

Annoucements

- No lecture and lab next week

- In the future, labs will also topically shift by a week, based on feedback
  - Day N: Lecture N Lab N-1
  - Day N+1: Lecture N+1 Lab N
  - …

Outline

0. **Projects**

1. Word embeddings (60 min)
    1. Vector semantics
    2. Words and vectors
    3. Cosine for computing word similarity
    4. TF-IDF
    5. Word2vec
    6. Word2vec: Learning the embeddings
    7. Properties of the embeddings

2. Tokenization (25 min)
    - Group work (5 min)

Project timeline

- Project period: 16.12.-10.2.
- Reports due 10.2.
- Peer grading: 17.-23.2.
- Presentations: 26.+28.2. all day

Project options

1. DataComp document selection
2. SemEval emotion detection
3. Long list elicitation
4. Saxon LLM construction

1. DataComp document selection

- https://www.datacomp.ai/dclm

- LLM architecture and training process is fixed, goal is to identify the best training dataset, from a larger noisy text dataset
  (to be covered in detail in lecture 5)

- Suggestion: The selection, in turn, could use LLMs

- Evaluation: Online leaderboard

2. SemEval Emotion detection

- https://github.com/emotion-analysis-project/SemEval2025-Task11
- Subtask 11a

- Text classification with 5 emotion labels
- Train an LLM for the task

- Suggestion: Consider auxiliary tasks, synthetic training data generation, or similar enhancements

- Evaluation: Online leaderboard

3. Long list elicitation from LLMs

- LLMs are trained to give terse answers
- Can you make them return all 6,400 species of mammal? All 16k inhabitated islands worldwide? All millions of people born in London? …

- Suggestion: Perform systematic prompt tuning, design iterative or recursive prompting schemes, use agent or self-criticism frameworks

- Open models or GPT-4o ($100)

- Evaluation: Precision and recall w.r.t. Wikidata

4. SäxyLLM: Build an LLM that speaks Saxon[2]

- Sächsisch is sexy, but current open models don’t speak it
- Build an LLM that speaks Saxonian by default
- Suggestion: Collect training data from web or try synthesizing it with commercial models, consider training objectives and evaluation
- Evaluation: TBD (Manual labelling or automated embedding similarity on test tasks)

[2] Note: „Saxon“ here refers to the German dialect spoken in today‘s state of Saxony https://en.wikipedia.org/wiki/Upper_Saxon_German

Grading

- 33% performance
- 33% report
- 33% presentation

- Criteria for report and presentation (all 5 equal weight):
  1. Discussion of and positioning w.r.t. state of the art
  2. Novelty of approach
  3. Technical depth
  4. Clarity and presentation
  5. Analysis/discussion

- Report template will be provided
  (max 8 pages double column ACL style)

Projects

- Survey on course webpage
  - Non-binding
- Kick-off meetings will follow
- Questions?

Outline

1. **Word embeddings (60 min)**
    1. **Vector semantics**
    2. Words and vectors
    3. Cosine for computing word similarity
    4. TF-IDF
    5. Word2vec
    6. Word2vec: Learning the embeddings
    7. Properties of the embeddings

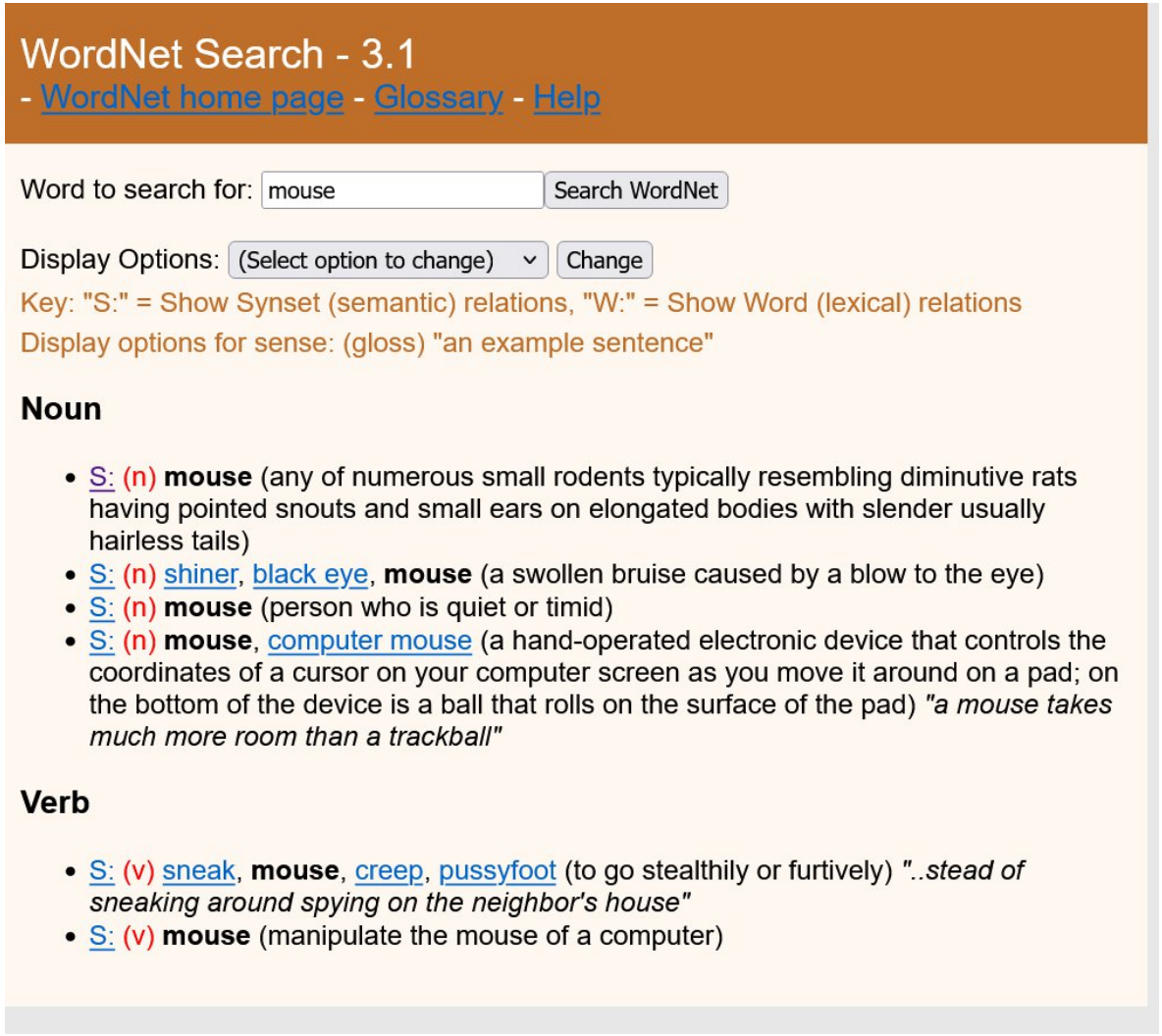
2. Tokenization (25 min)
    - Group work (5 min)

Desiderata for word meaning representation

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

- Have relations with each other
  - Synonymy
  - Antonymy
  - Hyponymy
  - Similarity
  - Relatedness
  - Connotation

Classic top-down approach: Wor meaning Database



WordNet Search - 3.1
- WordNet home page - Glossary - Help

Word to search for: mouse Search WordNet

Display Options: (Select option to change) Change

Key: "S:" = Show Synset (semantic) relations, "W:" = Show Word (lexical) relations

Display options for sense: (gloss) "an example sentence"

Noun

- S: (n) **mouse** (any of numerous small rodents typically resembling diminutive rats having pointed snouts and small ears on elongated bodies with slender usually hairless tails)
- S: (n) shiner, black eye, **mouse** (a swollen bruise caused by a blow to the eye)
- S: (n) **mouse** (person who is quiet or timid)
- S: (n) **mouse**, computer mouse (a hand-operated electronic device that controls the coordinates of a cursor on your computer screen as you move it around on a pad; on the bottom of the device is a ball that rolls on the surface of the pad) *"a mouse takes much more room than a trackball"*

Verb

- S: (v) sneak, **mouse**, creep, pussyfoot (to go stealthily or furtively) *"..stead of sneaking around spying on the neighbor's house"*
- S: (v) **mouse** (manipulate the mouse of a computer)

http://wordnetweb.princeton.edu/perl/webwn

Synonymy
Antonymy
Hyponymy
~~Similarity~~
~~Relatedness~~
~~Connotation~~

Computational models of word meaning

- Can we build a theory of how to represent word meaning, that accounts for most of the desiderata?
- We'll introduce **vector semantics**
  - The standard model in modern language processing!
  - Handles many of our goals!

Ludwig Wittgenstein

- Wittgenstein:
  "The meaning of a word is its use in the language“

- One way to define "usage": words are defined by their environments (the words around them)

- Zellig Harris (1954):
  - **If A and B have almost identical environments we say that they are synonyms**.

What does recent English borrowing *ongchoi* mean?

- Suppose you see these sentences:
  - Ong choi is delicious **sautéed with garlic**.
  - Ong choi is superb **over rice**
  - Ong choi **leaves** with salty sauces
- And you've also seen these:
  - …spinach **sautéed with garlic over rice**
  - Chard stems and **leaves** are **delicious**
  - Collard greens and other **salty** leafy greens
- Conclusion:
  - Ongchoi is a leafy green like spinach, chard, or collard greens
    - We could conclude this based on words like "leaves" and "delicious" and "sauteed"

Ongchoi: *Ipomoea aquatica "Water Spinach"*

空心菜
kangkong
rau muống
Wasserspinat
…

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

	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
	leadership	0.983		empty	0.081

NRC VAD Lexicon
(Mohammad 2018)

- 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" for historic reasons
- 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

Intuition: why vectors?

- Consider sentiment analysis:
  - With **words**, a feature is a word identity
    - Feature 5: 'The previous word was "terrible"'
    - requires **exact same word** to be in training and test
  - With **embeddings**:
    - Feature is a word vector
    - 'The previous word was vector [35,22,17…]
    - Now in the test set we might see a similar vector [34,21,14]
    - We can generalize to **similar but unseen** words!!!

We'll discuss 2 kinds of embeddings

- tf-idf
  - 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**

Outline

1. **Word embeddings (60 min)**
    1. Word meaning
    2. Vector semantics
    3. **Words and vectors**
    4. Cosine for computing word similarity
    5. TF-IDF
    6. Word2vec
    7. Word2vec: Learning the embeddings
    8. Properties of the embeddings

2. Tokenization (30 min)

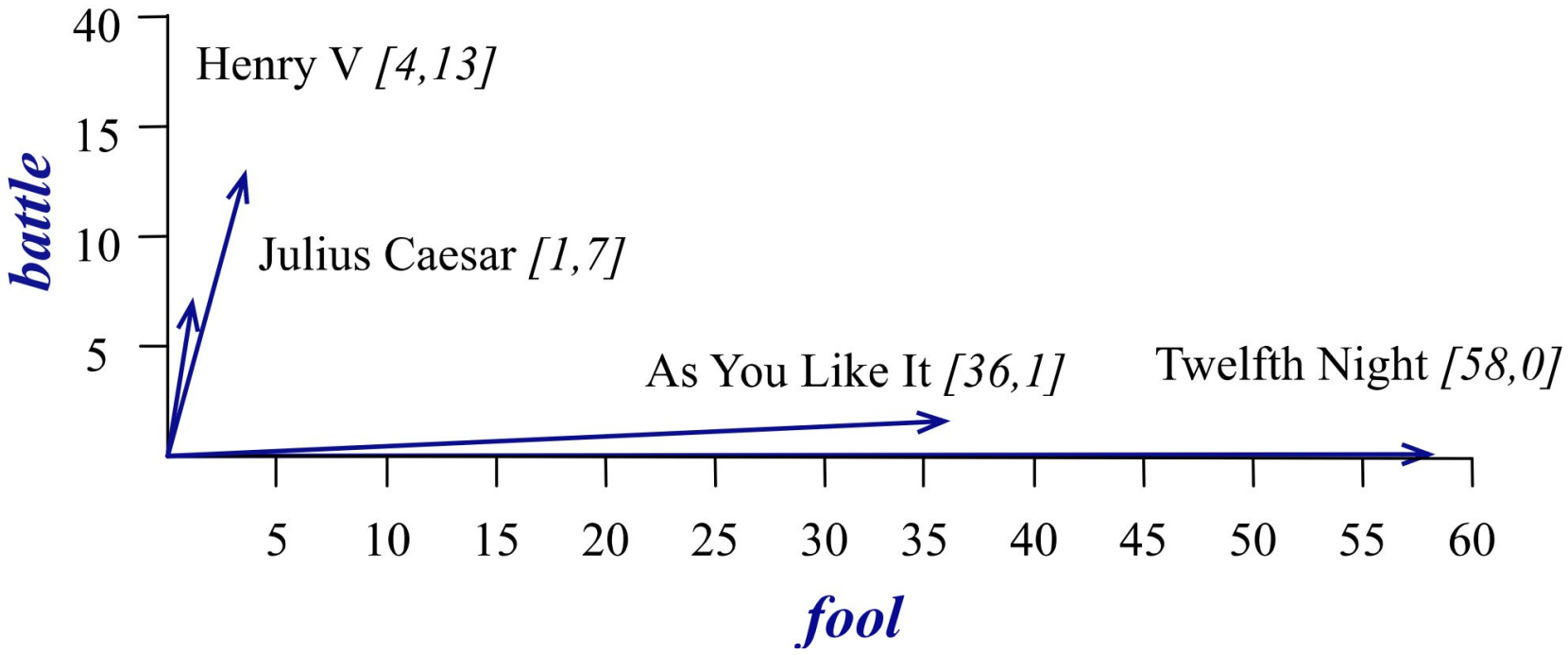
Term-document matrix

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

(Shakespeare drama's)

Visualizing document vectors



Vectors are the basis of information retrieval

	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

Vectors are similar for the two comedies

But comedies are different than the other two

Comedies have more *fools* and *wit* and fewer *battles*.

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"

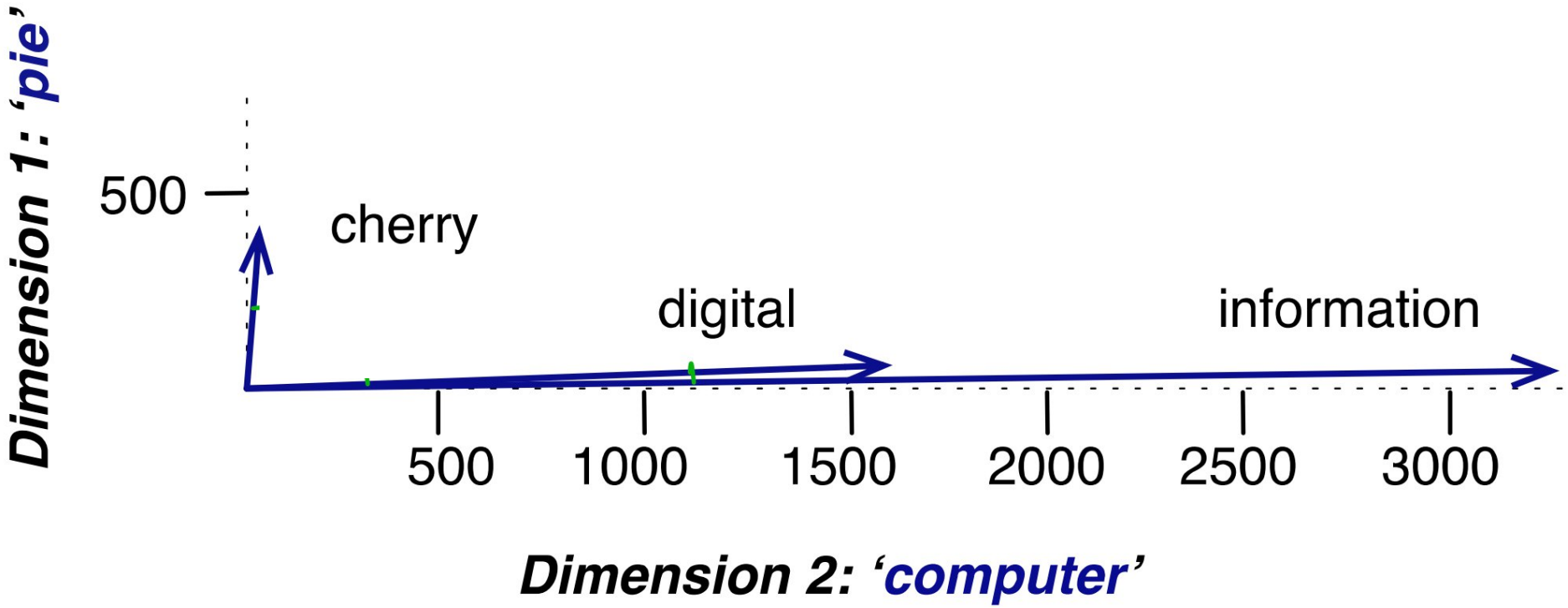
fool is "the kind of word that occurs in comedies, especially Twelfth Night"

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	...


Dimension 1: 'pie'
500
cherry
digital
information
500
1000
1500
2000
2500
3000
Dimension 2: 'computer'

Outline

1. **Word embeddings (60 min)**
    1. Word meaning
    2. Vector semantics
    3. Words and vectors
    4. **Cosine for computing word similarity**
    5. TF-IDF
    6. Word2vec
    7. Word2vec: Learning the embeddings
    8. Properties of the embeddings

2. Tokenization (30 min)

Computing word similarity: Dot product and cosine

- The dot product between two vectors is a scalar:

$$\text{dot product}(\mathbf{v},\mathbf{w}) = \mathbf{v}\cdot\mathbf{w} = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \ldots + v_N w_N$$

- The dot product tends to be high when the two vectors have large values in the same dimensions
- Dot product can thus be a useful similarity metric between vectors

Problem with raw dot-product

- Dot product favors long vectors
- Dot product is higher if a vector is longer (has higher values in many dimension)
- Vector length:

$$|\mathbf{v}| = \sqrt{\sum_{i=1}^{N} v_i^2}$$

- Frequent words (of, the, you) have long vectors (since they occur many times with other words).
- So dot product overly favors frequent words

Alternative: cosine for computing word similarity

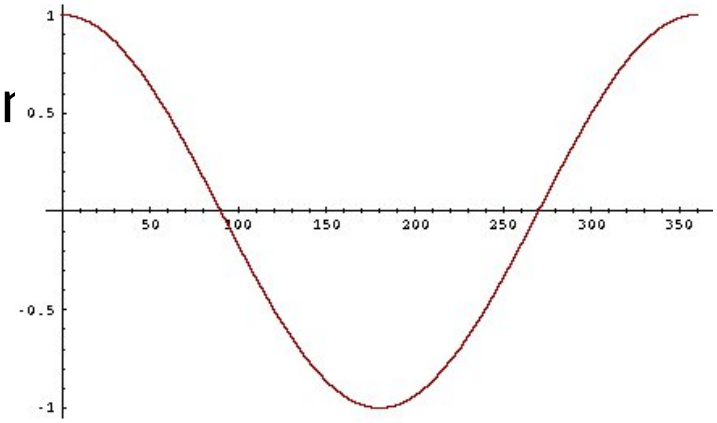
$$\text{cosine}(\vec{v},\vec{w}) = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

Based on the definition of the dot product between two vectors a and b

$$\mathbf{a}\cdot\mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$$
$$\frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}||\mathbf{b}|} = \cos\theta$$

Cosine as a similarity metric

- -1: vectors point in opposite directior
- +1: vectors point in same directions
- 0: vectors are orthogonal



- But since raw frequency values are non-negative, the cosine for term-term matrix vectors ranges from 0–1

Exercise time

$$\text{cosine}(\vec{v},\vec{w}) = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

Compute the similarity of V1 to the other vectors:

- V1 = (1, 1, 1, 1, 0, 0, 0, 0)
- V2 = (1, 1, 1, 1, 0, 0, 0, 0)
- V3 = (1, 1, 0, 0, 0, 0, 1, 1)
- V4 = (1, 0, 0, 0, 0, 1, 1, 1)
- V5 = (0, 0, 0, 0, 1, 1, 1, 1)

V1 = (3, 4)

V2 = (3, 4)
V3 = (4, 3)
V4 = (0, 4)
V5 = (0, 3)

Cosine examples

$$\cos(\vec{v},\vec{w}) = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\vec{v}}{|\vec{v}|}\cdot\frac{\vec{w}}{|\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

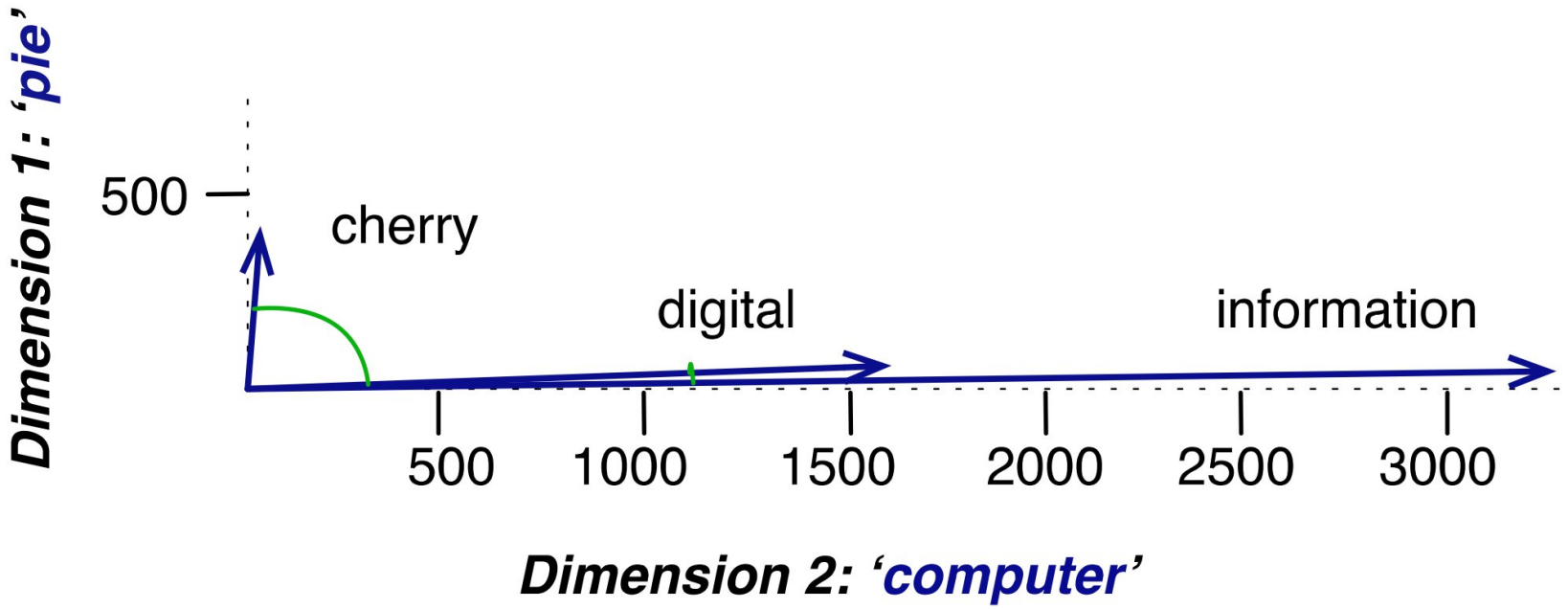
	pie	data	computer
cherry	442	8	2
digital	5	1683	1670
information	5	3982	3325

$$\cos(\text{cherry},\text{information}) = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = .017$$

$$\cos(\text{digital},\text{information}) = \frac{5*5+1683*3982+1670*3325}{\sqrt{5^2+1683^2+1670^2}\sqrt{5^2+3982^2+3325^2}} = .996$$

40

Visualizing cosines (well, angles)



Outline

1. **Word embeddings (60 min)**
    1. Word meaning
    2. Vector semantics
    3. Words and vectors
    4. Cosine for computing word similarity
    5. **TF-IDF**
    6. Word2vec
    7. Word2vec: Learning the embeddings
    8. Properties of the embeddings

2. Tokenization (30 min)

Frequency problem II

- Cosine fights one frequency problem: Some words that we want to represent are much more common than others
  - Therefore co-occur with other words more often
  - Computing raw dot product with such words tends to be large
- But there's another problem with frequencies
  - Some words very frequently appear in the context of almost every word
  - *E.g.*, *the*, *it*, *or*, *they*, …
  - Correspondingly, embedding vectors of all words would look similar in the largest dimensions

Two common solutions for word weighting

- **tf-idf:** tf-idf value for word t in document d:

$$w_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$$

Words like "the" or "it" have very low idf

- **PMI:** (Pointwise mutual information)
  - $\textbf{PMI}\,(w_1, w_2) = log \frac{p(w_1, w_2)}{p(w_1)p(w_2)}$

See if words like "good" appear more often with "great" than we would expect by chance

Term frequency (tf) in the tf-idf algorithm

- We could imagine using raw count:
- $\text{tf}_{t,d} = \text{count}(t,d)$
- But instead of using raw count, we usually squash a bit:

$$\text{tf}_{t,d} = \begin{cases} 1 + \log_{10} \text{count}(t,d) & \text{if } \text{count}(t,d) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Document frequency (df)

- df_t *is* the number of documents *t* occurs in.
- (note this is not collection frequency: total count across all documents)
- "*Romeo*" is very distinctive for one Shakespeare play:

	Collection Frequency	Document Frequency
Romeo	113	1
action	113	31

Inverse document frequency (idf)

$$\text{idf}_t = \log_{10}\left(\frac{N}{\text{df}_t}\right)$$

N is the total number of documents in the collection

Word	df	idf
Romeo	1	1.57
salad	2	1.27
Falstaff	4	0.967
forest	12	0.489
battle	21	0.246
wit	34	0.037
fool	36	0.012
good	37	0
sweet	37	0

What is a document?

- Could be a play or a Wikipedia article
- But for the purposes of tf-idf, documents can be **anything**; we often call each paragraph a document!

Final tf-idf weighted value for a word

- Raw counts: $w_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$

	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

- tf-idf:

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.246	0	0.454	0.520
good	0	0	0	0
fool	0.030	0.033	0.0012	0.0019
wit	0.085	0.081	0.048	0.054

Outline

1. **Word embeddings (60 min)**
    1. Word meaning
    2. Vector semantics
    3. Words and vectors
    4. Cosine for computing word similarity
    5. TF-IDF
    6. **Word2vec**
    7. Word2vec: Learning the embeddings
    8. Properties of the embeddings

2. Tokenization (30 min)

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 sparse dimensions usually fail to capture this
  - **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 called LSA – Latent Semantic Analysis
- 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
  - Future lectures…

Simple static embeddings you can download!

- Word2vec (Mikolov et al)
- https://code.google.com/archive/p/word2vec/

- GloVe (Pennington, Socher, Manning)
- http://nlp.stanford.edu/projects/glove/

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 counts 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:

[target]

…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 (**w**ord, **c**ontext) 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:
  - Similarity(w,c) $\propto w \bullet c$
- We'll need to normalize to get a probability
  - (cosine isn't a probability either)

Turning dot products into probabilities

- Sim(w,c) ≈ w · c
- To turn this into a probability
- We'll use the sigmoid from logistic regression:

$$P(+|w,c) = \sigma(c \cdot w) = \frac{1}{1+\exp(-c \cdot w)}$$

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

$$= \sigma(-c \cdot w) = \frac{1}{1+\exp(c \cdot w)}$$

How Skip-Gram Classifier computes *P*(+|*w*, *c*)

$$P(+|w,c) = \sigma(c \cdot w) = \frac{1}{1+\exp(-c \cdot w)}$$

- This is for one context word, but we can have lots of context words.
- We'll assume independence and just multiply them:

$$P(+|w,c_{1:L}) = \prod_{i=1}^{L} \sigma(c_i \cdot w)$$

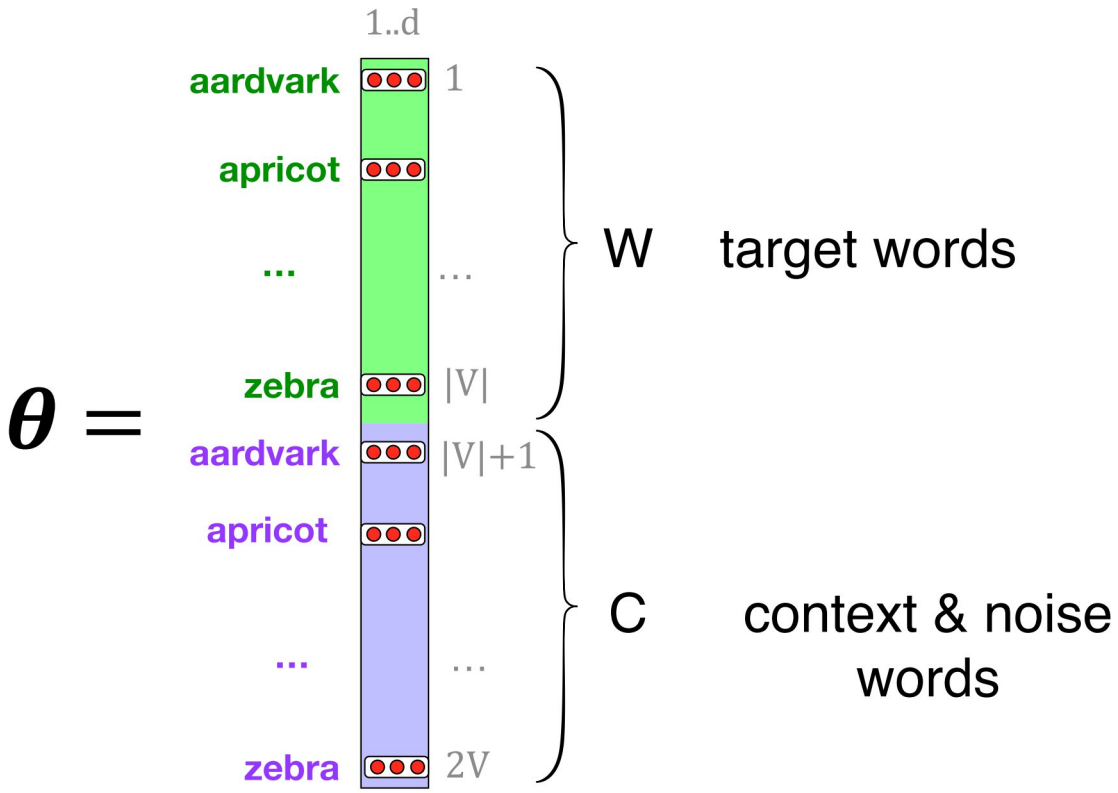
$$\log P(+|w,c_{1:L}) = \sum_{i=1}^{L} \log \sigma(c_i \cdot w)$$

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.

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



Outline

1. **Word embeddings (60 min)**
    1. Word meaning
    2. Vector semantics
    3. Words and vectors
    4. Cosine for computing word similarity
    5. TF-IDF
    6. Word2vec
    7. **Word2vec: Learning the embeddings**
    8. Properties of the embeddings

2. Tokenization (30 min)

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.

Loss function for one *w with* c_{pos} , c_{neg1} …c_{negk}

- Maximize the similarity of the target with the actual context words, and minimize the similarity of the target with the *k* negative sampled non-neighbor words.

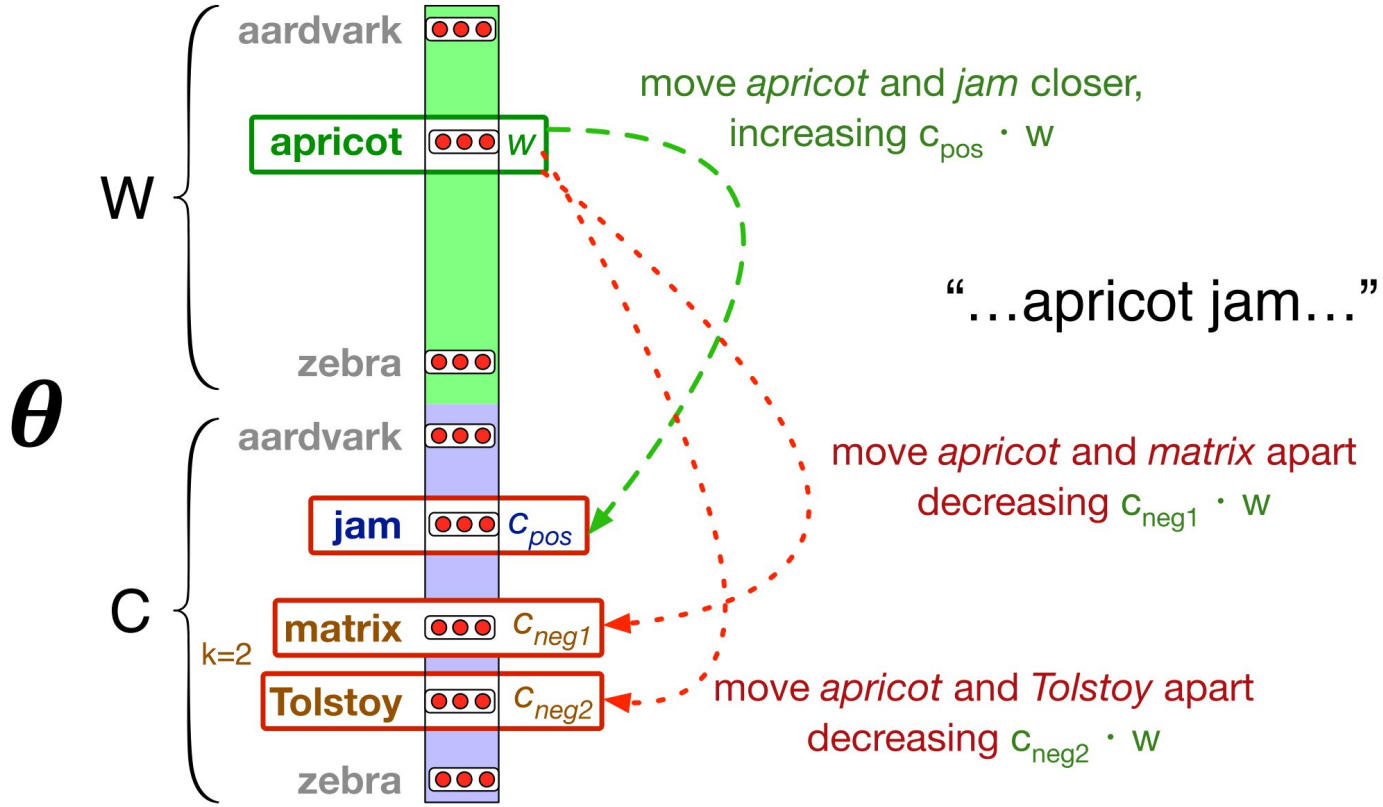
$$
\begin{aligned}
L_{CE} &= -\log\left[P(+|w,c_{pos})\prod_{i=1}^{k}P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos})+\sum_{i=1}^{k}\log P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos})+\sum_{i=1}^{k}\log\left(1-P(+|w,c_{neg_i})\right)\right] \\
&= -\left[\log \sigma(c_{pos}\cdot w)+\sum_{i=1}^{k}\log\sigma(-c_{neg_i}\cdot w)\right]
\end{aligned}
$$

Learning the classifier

- How to learn?
  - Stochastic gradient descent!

- We'll adjust the word weights to
  - make the positive pairs more likely
  - and the negative pairs less likely,
  - over the entire training set.

Intuition of one step of gradient descent



Two sets of embeddings

- SGNS learns two sets of embeddings
- Target embeddings matrix W
- Context embedding matrix C
- It's common to just add them together, representing word *i* as the vector $\mathbf{w}_i + \mathbf{c}_i$

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.

Outline

1. **Word embeddings (60 min)**
    1. Word meaning
    2. Vector semantics
    3. Words and vectors
    4. Cosine for computing word similarity
    5. TF-IDF
    6. Word2vec
    7. Word2vec: Learning the embeddings
    8. **Properties of the embeddings**

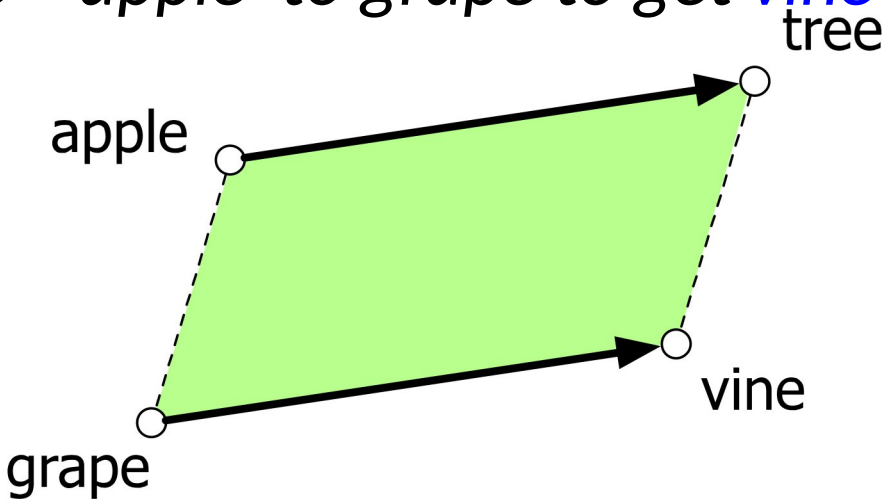
2. Tokenization (30 min)

The kinds of neighbors depend on window size

- **Small windows** (C= +/- 2): nearest words are syntactically similar words in same taxonomy
  - *Hogwarts* nearest neighbors are other fictional schools
    - *Sunnydale*, *Evernight*, *Blandings*
- **Large windows** (C= +/- 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* $\overrightarrow{tree}$ – $\overrightarrow{apple}$ *to* $\overrightarrow{grape}$ *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)

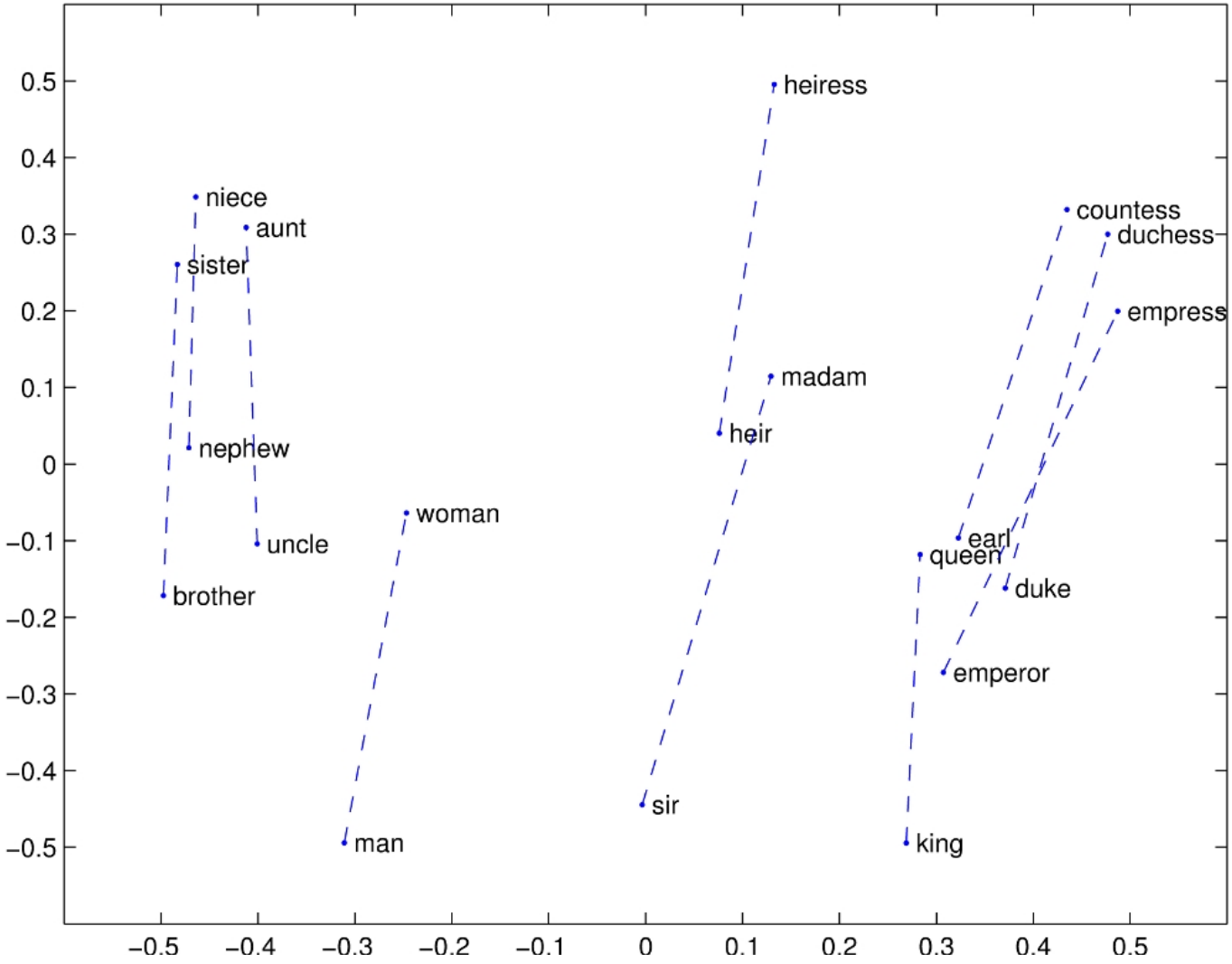
$\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ is close to $\overrightarrow{\text{queen}}$

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

- For a problem a:a*::b:b*, the parallelogram method is:

$$\hat{b}^* = \underset{x}{\text{argmax}}\ \text{distance}(x, a^* - a + b)$$

Structure in GloVE Embedding space



Embeddings as a window onto historical semantics

Train embeddings on different decades of historical text to see meanings shift

~30 million books, 1850-1990, Google Books data

William L. Hamilton, Jure Leskovec, and Dan Jurafsky. 2016. Diachronic Word Embeddings Reveal Statistical Laws of Semantic Change. Proceedings of ACL.

Embeddings reflect cultural bias!

Bolukbasi, Tolga, Kai-Wei Chang, James Y. Zou, Venkatesh Saligrama, and Adam T. Kalai. "Man is to computer programmer as woman is to homemaker? debiasing word embeddings." In *NeurIPS*, pp. 4349-4357. 2016.

- 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

Word2Vec demo

- http://epsilon-it.utu.fi/wv_demo/

Outline

1. Word embeddings (60 min)
    1. Word meaning
    2. Vector semantics
    3. Words and vectors
    4. Cosine for computing word similarity
    5. TF-IDF
    6. Word2vec
    7. Word2vec: Learning the embeddings
    8. Properties of the embeddings

2. **Tokenization (25 min)**
    - Group work (5 min)

Tokenization

- We've seen idea of embeddings, but what actually do we embed?
- Every word?

Input text: students opened their books

Input token IDs: ***11*** ***298*** ***34*** ***567***

This tokenization step requires an external *tokenizer* to detect word boundaries!

Problem 1: Tokenization is not simple

- Not just a split on whitespace and punctuation…

Mr. **O'Neill** thinks that the boys' stories about San Francisco **aren't** amusing.

- Word tokenizers require lots of specialized rules about how to handle specific inputs
  - Check out spaCy's tokenizers! (https://spacy.io/)

Problem 2: Unknown words

- What happens when we encounter a word at test time that we've never seen in our training data?
  - With word level tokenization, we have no way of assigning an index to an unseen word!
  - This means we don't have a word embedding for that word and thus cannot process the input sequence
- Workaround: replace low-frequency words in training data with a special <UNK> token, use this token to handle unseen words at test time too
  - Why use <UNK> tokens during training?

Limitations of <UNK>

- We lose lots of information about texts with a lot of rare words / entities

The chapel is sometimes referred to as "Hen Gapel Lligwy" ("*hen*" being the Welsh word for "old" and "*capel*" meaning "chapel").

The chapel is sometimes referred to as "Hen <unk> <unk>" ("hen" being the Welsh word for "old" and "<unk>" meaning "chapel").

Problem 3: Sparsity

- Word-level tokenization treats different forms of the same word (e.g., “open”, “opened”, “opens”, “opening”, etc) as separate types —> separate embeddings for each

This can be problematic especially when training over smaller datasets, why?

An alternative: character tokenization

- Small vocabulary, just the number of unique characters in the training data!
- However, you pay for this with longer input sequences.
- Look forward: Why would this be a problem for LLMs?

2016: subword tokenization

- Developed for machine translation by Sennrich et al., ACL 2016

  "The main motivation behind this paper is that the translation of some words is transparent in that they are translatable by a competent translator even if they are novel to him or her, based on a translation of known subword units such as morphemes or phonemes."

- Later used in BERT, T5, RoBERTa, GPT, etc.

- Relies on a simple algorithm called *byte pair encoding* (Gage, 1994)

Byte pair encoding

- Form base vocabulary
  (all characters that occur in the training data)

word	frequency
hug	10
pug	5
pun	12
bun	4
hugs	5

- Base vocab: b, g, h, n, p, s, u

Example from https://huggingface.co/transformers/tokenizer_summary.html

Byte pair encoding

- Now, count up the frequency of each character *pair* in the data, and choose the one that occurs most frequently

word	frequency
h+u+g	10
p+u+g	5
p+u+n	12
b+u+n	4
h+u+g+s	5

character pair	frequency
u+g	20
p+u	17
u+n	16
h+u	15
g+s	5

...

Example from https://huggingface.co/transformers/tokenizer_summary.html

Byte pair encoding

- Now, choose the most common pair (ug) and then merge the characters together into one symbol. Add this new symbol to the vocabulary.
- New vocab: b, g, h, n, p, s, u, ug
- Then, retokenize the data

word	frequency
h+*ug*	10
p+*ug*	5
p+u+n	12
b+u+n	4
h+*ug*+s	5

character pair	frequency
u+n	16
h+ug	15
p+u	12
p+ug	5
ug+s	5

Example from https://huggingface.co/transformers/tokenizer_summary.html

Byte pair encoding

- Keep repeating this process! This time we choose *un* to merge
- New vocab: b, g, h, n, p, s, u, ug, un
- Next time we choose *h*+*ug*, etc.

word	frequency
h+*ug*	10
p+*ug*	5
p+un	12
b+un	4
h+*ug*+s	5

character pair	frequency
h+*ug*	15
p+*u*	12
p+*ug*	5
ug+*s*	5
...	

Example from https://huggingface.co/transformers/tokenizer_summary.html

Byte pair encoding

- Eventually, after a fixed number of merge steps, we stop

  - Final vocab: b, g, h, n, p, s, u, *ug*, *un*, *hug*

Example from https://huggingface.co/transformers/tokenizer_summary.html

Byte pair encoding

- Space is commonly just treated like any other character
- To avoid <UNK>, all possible characters / symbols need to be included in the base vocab. This can be a lot if including all unicode characters (there are ~138K unicode symbols)!
- GPT-2 uses *bytes* as the base vocabulary (size 256) and then applies BPE on top of this sequence (with some rules to prevent certain types of merges).
- Commonly have vocabulary sizes of 32K to 64K

Other subword encoding schemes

- WordPiece (Schuster et al., ICASSP 2012): merge by likelihood as measured by language model, not by frequency

- SentencePiece (Kudo et al., 2018): can do subword tokenization without pretokenization *(good for languages that don't always separate words w/ spaces)*, although pretokenization usually improves performance
- Tokenizer matters, especially for multilingual models

[Tokenizer Choice For LLM Training: Negligible or Crucial? Ali et al., 2024]

Tokens and LLMs

- Tokens are the staple of LLMs, input and charging unit

GPT-4o

Model	Pricing	Pricing with Batch API*
gpt-4o	$2.50 / 1M input tokens	$1.25 / 1M input tokens
	$1.25 / 1M cached** input tokens	
	$10.00 / 1M output tokens	$5.00 / 1M output tokens

- OpenAI Rule of thumb: one token = ca. 4 characters of text for English text. This translates to roughly ¾ of a word (so 100 tokens ~= 75 words).

Table 1: Premiums with respect to English on FLORES-200 for several **English-centric** models. The top and bottom three languages for any tokenizer and the ones discussed in the text are shown.

	GPT-2 RoBERTa	ChatGPT GPT-4	FlanT5
Bulgarian	5.51	2.64	—
Burmese	16.89	11.70	—
Chinese (Simplified)	3.21	1.91	—
Dzongkha	16.36	12.33	—
English	1.00	1.00	1.00
French	2.00	1.60	1.60
German	2.14	1.58	1.37
Italian	2.01	1.64	2.18
Japanese	3.00	2.30	—
Jingpho	2.65	2.35	3.41
Maori	2.45	2.35	3.28
Norwegian Bokmål	1.86	1.56	2.24
Odia	13.38	12.48	—

[Language Model Tokenizers Introduce Unfairness Between Languages]

Demo: OpenAI Tokenizer

- https://platform.openai.com/tokenizer

Outline

1. Word embeddings (60 min)
    1. Word meaning
    2. Vector semantics
    3. Words and vectors
    4. Cosine for computing word similarity
    5. TF-IDF
    6. Word2vec
    7. Word2vec: Learning the embeddings
    8. Properties of the embeddings

2. Tokenization (25 min)
    - **Group work (5 min)**

Group work

- Discuss the following questions with your neighbor:

1. What is meant by sparse vs. dense embeddings? What are advantages of either?

2. What is normalization by vector length, versus by IDF? Why do both matter?

3. Why do we need dedicated tokenizers? Why not just split at word borders?

Take home – Lecture 4

1. Vector space model as foundation of meaning representation
2. Popular static embedding model: Word2Vec
   - We will later learn about contextual embedding models like BERT
3. Tokenization of text using byte-pair-encoding algorithm

- Lab today:
  - Select your own corpus
  - Build a tokenizer
  - Build word embeddings
  - Discover analogies