Text Representations

Today’s agenda

Embeddings

Tokenization

Embeddings

How to represent a word as a vector?

$$
\begin{aligned}
v_{\text{cat}} &= \left[\quad\right] \\
v_{\text{dog}} &= \left[\quad\right] \\
v_{\text{king}} &= \left[\quad\right] \\
v_{\text{queen}} &= \left[\quad\right]
\end{aligned}
$$

A naive option: One-hot encoding

$$
\begin{aligned}
v_{\text{cat}} &= [1, 0, 0, 0, \ldots] \\
v_{\text{dog}} &= [0, 1, 0, 0, \ldots] \\
v_{\text{king}} &= [0, 0, 1, 0, \ldots] \\
v_{\text{queen}} &= [0, 0, 0, 1, \ldots]
\end{aligned}
$$

Lookup table: cat = 1, dog = 2, king = 3, queen = 4

Vector dimension = number of words in vocabulary (e.g., 500,000+)

Limitations of naive option

In a web search, if a user searches for “Seattle motel”, they may also want to find pages mentioning “Seattle hotel”.

However, the vectors for *motel* and *hotel* are **orthogonal**.

There is no natural notion of textual **similarity** for one-hot vectors!

$$v_{\text{motel}} = [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]$$
$$v_{\text{hotel}} = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]$$

Limitations of naive option

We want vectors that capture *meaning*

What if ...

$$v_{\text{hotel}} = [35, 22, 17, \dots]$$
$$v_{\text{motel}} = [34, 21, 19, \dots]$$

Semantics

Denotational semantics

signifier (symbol) ⟺ signified (idea or thing)

“tree” ⟺ {🌳, 🌲, 🌴, 🌲, 🎄, ...}

“the largest reptile” ⟺ 🐊

“the largest reptile” ⟺ 🐉

Semantics

Synonyms: couch/sofa, car/automobile, filbert/hazelnut

There's no true synonymity! Words may still may differ based on politeness, slang, register, genre, …

Relatedness, e.g. coffee/cup, house/door, chef/menu

Semantic field: words that cover a particular domain, with structured relations.

Antonyms: dark/light, rise/fall, up/down

Hypernyms & hyponyms: vehicle → car/truck, ungulates → cow/horse

Affective meanings or connotations, e.g. "love" has positive valence

Distributional hypothesis

"You shall know a word by the company it keeps"

- J.R. Firth 1957

Example:

```
The _____ sat on the throne.
```

Words that occur in similar contexts tend to have similar meanings.

Turns semantics into a statistical problem over text!

Guess the meaning

Everybody likes **tesgüino**.

We make **tesgüino** out of corn.

A bottle of **tesgüino** is on the table.

Don’t have **tesgüino** before you drive.

Example from Jacob Eistenstein’s NLP textbook

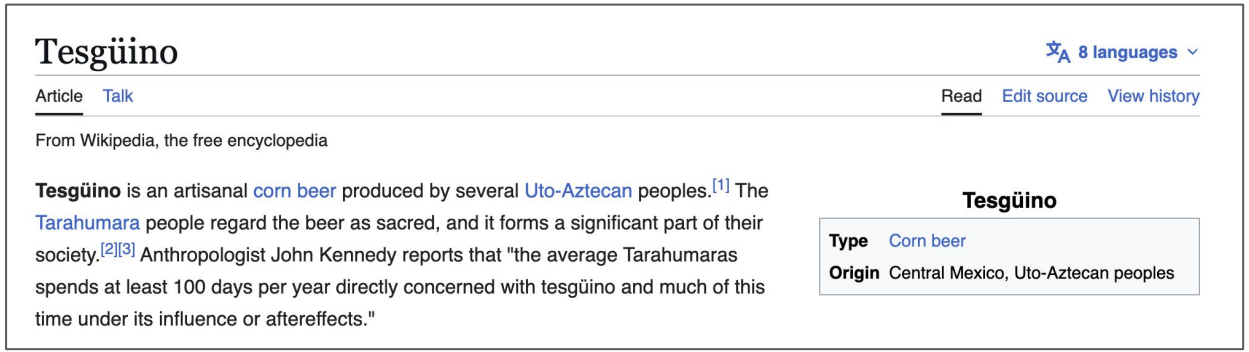
Guess the meaning

Everybody likes **tesgüino**.

We make **tesgüino** out of corn.

A bottle of **tesgüino** is on the table.

Don’t have **tesgüino** before you drive.



Tesgüino

8 languages

Article Talk Read Edit source View history

From Wikipedia, the free encyclopedia

Tesgüino is an artisanal corn beer produced by several Uto-Aztecan peoples.[1] The Tarahumara people regard the beer as sacred, and it forms a significant part of their society.[2][3] Anthropologist John Kennedy reports that "the average Tarahumaras spends at least 100 days per year directly concerned with tesgüino and much of this time under its influence or aftereffects."

Tesgüino

Type	Corn beer
Origin	Central Mexico, Uto-Aztecan peoples

Example from Jacob Eistenstein’s NLP textbook

Distributional semantics

The **king** sat on the throne.

Legend says that **King** Arthur was a leader in battle.

The court witnessed the mad **king** knighting his jester.

Key idea: use the many contexts around *king* in a dataset, or corpus, to build up a vector representation of *king*.

How can we do this computationally?

One idea: word-word co-occurrence matrix

Use word co-occurrence counts to represent the meaning of words.

Each word is represented by its row vector in the matrix.

Example

Context window = 4 words to the left + 4 words to the right

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	…
…	…	…	…	…	…	…	…	…

Example

Context window = 4 words to the left + 4 words to the right

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	…
…	…	…	…	…	…	…	…	…

Word similarity

Cosine of the angle between the two vectors

$$\cos(u, v) = \frac{u \cdot v}{||u||\ ||v||}$$

➔ The larger the cosine, the more similar the two vectors are

Q: Why cosine similarity instead of dot product $u \cdot v$?

Problem with dot product

Dot product favors long vectors

It is higher if a vector is longer, which happens when it has high values in multiple dimensions

With a word co-occurrence approach, frequent words (e.g. *you*, *of, the*) have long vectors because they co-occur many times with other words.

$$\cos(u, v) = \frac{u \cdot v}{||u|| \ ||v||}$$

Question

What is the range of $\cos(u, v)$ if our vectors are co-occurrence count vectors?

(A) [-1, 1]

(B) [0, 1]

(C) (0, 1)

(D) (-1, 1)

(E) (−∞, +∞)

Question

What is the range of $\cos(u, v)$ if our vectors are co-occurrence count vectors?

(A) [-1, 1]

(B) [0, 1]

(C) (0, 1)

(D) (-1, 1)

(E) (−∞, +∞)

The answer is (b).

An issue with our approach

Raw frequency counts are not ideal

- Very informative: **pie** and **cherry** co-occur
- Not so informative: **the** and **pie** co-occur

Solution: use a **weighted function** instead of raw counts

Using a weighted function: PMI

Pointwise mutual information (PMI)

- Does word *w* and context word *c* occur together more or less than if they were independent?

$$\text{PMI}(w,\, c) = \log_2 \frac{P\,(w,\, c)}{P\,(w) P\,(c)}$$

Using a weighted function: PMI

Pointwise mutual information (PMI)

- Does “pie” occur more often than we’d expect near “cherry”?

$$\text{PMI}(\text{pie},\ \text{cherry}) = \log_2 \frac{P\left(\text{cherry}\ \&\ \text{pie}\right)}{P\left(\text{cherry}\right)P\left(\text{pie}\right)}$$

Another issue with our approach

The vectors in the word-word co-occurrence matrix are

- High-dimensional, or large vocabulary size
- Sparse: most values are 0's

Solution: represent words as **short** (50-300 dimensional) & **dense** vectors

Fewer dimensions are more comfy for ML systems (curse of dimensionality)

They **generalize** better too, by capturing higher-order co-occurrence

- "co-occurs with *car*" and "co-occurs with *automobile*" do not need to be distinct dimensions

How to get dense vectors?

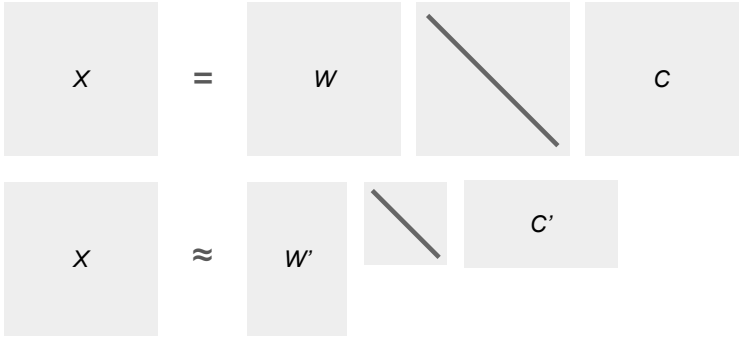
From count-based methods: **singular value decomposition**

Decompose a co-occurrence matrix *X* for vocabulary *V* into three matrices.

The middle matrix is a diagonal matrix of |*V*| singular values.

Keep only the top *k* singular values, so that each row of *W*, which contains our new embeddings, is only *k* dimensions.

Embedding now have fewer dims! |*V*| >> *k*



How to get dense vectors?

Prediction-based methods

Vectors are created by training a classifier to predict whether word *w* is likely to appear in the context of word *c*.

Word representations are **learned** from text.

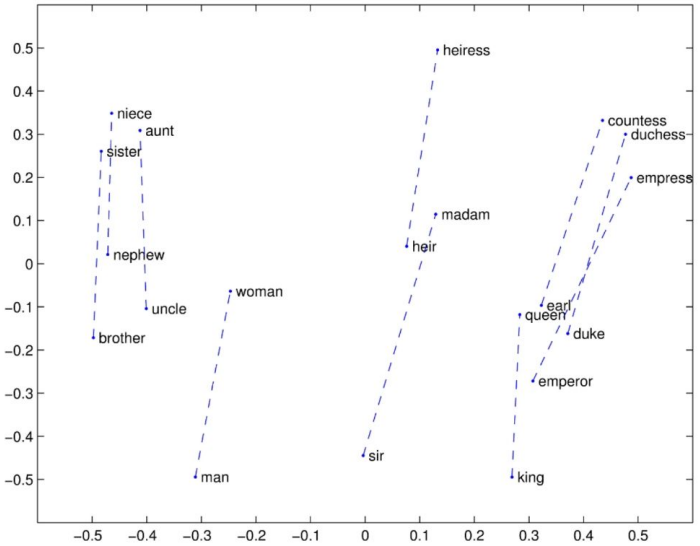
Each dimension isn't interpretable, but **cos(*cherry*, *pie*) > cos(*digital*, *pie*)**

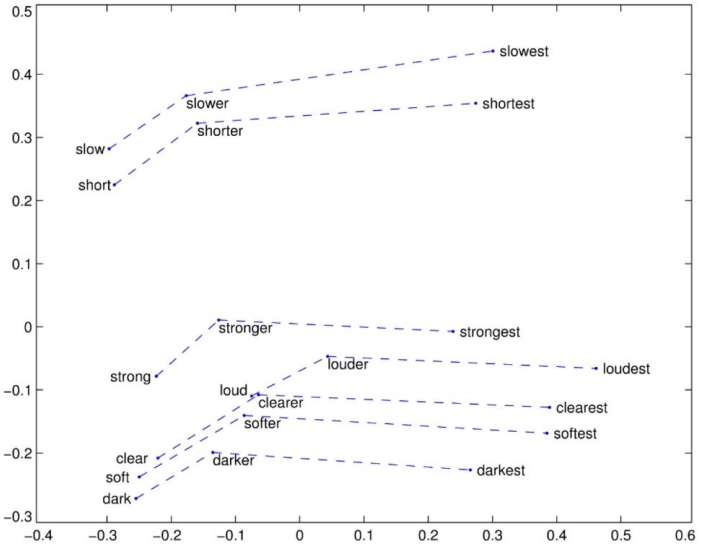
Inputs:

- A large textual dataset
- *V*, or a pre-defined vocabulary
- *d*, or vector dimension, e.g. 300

Output: |*V*| x *d* matrix of word embeddings

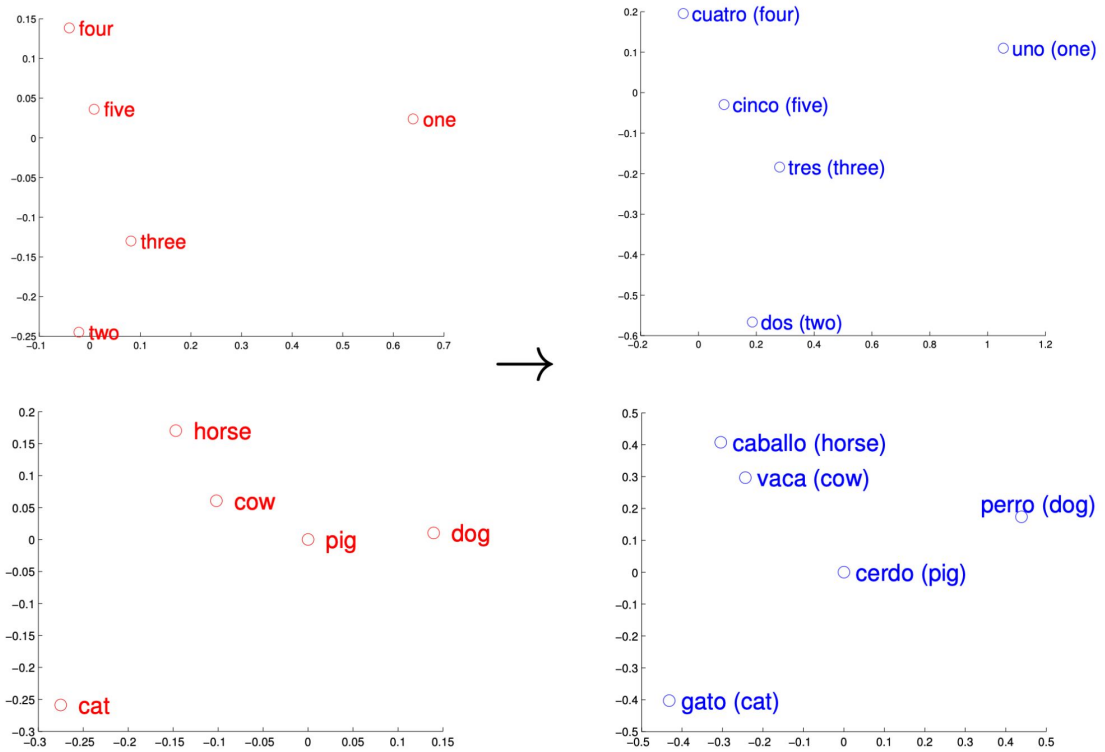
Word analogies & vector arithmetic





$$v_{\text{man}} - v_{\text{woman}} \approx v_{\text{king}} - v_{\text{queen}}$$
$$v_{\text{Paris}} - v_{\text{France}} \approx v_{\text{Rome}} - v_{\text{Italy}}$$

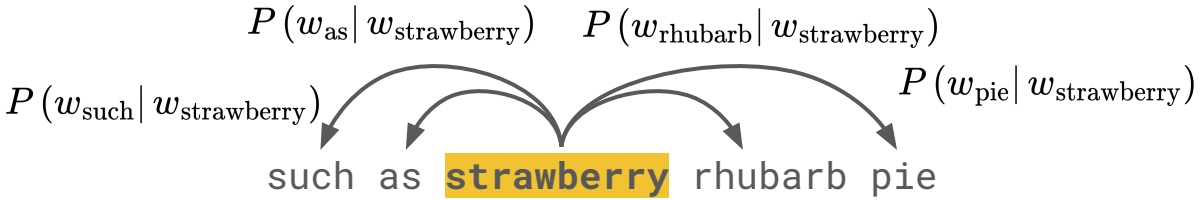
Other nice properties



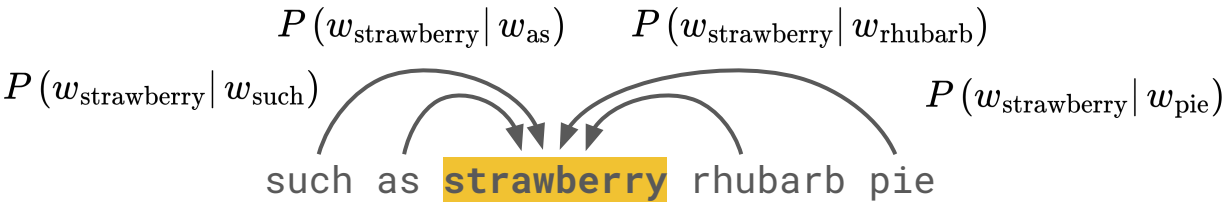
From [Mikolov et al. 2013]

word2vec

Skip-gram: predicts surrounding words given the current word



CBOW: predict the target word given its context words



Skip-gram word2vec intuition

Initialize word vectors as random vectors

For each target word *w* in a dataset and each of its context words *c*

Use the similarity of *w* and *c*'s word vectors to calculate $P(c \mid w)$

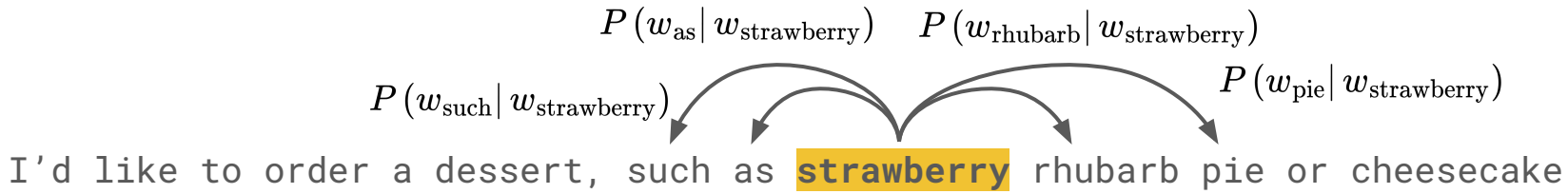
Keep adjusting word vectors, via gradient descent, to maximize these probabilities

Key idea: learn word vectors (a.k.a. model parameters θ) by reformulating semantics into a **classification problem**!

Self-supervision: turn unlabeled text into supervised training data (a.k.a. "pie" is the gold answer given "cherry"), without human labeling

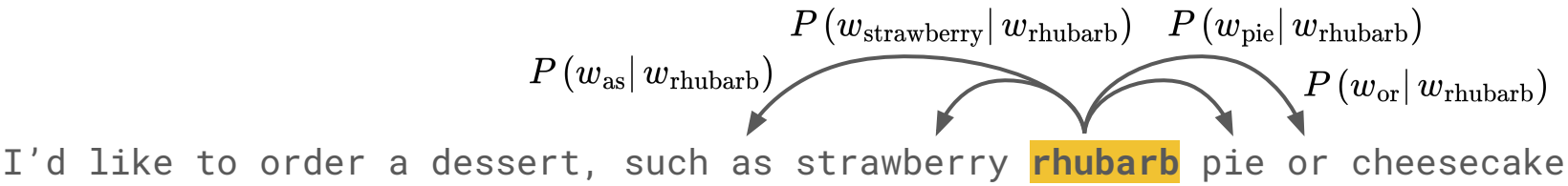
Skip-gram word2vec

Example, w/ a context window = 2



Skip-gram word2vec

Keep moving through the training data ...



Skip-gram word2vec

Goal: find parameters θ that can maximize the product of probabilities:

$$\ldots \times P\left(w_{\text{such}} \middle| w_{\text{strawberry}}\right) \times P\left(w_{\text{as}} \middle| w_{\text{strawberry}}\right) \times P\left(w_{\text{rhubarb}} \middle| w_{\text{strawberry}}\right) \times P\left(w_{\text{pie}} \middle| w_{\text{strawberry}}\right) \times$$
$$P\left(w_{\text{as}} \middle| w_{\text{rhubarb}}\right) \times P\left(w_{\text{strawberry}} \middle| w_{\text{rhubarb}}\right) \times P\left(w_{\text{pie}} \middle| w_{\text{rhubarb}}\right) \times P\left(w_{\text{or}} \middle| w_{\text{rhubarb}}\right) \times \ldots$$

$$= \prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P\left(w_{t+j} | w_t; \theta\right)$$

Word vectors!

For each position t in a text dataset of length T

for each word in the context window of size m around a word at position t

Compute the product of conditional probabilities

Skip-gram word2vec

Optimize θ to maximize data likelihood:

$$L(\theta) = \prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \ne 0} P(w_{t+j} | w_t; \theta)$$

This is equivalent to minimizing the average negative log likelihood, or our **objective function** (a.k.a. cost or loss function):

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \log P(w_{t+j} \mid w_t; \theta)$$

Minimizing this function = maximizing accuracy of predicting context words

Skip-gram word2vec

So how do we calculate each conditional probability?

Use two vectors per word *w*:

- v_w for when it's a context word
- u_w for when it's a target word

Dot product compares similarity of two vectors. Larger dot product = more similar. Exp makes value +

$$P\left(w_{t+j} \mid w_t\right) = \frac{\exp\left(u_{w_t} \cdot v_{w_{t+j}}\right)}{\sum_{k \in V} \exp\left(u_{w_t} \cdot v_k\right)}$$

Denominator normalizes things over entire vocabulary to produce a probability distribution.

Skip-gram word2vec

$$P\left(w_{t+j} \mid w_t\right) = \frac{\exp\left(u_{w_t} \cdot v_{w_{t+j}}\right)}{\sum_{k \in V} \exp\left(u_{w_t} \cdot v_k\right)}$$

It's an example of a softmax function!

$$\operatorname{softmax}\left(x_i\right) = \frac{\exp\left(x_i\right)}{\sum_{j=1}^{n} \exp\left(x_j\right)}$$

Softmax maps arbitrary values to probability distributions, where every mapped value is between 0 and 1 and when summed, adds up to 1.

Looks like multinomial logistic regression

Essentially a |*V*|-way classification problem

$$P\left(w_{t+j} \mid w_t\right) = \frac{\exp\left(u_{w_t} \cdot v_{w_{t+j}}\right)}{\sum_{k \in V} \exp\left(u_{w_t} \cdot v_k\right)} \qquad P\left(y = c \mid x\right) = \frac{\exp\left(w_c \cdot x + b_c\right)}{\sum_{j=1}^{m} \exp\left(w_j \cdot x + b_j\right)}$$

Some differences:

- We have to learn both *u* and *v* together, while on the right, *x* is fixed
  - This makes objective function non-convex (squiggly-shaped, harder to find global minimum)
- We don't actually care about the classifier or classification task, we only want the learned weights/parameters *θ*
  - This will be a recurring theme in pre-training models

Training word2vec

Make adjustments to parameters θ using stochastic gradient descent

Remember, θ includes two word vectors, u and v, for each and every word

We compute all vector gradients!

$$P(c \mid w) = \frac{\exp(u_w \cdot v_c)}{\sum_{k \in V} \exp(u_w \cdot v_k)}$$

$$u_w \leftarrow u_w - \eta \frac{\partial y}{\partial u_w}; \; \frac{\partial y}{\partial u_w} = -v_c + \sum_{k \in V} P(k \mid w) v_k$$

$$v_k \leftarrow v_k - \eta \frac{\partial y}{\partial v_k}, \forall k \in V; \; \frac{\partial y}{\partial v_k} = \begin{cases} (P(k \mid w) - 1) u_w & k = c \\ P(k \mid w) u_w & k \neq c \end{cases}$$

Problem: Every time you see a pair of *w* and *c* in the training data, you have to update *v* for every word in the vocabulary!

Skip-gram with negative sampling (SGNS)

Solution: instead of doing $|V|$-way classification, do **binary classification**

- **Previously**: Given a pair of words (w, c), let's encourage the model to predict c from w among all possible vocabulary words
- **Now**: Given a pair of words (w, c), where c may or may not be a true context word for w, let's classify whether c is a correct context word or not.

Train binary logistic regressions to differentiate a true pair (target word and context word) versus a sample of k "noise" pairs (target word paired with random word)

➔ Maximize similarity of true pairs, minimize similarity of noise pairs

Skip-gram with negative sampling (SGNS)

Similar to **binary logistic regression**

But like before, still need to optimize ***u*** and ***v*** together

$$L_{CE} = -\log \left(P(+ \mid w, c_{pos}) \prod_{i=1}^{k} P(- \mid w, c_{neg_i}) \right)$$

Skip-gram with negative sampling (SGNS)

Min of neg log likelihood

Probability that true pair is correct

Probability that *k* noise pairs are incorrect

$$L_{CE} = -\log \left(P(+ \mid w, c_{pos}) \prod_{i=1}^{k} P(- \mid w, c_{neg_i}) \right)$$

Skip-gram with negative sampling (SGNS)

Min of neg log likelihood

Probability that true pair is correct

Probability that *k* noise pairs are incorrect

Uses the sigmoid function from logistic regression instead of softmax, since binary instead of multi-class

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$L_{CE} = -\log\left(P(+\mid w, c_{pos}) \prod_{i=1}^{k} P(-\mid w, c_{neg_i})\right)$$

$$= -\log\sigma\left(u_w \cdot v_{c_{pos}}\right) - \sum_{i=1}^{k} \log\sigma\left(-u_w \cdot v_{c_{neg_i}}\right)$$

Less frequent words are sampled as “noise” context words more often.

Skip-gram w/ negative sampling summary

Start with $|V|$ random d-dimensional vectors as initial embeddings

Train a classifier based on embedding similarity

1. Take a dataset and take pairs of words that co-occur (+)
2. Take pairs of target word and some random words (-)
3. Train the classifier to distinguish these by slowly adjusting all the embeddings to improve classifier performance
4. Throw away the classifier and keep just the embeddings!

Since each word gets two vectors (one as target and one as context), can just add them together to get the final word embedding.

GloVe

Take a co-occurrence matrix computed over training data: X

Approximate words' similarity using words' co-occurrence counts:

$$u_i \cdot v_j \approx \log X_{i,j}$$

Log makes very big counts more manageable; exponential diffs become linear

Objective function:

$$J(\theta) = \sum_{i,j \in V} f(X_{i,j}) \left(u_i \cdot v_j + b_i + \tilde{b}_j - \log X_{i,j} \right)$$

Weighting function that gives more importance to more common word pairs

Make vector dot product plus some bias as similar to co-occurrence as possible.

Advances over word2vec:
Trains faster
Scalable to large corpora
Better evaluation results

FastText

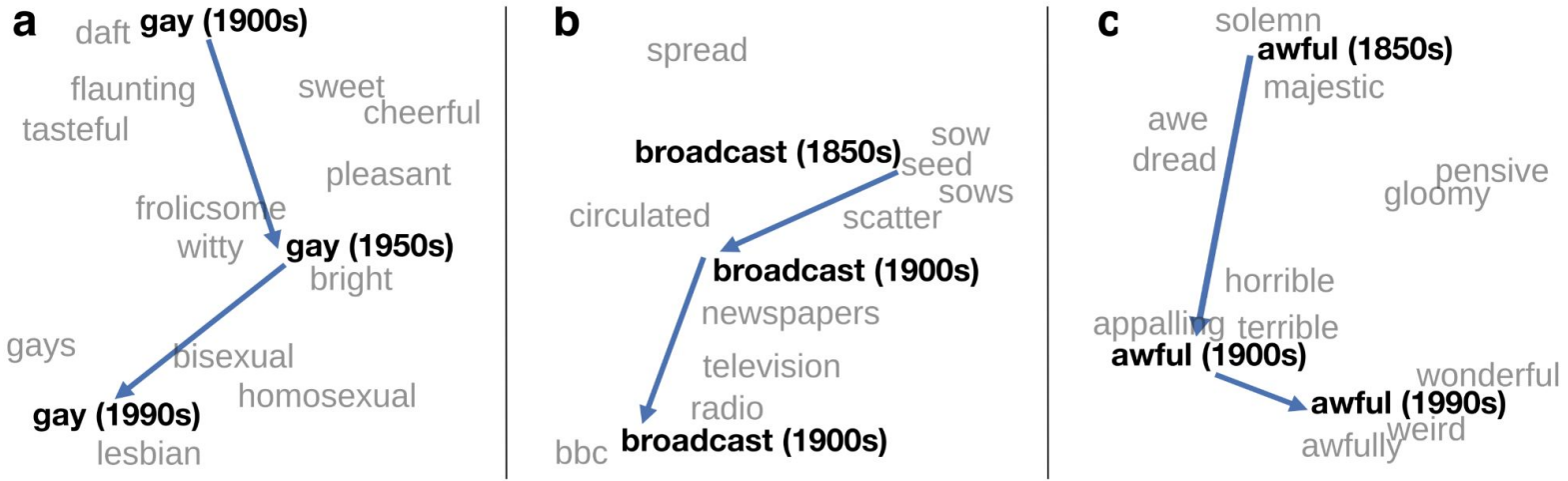
Similar to skip-gram, but break words into character *n*-grams with *n* = 3 to 6

```
3-grams: <wh, whe, her, ere, re>
4-grams: <whe, wher, here, ere>
5-grams: <wher, where, here>
6-grams: <where, where>
```

So, instead of calculating $u_i \cdot v_j$, calculate $\sum_{g \in n\text{-grams}(w_i)} u_g \cdot v_j$

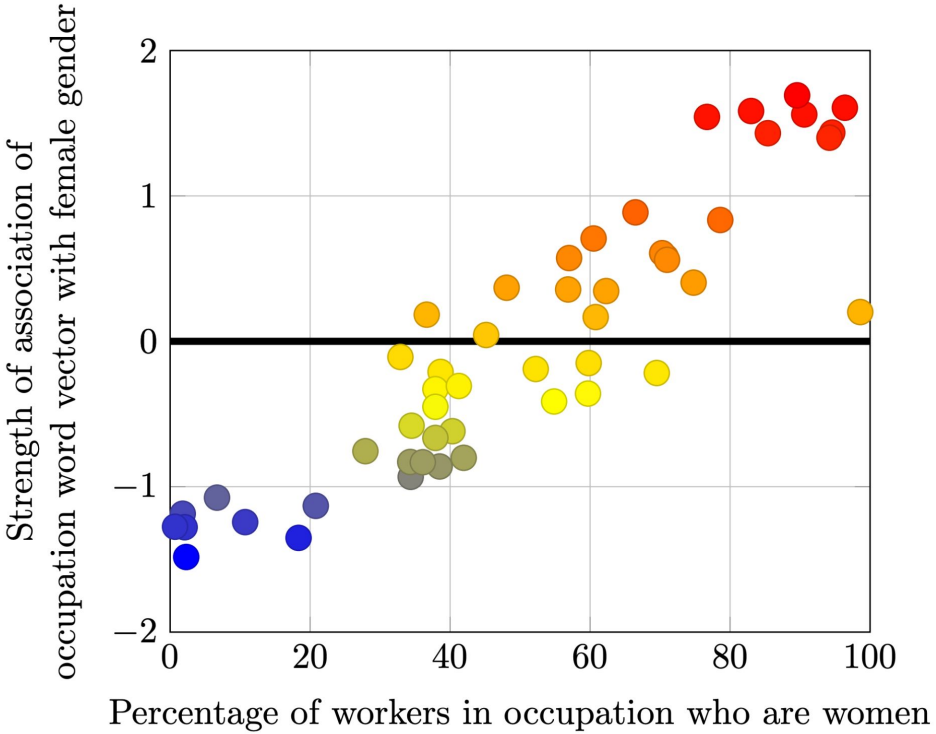
Better handling of rare words, out-of-vocab words, plurals, tenses, etc

Word embeddings for content analysis



From [Hamilton et al. 2016]

Word embeddings have biases



Cosine similarity of
`programmer, engineer, scientist, … nurse, teacher, librarian, …`

w/
`female, woman, girl, sister, she, her, hers, daughter`

From Caliskan et al. 2017

Pre-training data for word embeddings

Word2vec

- Google News corpus

GloVe

- Twitter
- Common Crawl (web data)
- Wikipedia and Gigaword (news)

Fasttext

- Common Crawl & Wikipedia

Underlying assumption: these datasets are "general-purpose" enough

Embeddings can be domain-adapted

Mittens: GloVe *with a warm start* (Dingwall & Potts 2018)

$$J(\theta) = \sum_{i,j \in V} f(X_{i,j}) \left(u_i \cdot v_j + b_i + \tilde{b}_j - \log X_{i,j} \right)$$

$$J_{\text{Mittens}} = J + \mu \sum_{i \in R} ||\hat{w}_i - r_i||^2$$

Controls the trade-off

GloVe object function, which favors changing representations based on specialized domain

Penalize distance of the learned embedding from an existing pre-trained, "general-purpose" one

Thematic takeaway: even though someone else can pre-train a model for us, sometimes we want to adapt our models to a specific domain or task

Evaluating word embeddings

Intrinsic evaluation

Evaluate on an intermediate subtask

Fast to compute ✔

Not clear if it really helps downstream tasks ✗

Intrinsic evaluation

Word similarity

Spearman rank correlation of embeddings' cosine similarity with human judgements of similarity, e.g. tiger & cat (7.35), stock & jaguar (0.92)

Word analogy test

Accuracy of predicting analogy $a : a^* :: b : b^*$

$$b^* = \arg\max_{w \in V} \left(\cos\left(e\left(w\right), e\left(a^*\right) - e\left(a\right) + e\left(b\right)\right)\right)$$

Chicago : Illinois :: Philadelphia : ??

bad : worst :: cool : ??

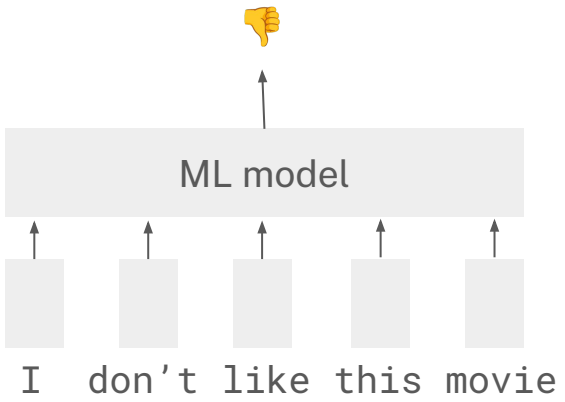
Evaluating word embeddings

Extrinsic evaluation

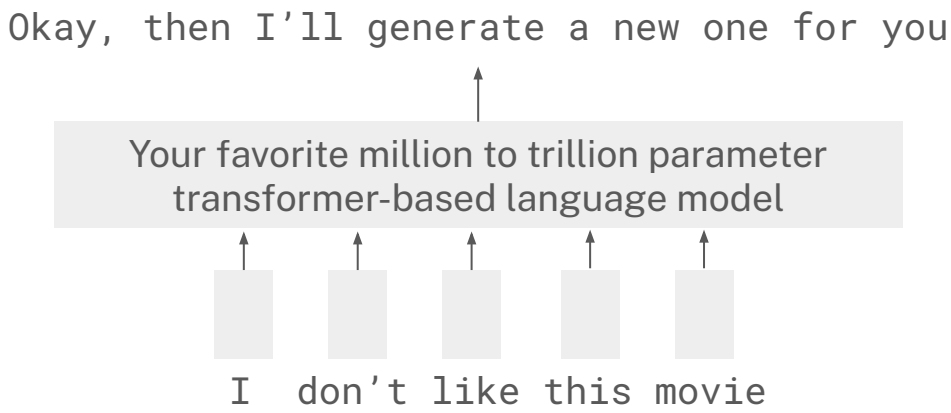
Plug word embeddings into a real NLP system and see whether they improve performance

Could take a long time ✗

The most important evaluation metric ✔



Where are embeddings today?



Inside every large language model!

They're just learned end-to-end as part of the model rather than trained separately – we'll talk more about this in another lecture

Foreshadowing

I bought a new computer **mouse** so I can play Valorant.

My cat is unable to catch the **mouse** living in my kitchen.

A single word embedding doesn’t do justice to this!

In a later week, we’ll talk about **contextualized** word embeddings.

Foreshadowing

I bought a new computer **mouse** so I can play Valorant.

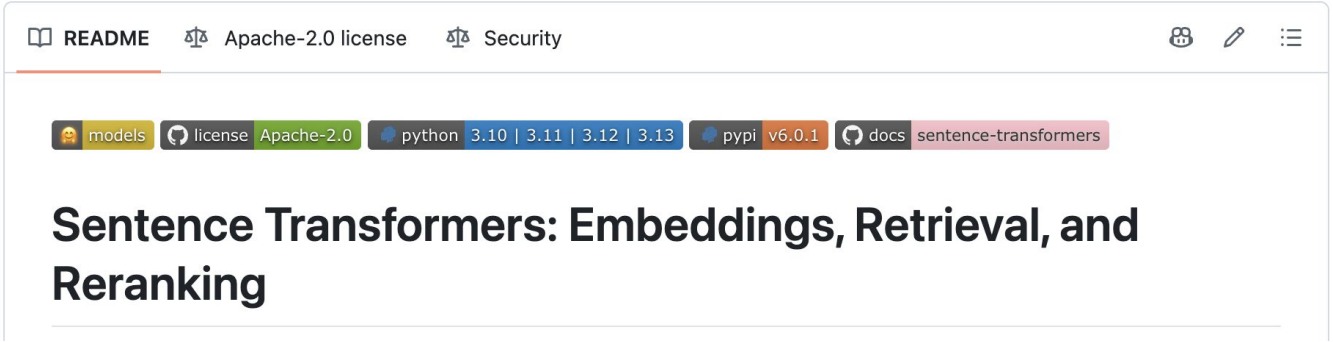
My cat is unable to catch the **mouse** living in my kitchen.

A single word embeddin

In a later week, we'll talk

Can also learn document-level representations!

Sentence embeddings



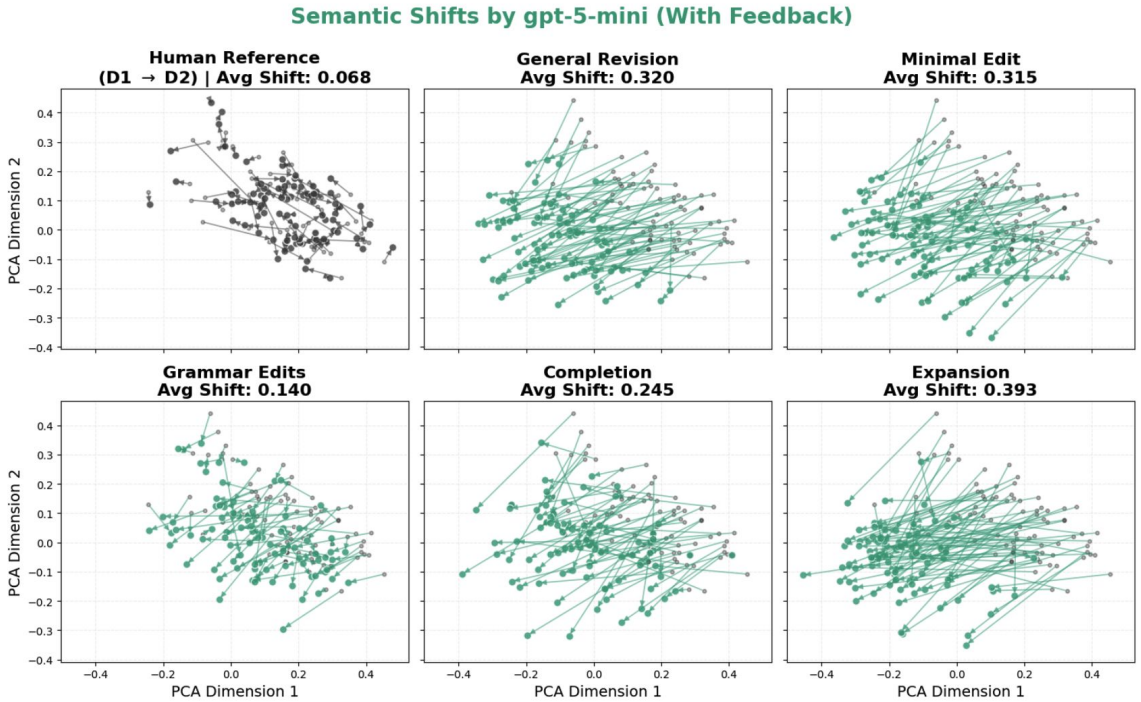
$$\max\left(||s_a - s_p|| - ||s_a - s_n|| + \epsilon,\ 0\right)$$

Make sentence *a* closer to sentence *p* in same section of doc

Push sentence *a* away from sentence *n* in diff section of doc

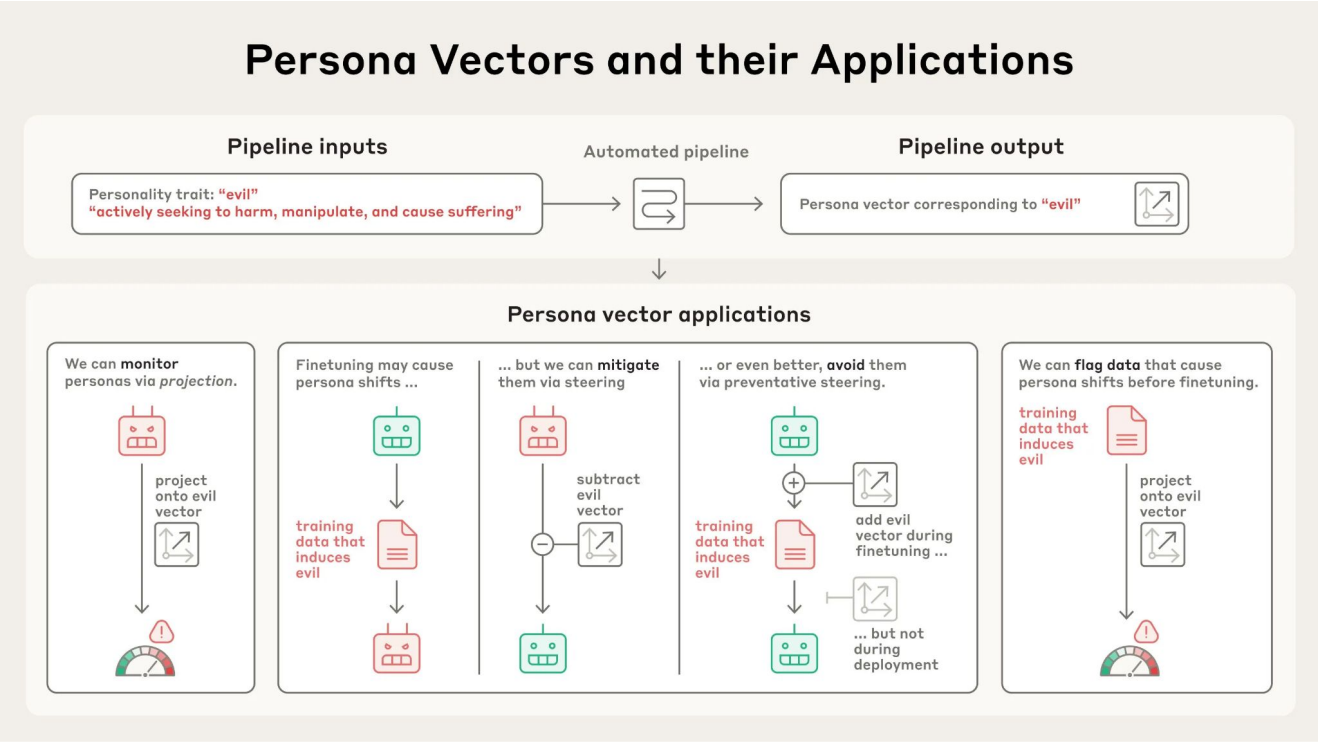
Margin ensures sentence *p* is at least ϵ closer to *a* than *n*

Uses of document representations



From [Abdulhai et al. 2026]

Concept representations



From Anthropic's blog post

Other uses for embeddings

Semantic search and retrieval. Convert docs and queries into vectors, so systems can find relevant passages by similarity. We have a retrieval lecture later!

Recommendation and personalization. Products, songs, articles, and users can be embedded in shared vector spaces.

Tokenization

How many words?

I’m

Orthographically one word

But what if we’re working with Thai, Chinese, or Japanese? They don’t have spaces like English does …

Grammatically two words:

- The subject pronoun *I*
- The verb *‘m*, short for *am*

Morphology

Words are structured in *morphemes*, the smallest meaningful unit

Example: walking → walk -ing

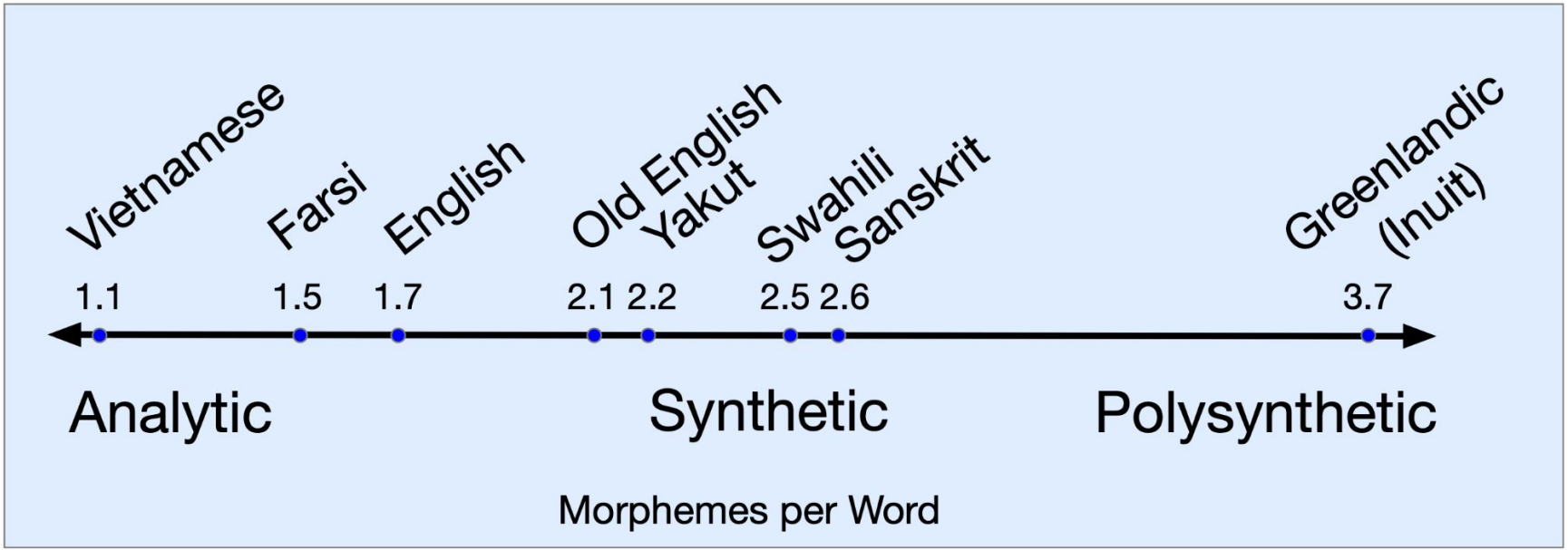
Example: unlockable → un- lock -able

Example: 电脑 (computer) → 电 脑 (electric brain)

Note: *unlockable* is funky one; it could mean unable to be locked, or un(lockable), or able to be unlocked, or (unlock)able.

Lots of edge cases of what a “morpheme” is across languages – hard to define!

Why morphology matters for tokenization



From SLP textbook

Why morphology matters for tokenization

How easily can we segment text into subwords?

Agglutinative languages, e.g. Turkish

- Very clean boundaries between morphemes

Fusion languages, where a single affix may conflate multiple morphemes

- Russian: -om in *stolom*.
- English: –s in "She reads the article" means both "third person" and "present tense"

These are tendencies rather than absolutes

We have too many words

Type: an element of the vocabulary *V*

- The number of types = vocabulary size |*V*|

Instance: an instance of that type in running text, e.g. *My cat saw the other cat* has two instances of *cat*.

Heap's law: Vocab size goes up with the square root of the number of instances *N*: $|V| = kN^{\beta}$

	Number of word types
Shakespeare	31 thousand
Brown Corpus	38 thousand
Switchboard conversations	20 thousand
COCA	2 million
Google n-grams	13+ million

Oh no, unks

If we have a fixed vocab of words, built from a training set, then all *novel* words seen at test time get mapped to a single UNK.

word		**vocab mapping**
tasty	→	tasty
learn	→	learn
taaaaaasty	→	UNK
laern	→	UNK
transformerify	→	UNK

Desiderata

We want **a method** that can map text to a sequence of discrete *tokens* from a vocabulary.

We want a vocabulary *V* that is

- **Expressive**: represent any text (English, Japanese, code, ...)
- **Efficient**
  - **Not too large:** larger vocab means more parameters to learn and store
  - **Not too small**: smaller vocab means longer inputs

Naive idea: UTF-8

Tokenize text as UTF-8 bytes

```
>>> utf = "我正在讲课。 👩‍🏫".encode('utf-8')
>>> print([x for x in utf])

[230, 136, 145, 230, 173, 163, 229, 156, 168, 232, 174, 178, 232, 175, 190, 227,
128, 130, 240, 159, 145, 169, 240, 159, 143, 187, 226, 128, 141, 240, 159, 143, 171]
```

Expressive ✓: UTF-8 bytes has 150k characters, can support 168 scripts (incl. dead languages like Sumerian cuneiform, conlangs like Klingon)

Sequences are too long ✗: inefficient

No inherent meaning ✗ in single characters

What could be better?

Split less common words into multiple **subword tokens**

The companies are expanding → `the compan _ies are expand _ing`

Benefits:

- Shared parameters between subwords
- Reduce parameter size, saving compute & memory

Since morphemes are hard to define, we'll **use the data to teach us** how to tokenize.

Byte Pair Encoding

Merge the most common neighboring token pairs into new tokens

- Start with a base vocabulary (e.g. UTF-8) and a training set
- Repeat:
  - Find the token pair that occurs most often, e.g. (‘e d’)
  - Introduce a new token (‘ed’)
  - Replace each instance of the token pair with the new token
- Stop when a desired vocab size or number of iterations is reached.

Note: usually BPE is run within space-separated words so that merges can’t happen across word boundaries.

Byte Pair Encoding

Vocabulary

[A, B, C, D, E]

Corpus / dataset

A B D C A B E C A B

Byte Pair Encoding

Vocabulary

[A, B, C, D, E]

Corpus / dataset

A B D C **A B** E C **A B**

Byte Pair Encoding

Vocabulary

[A, B, C, D, E, AB]

Corpus / dataset

AB D C AB E C AB

Byte Pair Encoding

Vocabulary

[A, B, C, D, E, AB]

Corpus / dataset

AB D **C** **AB** E **C** **AB**

Byte Pair Encoding

Vocabulary

[A, B, C, D, E, AB, CAB]

Corpus / dataset

AB D CAB E CAB

Practical tools

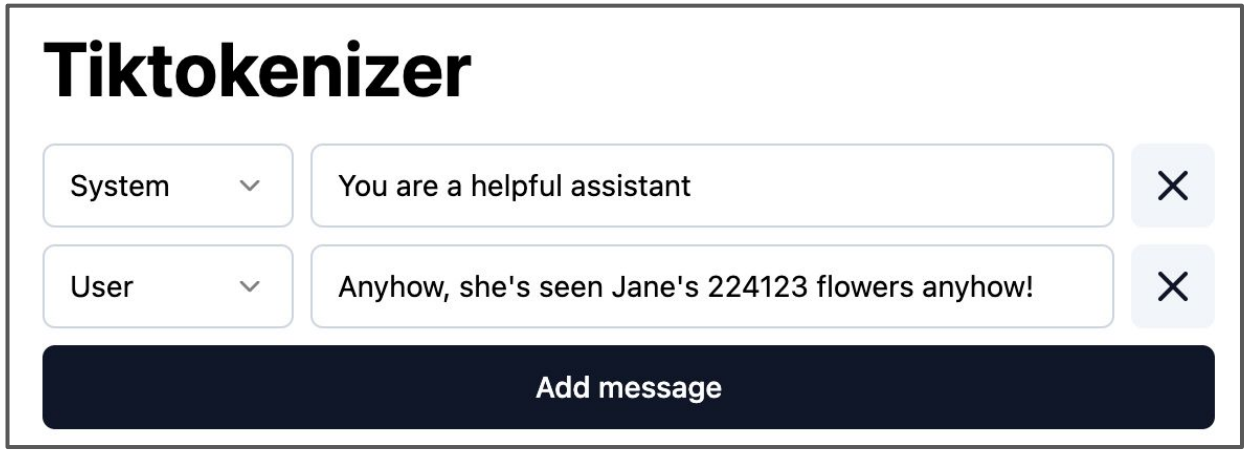
tiktoken: Load pre-existing OpenAI vocabularies (e.g. GPT-2, GPT-4), and tokenize and decode text:

```
>>> import tiktoken
>>> enc = tiktoken.get_encoding("gpt2")
>>> print(enc.encode("This is unbreakable glass."))
[1212, 318, 555, 9032, 540, 5405, 13]
>>> enc.decode([1212, 318, 555, 9032, 540, 5405, 13])
'This is unbreakable glass.'
>>> enc.decode([555])
' un'
```

SentencePiece: offers support for *training* tokenizers

Tiktokenizer visualizer

https://tiktokenizer.vercel.app/



Tiktokenizer visualizer

Token count
29

<|im_start|>system<|im_sep|>You are a helpful assistant<|im_end|><|im_start|>user<|im_sep|>Anyhow, she's seen Jane's 224123 flowers anyhow!<|im_end|><|im_start|>assistant<|im_sep|>

200264, 17360, 200266, 3575, 553, 261, 10297, 29186, 200265, 200264, 1428, 200266, 11865, 8923, 11, 31211, 6177, 23919, 885, 220, 19427, 7633, 18887, 147065, 0, 200265, 200264, 173781, 200266

Observations

Most words are tokens, w/ initial space

Clitics like ‘s are segmented off Jane, but part of frequent words like she’s.

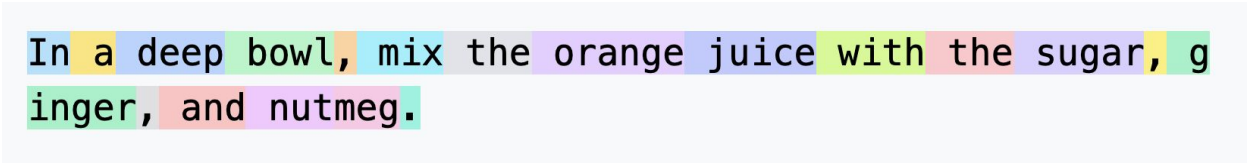
Numbers segmented into chunks (from preprocessing)

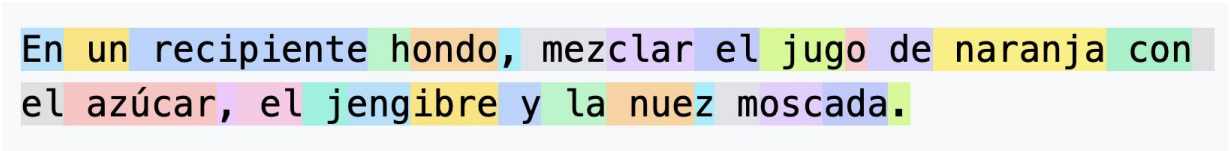
Anyhow and _anyhow are segmented differently

Tokenizing across languages

Even though BPE tokenizers are multilingual, training data is mostly English

- ➔ Most BPE tokens are used up by English, leaving less for other languages!
- ➔ Words in other languages are often split up
- ➔ Mitigation strategy: upsample underrepresented languages during training



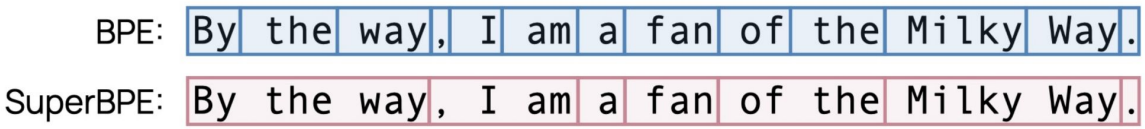


LM performance begins with tokenization

Better tokenization → better model performance

Example: SuperBPE (Liu et al. 2025)

- Start with regular BPE, then switch to discovering common word sequences and forming “superword” tokens
- +4.0% improvement over BPE in task performance
- Requires 27% less compute at inference time



Tokenization matters for $$

Pricing

Flagship models

Our latest models

Prices per 1M tokens.

Model	Short context Input	Cached input	Cache writes	Output
gpt-5.6-sol	$4.00	$0.40	$5.00	$20.00
gpt-5.6-terra	$2.00	$0.20	$2.50	$12.00
gpt-5.6-luna	$0.20	$0.02	$0.25	$1.20

Tokenization matters for $$

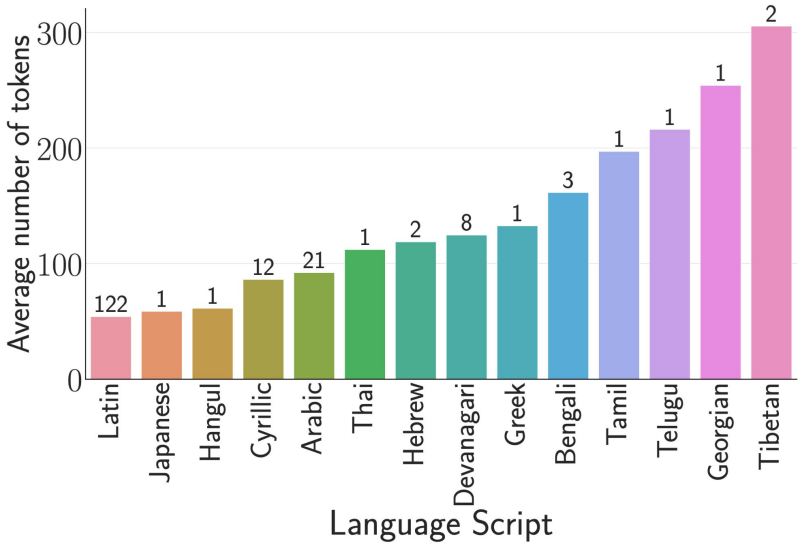
```
In a deep bowl, mix the orange juice with the sugar, g
inger, and nutmeg.
```

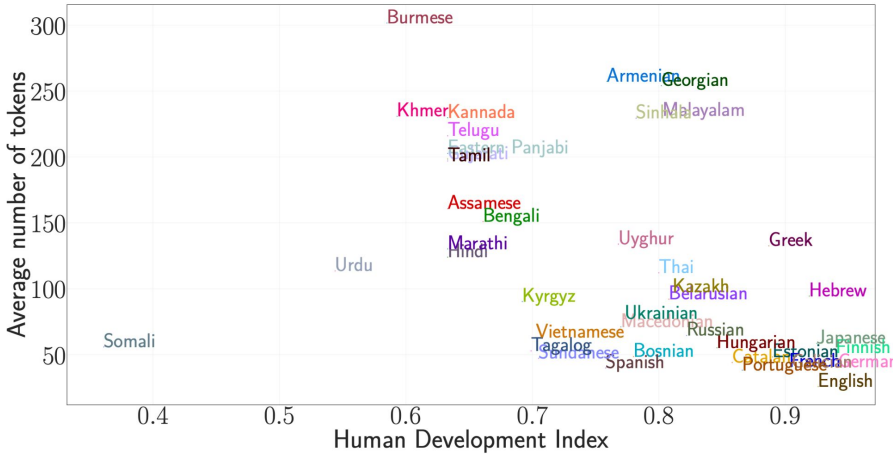
19 tokens

```
En un recipiente hondo, mezclar el jugo de naranja con
el azúcar, el jengibre y la nuez moscada.
```

29 tokens

Tokenization matters for $$

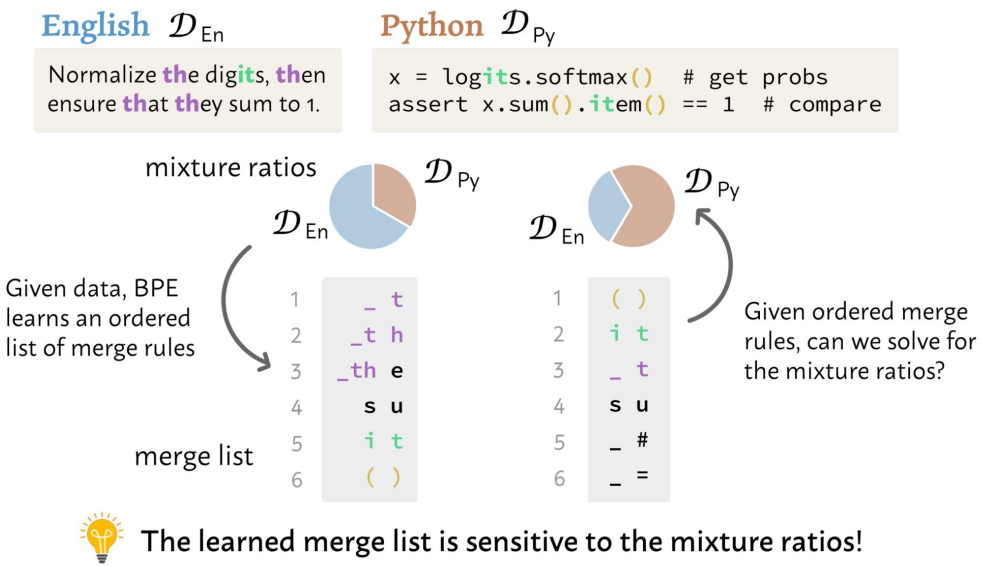




From [Ahia et al. 2023]

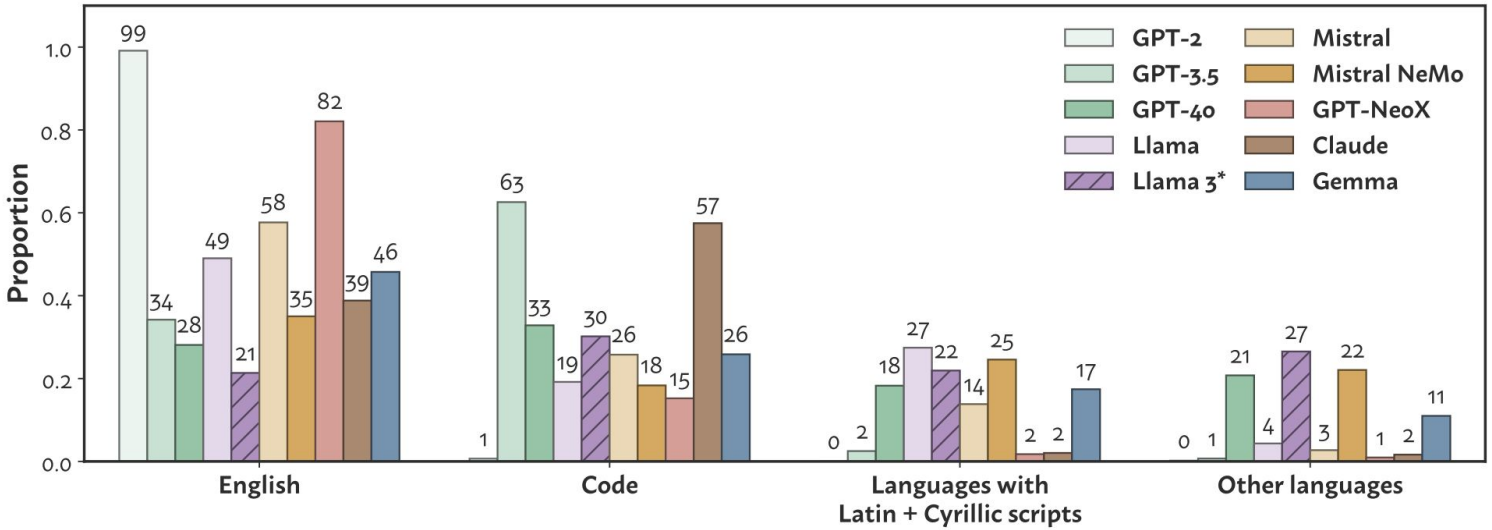
Tokenization can uncover secrets

The ordered list of merge rules learned by a BPE tokenizer can be used to infer training data mixtures (Hayase et al. 2024).



Tokenization can uncover secrets

The ordered list of merge rules learned by a BPE tokenizer can be used to infer training data mixtures (Hayase et al. 2024).



Fill out this sheet

If you're looking for project group members

Groups due on Canvas end-of-day Sept 14.

If you do not form a group on canvas by then, I will make groups for you.

The more team members, the more Google Cloud credits you have

https://bit.ly/4gTAMRx