Introduction to NLP

Vector and matrix models

Compressed out of NLP courses from **Dan Jurafsky** (Stanford), & David Bamman (Berkeley), Michael Collins (MIT & Columbia), and some online (Udemy) courses

Book: **Speech and Language Processing** by Jurafsky & Martin (3rd edition)

What do words **mean**?

- N-gram or text classification methods we've seen so far
  - Words are just strings (or indices w_i in a vocabulary list)
  - That's not very satisfactory!
- Formal logic classes:
  - $\forall x$ dog(X) ⟶ mammal(X)
  - $\forall x$ cat(X) ?
  - But again, just atomic symbols
- What should a good representation of word meaning do for us?
- Let's look at some desiderata from lexical semantics
  - the linguistic study of word meaning

Relations between words: **Synonymy**

- Synonyms have the same meaning in some or all contexts.
  - couch / sofa
  - big / large
  - automobile / car

- 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.

- Example: **big** vs. **large**
  - my big sister != my large sister

- …Difference in form → difference in meaning

Relation: **Similarity**

- No synonymy, but words can have **similar** meanings.
  - **car** vs. **bicycle**
  - **cow** vs. **horse**
- How to find out? Ask humans!

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)

Other word relations

Words can be related in a number of ways:

- Via a semantic frame (“topic”)
  - coffee, tea: **similar**
  - coffee, cup: **related** (not similar)
- Antonymy
  - dark - light
  - short - long
  - fast - slow

note that these are actually very similar!

- Connotation (sentiment)
  - great - love
  - terrible - hate

Sentiment

Words seem to vary along 3 affective dimensions:

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

	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

Values from NRC VAD Lexicon (Mohammad 2018)

Distributional semantics

- aka vector semantics

Computational models of word meaning

- **Vector (distributional) semantics**
  - The standard model in language processing!
  - Handles many of our linguistic goals!

- **Idea**: Words are defined by their environments (the words around them)
  - Wittgenstein: "*The meaning of a word is its use in the language*"
  - Firth (1957): "*You shall know a word by the company it keeps*"

- From the common notion of synonymy:
  - If A and B have almost identical environments, they are synonyms!

Example: What does *”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: *"Water Spinach"*

空心菜
kangkong
rau muống
…

Yamaguchi, Wikimedia Commons, public domain

Model of word meaning

- Idea 1: Let's define the meaning of a word by its distribution in language
  - meaning its neighboring words

- Idea 2: Meaning is a point in multidimensional space

example with connotation:

	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

Defining meaning as a point in space

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

- These vectors are commonly called "**embedding**"
  - because they are embedded into shared space
- The standard way to represent meaning in NLP
  - **Every modern NLP algorithm uses embeddings**

- Fine-grained model of meaning for **similarity**

- **This is in contrast to thesaurus/logic-based meaning where**
  - We don’t have a thesaurus for every language
  - Even if we do, they have problems with **recall**
    - Many words are missing
    - Most (if not all) phrases are missing
    - Some connections between senses are missing

Intuition: why vectors?

Consider sentiment analysis:

- With **words**, a feature is a word identity
  - Feature 5: 'The previous word was "terrible"'
  - requires the **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!!!

Words as vectors

- document & word matrices

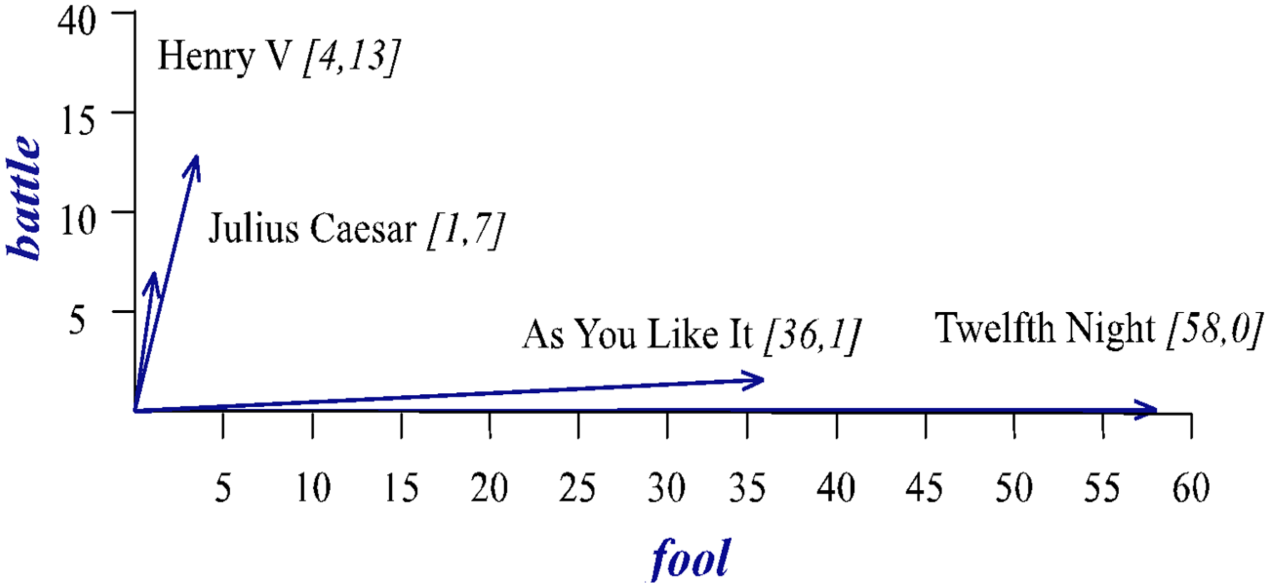
Term-document matrix

We already know that each **document** can be represented by a **count 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

- This is called the **term-document matrix**
  - This representation is fundamental in indexing and information retrieval

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
  - As You like It & Twelfth Night
- But comedies are different than the other two
  - Comedies have more *fools* and *wit* and fewer *battles*.

Words as rows in term-document matrix

- Similarly to documents, **words can be considered as 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*"

Term-context matrix

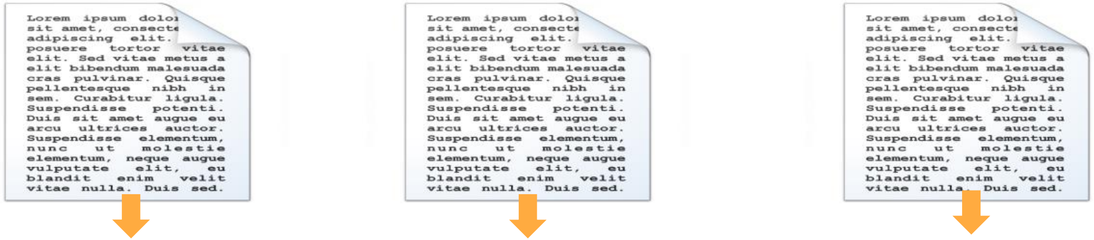
- We may now completely skip the documents and focus on the words
- This lead to the **term-context matrix**
  - or “**word-word**” matrix of size VxV
- *The words are similar in meaning if their **context vectors** are similar*

	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	...

remember bi-grams?

Word context creation

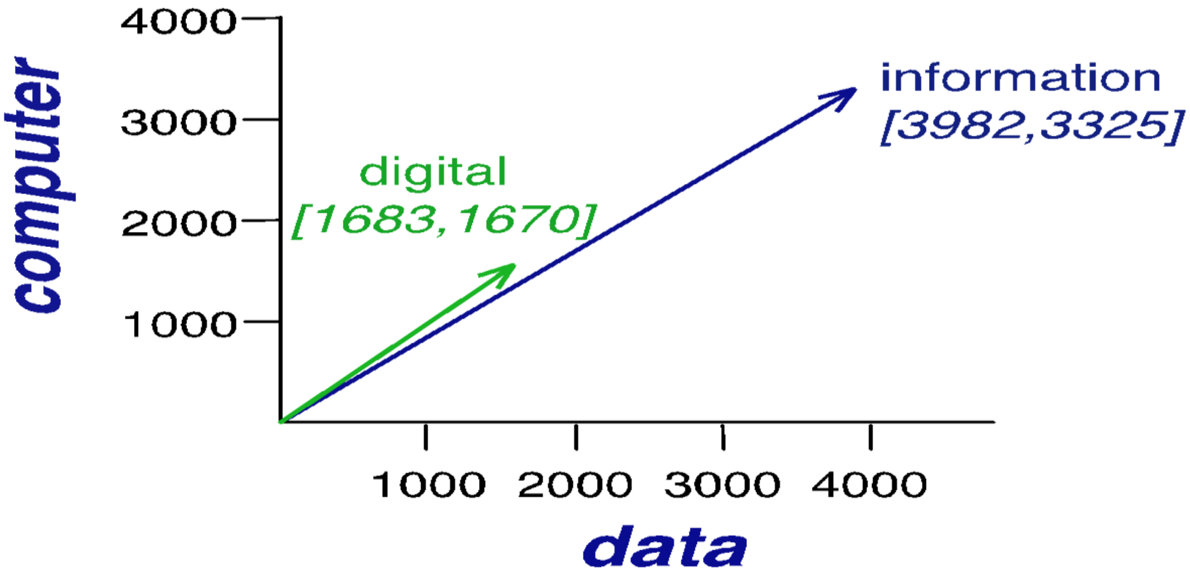
Instead of using entire documents, we can extract smaller **context windows:**



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

- The size of the context window depends on our goal
- The shorter the windows the more **syntactic** the representation (± 1-3 words)
- The longer the windows the more **semantic** the representation (± 4-10 words)

Visualizing **word** vectors



Word similarity

- cosine similarity

Computing word similarity: Dot product

Reminder: dot product between two vectors is a scalar:

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

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

Problem:

- Dot product favors long vectors
  - those that have higher values in many dimensions
- Frequent words will have generally longer vectors!
  - since they co-occur many times with other words
    - “of, and, the, you, …”

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

Alternative: cosine similarity

Solution: **normalize** by the length of the vectors…

= Cosine similarity

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

- by far the most popular similarity metric in NLP
- using the definition of the dot product between two vectors:

$$\mathbf{v}\cdot\mathbf{w} = |v||w|\ cos\ \theta \qquad \frac{\mathbf{v}\cdot\mathbf{w}}{|v||w|} = cos\ \theta$$

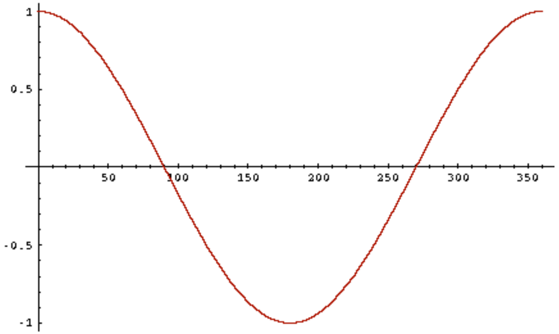
Cosine as a similarity metric

Generally:

-1: vectors point in **opposite** directions

+1: vectors point in **same** directions

0: vectors are **orthogonal**



With count vectors:

- The frequency values are non-negative
- Hence the cosine for term-term matrix vectors ranges from 0–1

Cosine examples

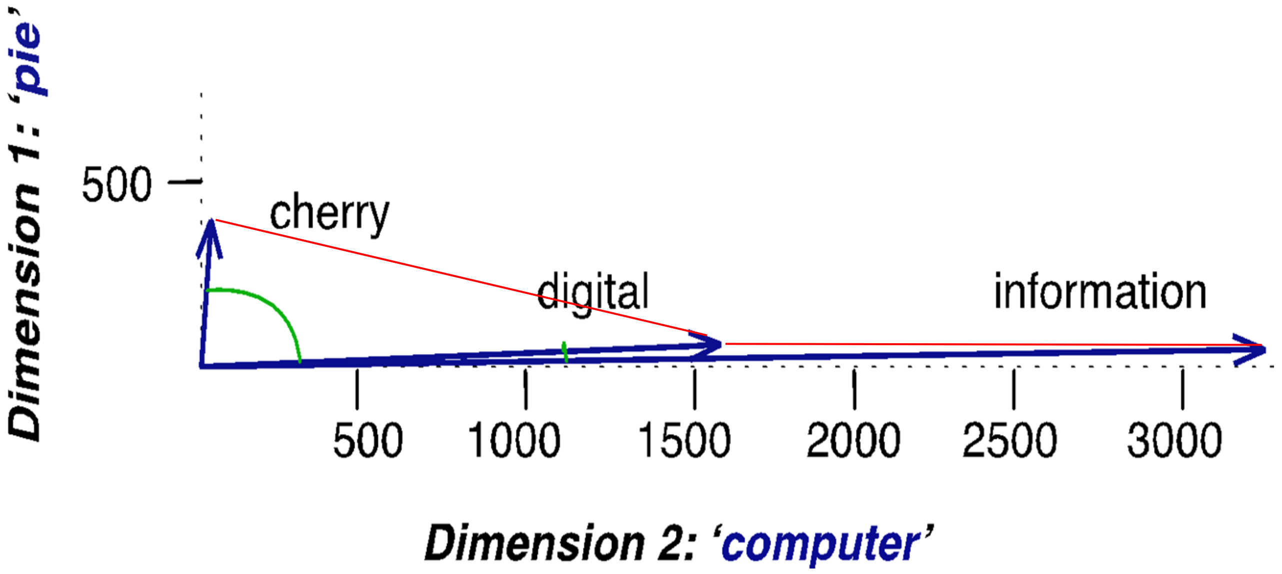
$$\text{cosine}(\vec{v},\vec{w}) = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

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

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

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

Visualizing cosines (angles)



Vector Semantics

- **TF-IDF** for **Term-Document** matrix weighting

Raw frequency is a bad representation

- The co-occurrence matrices we have seen represent raw **frequencies**.
- Frequency is clearly useful:
    - if *sugar* appears a lot near *apricot*, that's useful information.
- But overly frequent words are not very informative about the context
    - e.g., words like *the*, *it, and* or *they*
- It's a paradox! How can we balance these two conflicting constraints?

Two common solutions for word weighting

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

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

commonly used for weighting **document** dimensions of words

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

PMI: (Pointwise mutual information)

- $\text{PMI}(w_1, w_2) = log \frac{p(w_1,w_2)}{p(w_1)p(w_2)}$

commonly used for weighting **word** dimensions of words

- Statistical measure: see if words like "good" appear more often with "great" than we would expect by chance

TF-IDF for Term-Document matrix weighting

1) Term frequency (**tf**)

- **$tf_{t,d}$** = count(t,d)
- Instead of using raw count, we commonly squash a bit:
- $tf_{t,d} = \log_{10}(\text{count}(t,d)+1)$

2) Document frequency (**df**)

- **df_t** is the number of documents a term 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

TF-IDF for Term-Document matrix weighting

2') **Inverse** document frequency (**idf**)

- emphasize words that appear in **few** documents
- **idf_t** $= N / df_t$
- again, more commonly:

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

- where **N** is the total number of documents in the collection
- Note that documents can be **anything**
  - we often call each paragraph a document!

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

Final TF-IDF word weighting

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

Raw counts:

	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	0.074	0	0.22	0.28
good	0	0	0	0
fool	0.019	0.021	0.0036	0.0083
wit	0.049	0.044	0.018	0.022

Vector Semantics

- **Positive PMI** for **Term-Term** matrix weighting

Two common solutions for word weighting

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

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

commonly used for weighting **document** dimensions of words

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

PMI: (Pointwise mutual information)

- $\text{PMI}(w_1, w_2) = log \frac{p(w_1, w_2)}{p(w_1)p(w_2)}$

commonly used for weighting **word** dimensions of words

- Statistical measure: see if words like "good" appear more often with "great" than we would expect by chance

Pointwise Mutual Information

Pointwise mutual information:

Do events x and y co-occur more than if they were independent?

$$\text{PMI}(X,Y) = \log_2 \frac{P(x,y)}{P(x)P(y)}$$

PMI between two words: (Church & Hanks 1989)

Do words x and y co-occur more than if they were independent?

$$\text{PMI}(word_1, word_2) = \log_2 \frac{P(word_1, word_2)}{P(word_1)P(word_2)}$$

Positive Pointwise Mutual Information

PMI generally ranges from –inf to +inf

- Positive values mean w1 and w2 co-occur **more than** by chance
- Zero values mean w1 and w2 co-occur **exactly as if** by chance
- Negative values mean w1 and w2 co-occur **less than** by chance

In practice, we commonly care only about emphasizing the positive case

- Leading a modification called Positive PMI = **PPMI**

$$PPMI = \begin{cases} PMI & if\ PMI > 0 \\ 0 & else \end{cases}$$

Computing PPMI on a term-context matrix

- Matrix **F** (frequency)
  - with **W** rows (words) and **C** columns (contexts)
- f_{ij} is the number of times w_i occurs in context c_j

$$p_{ij} = \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \qquad p_{i*} = \frac{\sum_{j=1}^{C} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \qquad p_{*j} = \frac{\sum_{i=1}^{W} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

	computer	data	result	pie	sugar	count(w)
cherry	2	8	9	442	25	486
strawberry	0	0	1	60	19	80
digital	1670	1683	85	5	4	3447
information	3325	3982	378	5	13	7703
count(context)	4997	5673	473	512	61	11716

$$pmi_{ij} = \log_2 \frac{p_{ij}}{p_{i*}p_{*j}} \qquad ppmi_{ij} = \begin{cases} pmi_{ij} & \text{if } pmi_{ij} > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$p_{ij} = \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

	computer	data	result	pie	sugar	count(w)
cherry	2	8	9	442	25	486
strawberry	0	0	1	60	19	80
digital	1670	1683	85	5	4	3447
information	3325	3982	378	5	13	7703
count(context)	4997	5673	473	512	61	11716

p(w=**information**, c=**data**) = 3982/111716 = .3399

p(w=**information**) = 7703/11716 = .6575

p(c=**data**) = 5673/11716 = .4842

$$p(w_i) = \frac{\sum_{j=1}^{C} f_{ij}}{N} \qquad p(c_j) = \frac{\sum_{i=1}^{W} f_{ij}}{N}$$

	p(w,context)					p(w)
	computer	data	result	pie	sugar	p(w)
cherry	0.0002	0.0007	0.0008	0.0377	0.0021	0.0415
strawberry	0.0000	0.0000	0.0001	0.0051	0.0016	0.0068
digital	0.1425	0.1436	0.0073	0.0004	0.0003	0.2942
information	0.2838	0.3399	0.0323	0.0004	0.0011	0.6575
p(context)	0.4265	0.4842	0.0404	0.0437	0.0052	

$$pmi_{ij} = \log_2 \frac{p_{ij}}{p_{i*}p_{*j}}$$

	p(w,context)					p(w)
	computer	data	result	pie	sugar	p(w)
cherry	0.0002	0.0007	0.0008	0.0377	0.0021	0.0415
strawberry	0.0000	0.0000	0.0001	0.0051	0.0016	0.0068
digital	0.1425	0.1436	0.0073	0.0004	0.0003	0.2942
information	0.2838	0.3399	0.0323	0.0004	0.0011	0.6575
p(context)	0.4265	0.4842	0.0404	0.0437	0.0052	

pmi(information,data) = $\log_2$ (.3399 / (.6575*.4842)) = .0944

Resulting PPMI matrix (negatives replaced by 0)

	computer	data	result	pie	sugar
cherry	0	0	0	4.38	3.30
strawberry	0	0	0	4.10	5.51
digital	0.18	0.01	0	0	0
information	0.02	0.09	0.28	0	0

Technical note: Modifying PMI

Problem: PMI is biased toward infrequent events

- Very rare words have very high PMI values

Solution:

- Use add-one **smoothing**

Vector Semantics

- Dense vectors

Sparse versus dense vectors

- TF-IDF / PPMI vectors are
  - **long** (length |V|= 20,000 to 300,000)
  - **sparse** (most elements are zero)
- Alternative: learn vectors which are
  - **short** (length 50-1000)
  - **dense** (most elements are non-zero)

Short dense vectors (embeddings)

- Why dense vectors?
  - **They work better in practice!**
  - Short vectors may be easier to use as features in machine learning (less weights to tune)
  - Dense vectors may generalize better than storing explicit counts
  - They may do better at capturing synonymy
    - *car* and *automobile* are synonyms, but are represented as distinct dimensions

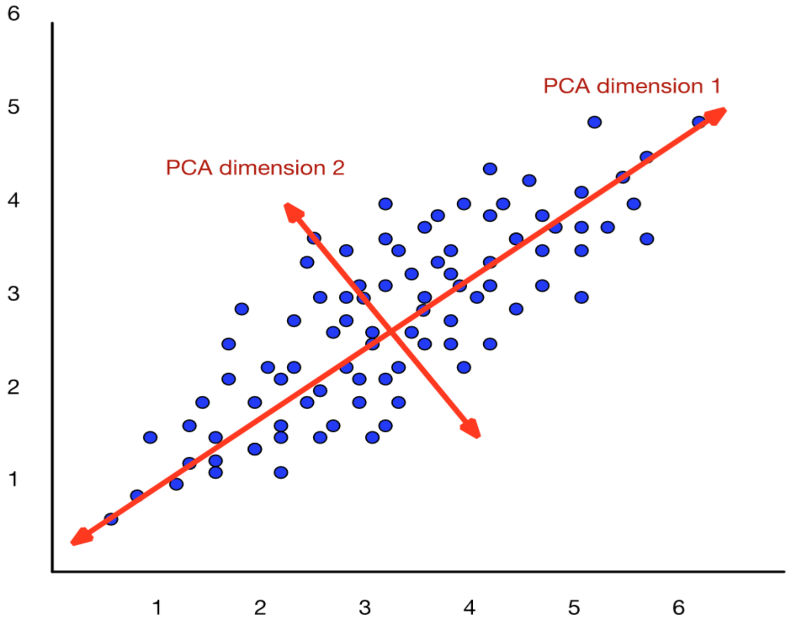
- How to obtain them?

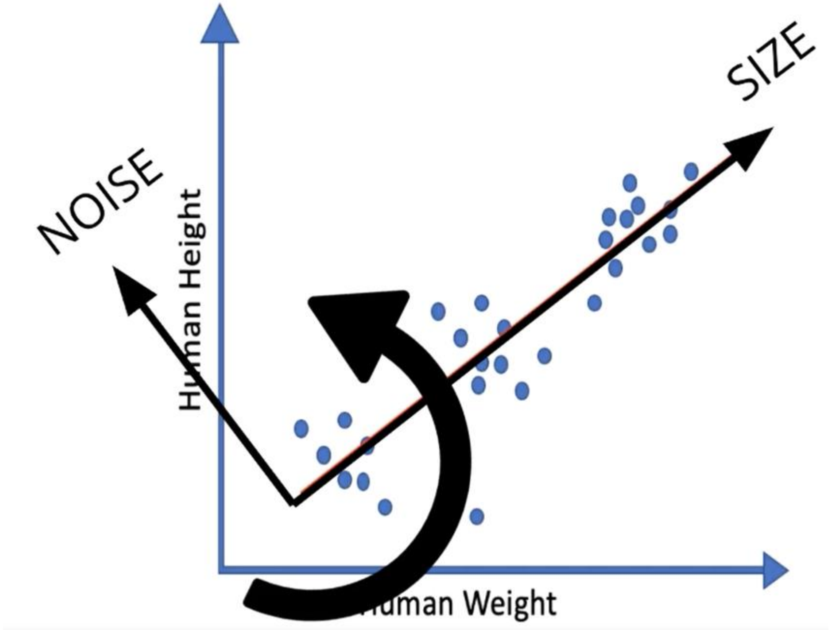
1. Matrix factorization
   - LSA (SVD), NNMF
2. “Neural” Models
   - word2vec, GloVe

Vector Semantics

- Dense vectors via **SVD** : Term-Document matrix

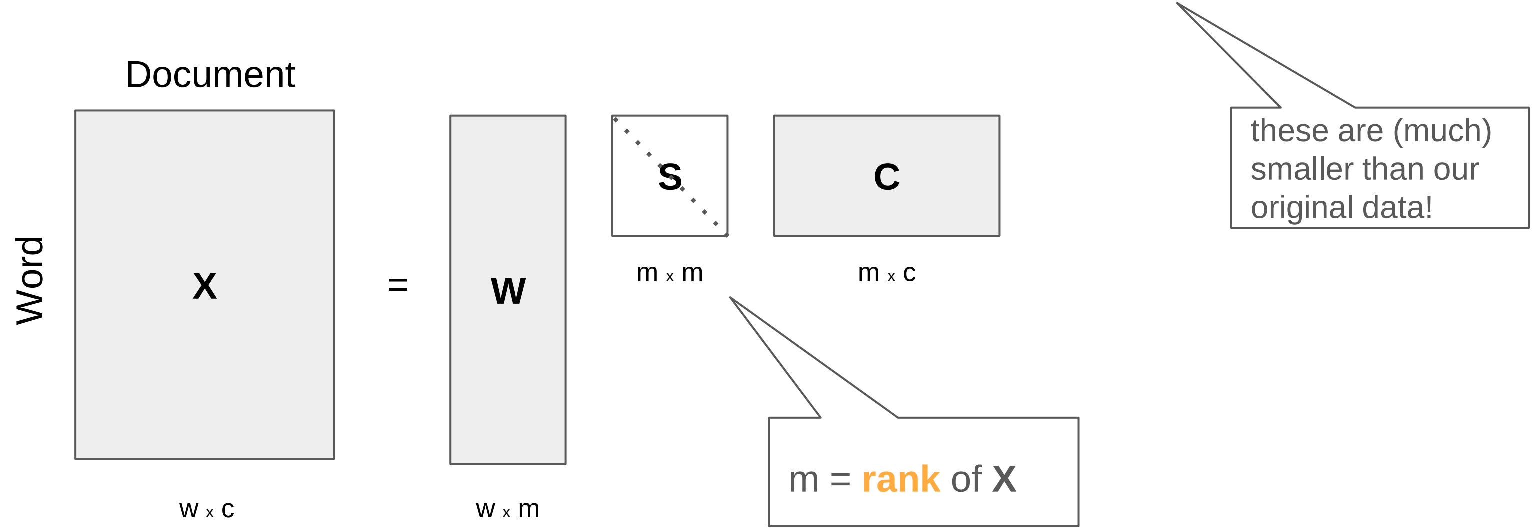
Dimensionality reduction





Singular Value Decomposition (SVD)

Any w x c matrix X equals the product of 3 matrices: **X = W S C**



- note that rank(**X**) ≦ min(w,c)
- reveals the "true" dimensionality of our data

Singular Value Decomposition

Any w x c matrix X equals the product of 3 matrices: **X = W S C**

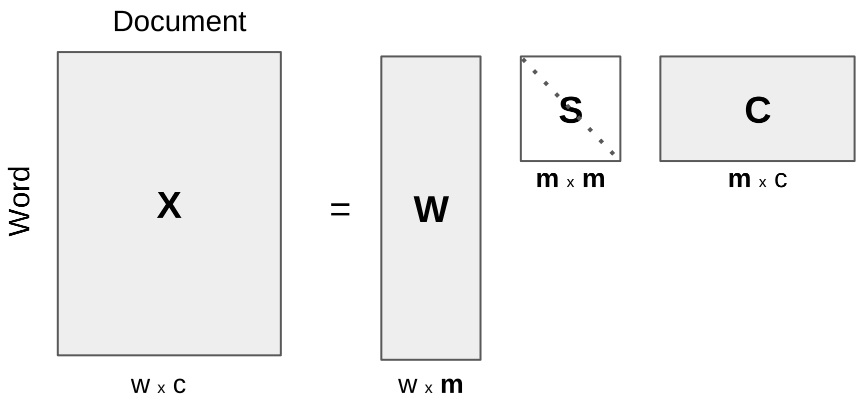
- **W** (w x **m**): rows corresponding to original, but **m** columns represents a dimension in a new latent space, such that
  - **m** column vectors are orthogonal to each other
  - Columns are ordered by the amount of variance in the dataset each new dimension explains

- **S** (**m** x **m**): **diagonal** *m* x *m* matrix of **singular values** expressing the importance of each dimension.

- **C** (**m** x c): columns corresponding to original, but m rows corresponding to the singular values

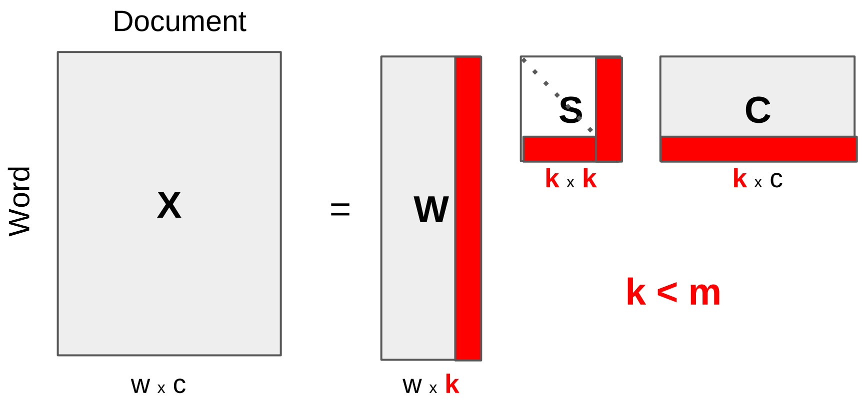
Truncated Singular Value Decomposition

- Often, **m** is not small enough
- Instead of keeping all m dimensions, we just keep the **top k** singular values.
  - Let’s say 300.
- The result is a least-squares approximation to the original **X**
- Each row of W is:
  - k-dimensional vector
  - Representing word W



Truncated Singular Value Decomposition

- Often, **m** is not small enough
- Instead of keeping all m dimensions, we just keep the **top k** singular values.
  - Let's say 300.
- The result is a least-squares approximation to the original **X**
- Each row of W is:
  - k-dimensional vector
  - Representing word W



Vector Semantics

- Revisiting topic modelling

Latent Semantic Analysis

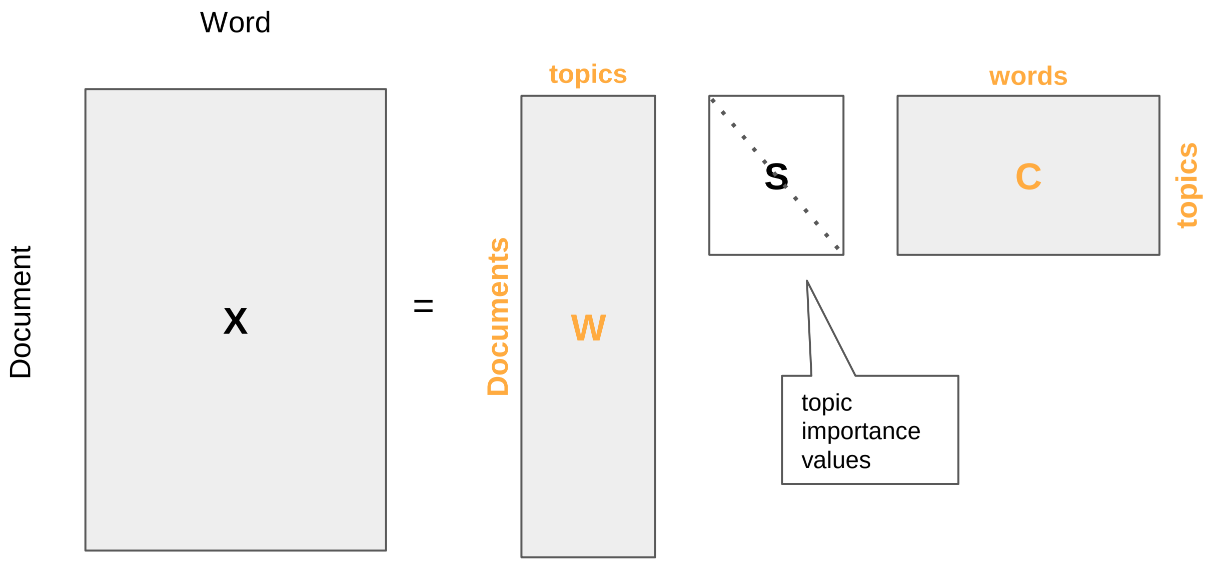
- LSA is often referred to as “topic modelling” itself

- SVD applied to the Document-Term matrix = **Latent Semantic Analysis**
    - 300 dimensions are commonly used for **k**
- The cells are commonly weighted by **TF-IDF**

- **k** topics = **k** latent dimensions
- we expect the word distr. across the topics to be distinct/orthogonal
    - this is exactly what SVD does!

> "*SVD is not nearly as famous as it should be.*" - Gilbert Strang

Topic modelling with LSA



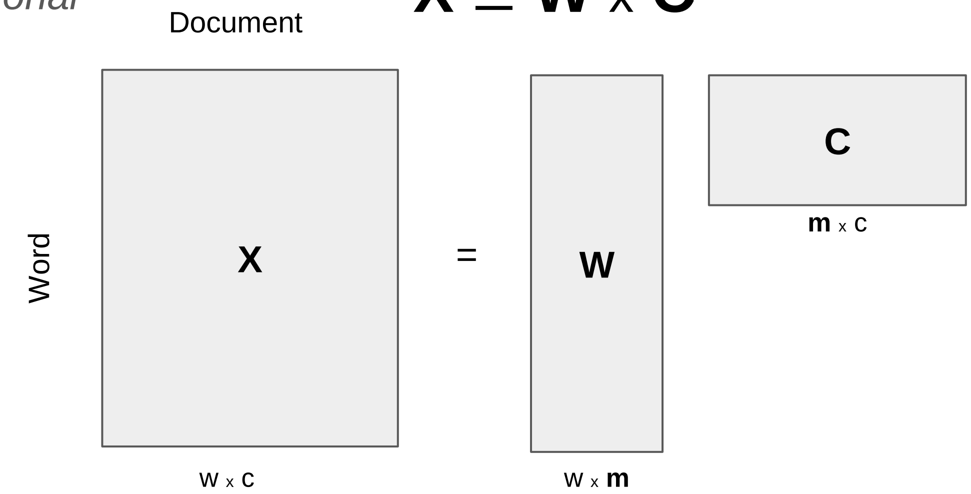
= the same output format as we have seen from LDA!

Non-negative Matrix Factorization

Alternative decomposition: **Non-negative Matrix Factorization** (NNMF)

- Idea: constrain the latent topics to be ***non-negative***
  - rather than constraining to be *orthogonal*
- This is easier to **interpret** the topics
  - W ~ amount of words in topics
  - C ~ amount of topics in documents

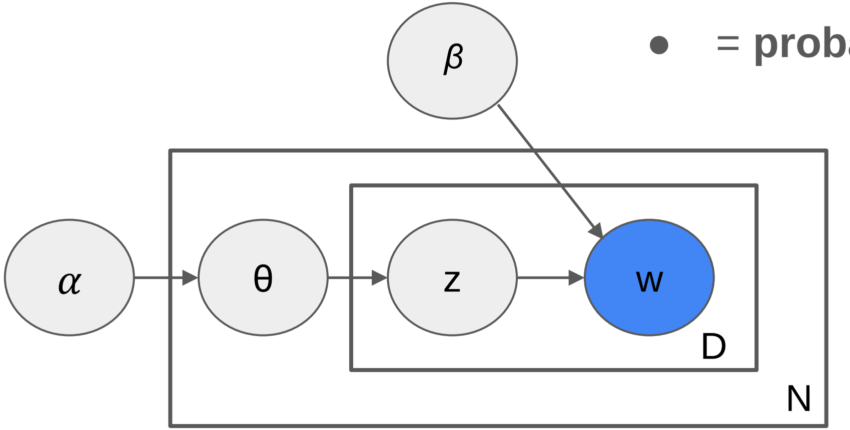
$$\mathbf{X} \cong \mathbf{W} \times \mathbf{C}$$



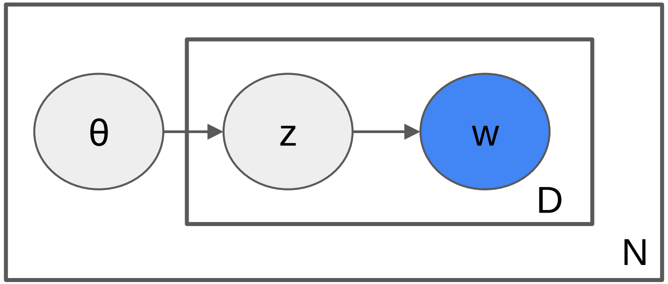
Non-negative Matrix Factorization

NNMF is only approximate

- different optimization criteria for the $\mathbf{X} \cong \mathbf{W}_{x}\mathbf{C}$ problem
- with Kullback-Leibler divergence $KL(X; W_{x}C)$
    - = **probabilistic LSA (pLSA)**



LDA = Bayesian



pLSA = not Bayesian

Vector Semantics

- Dense vectors via **SVD: Term-Term** matrix

SVD applied to Term-Term matrix

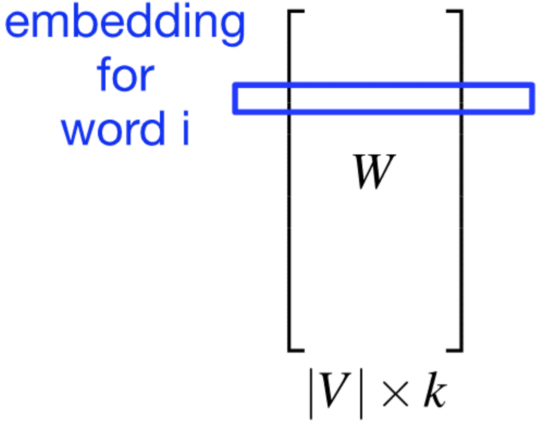
- …let's return to the PPMI **Term-Term** matrices
  - can we apply SVD to them?

$$\underset{|V| \times |V|}{\begin{bmatrix} \\ X \\ \\ \end{bmatrix}} = \underset{|V| \times |V|}{\begin{bmatrix} \\ W \\ \\ \end{bmatrix}} \underset{|V| \times |V|}{\begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_V \end{bmatrix}} \underset{|V| \times |V|}{\begin{bmatrix} \\ C \\ \\ \end{bmatrix}}$$

- simplifying assumption: the matrix has rank |V|

Truncated SVD produces **embeddings**

$$\underset{|V|\times|V|}{\begin{bmatrix} \\ X \\ \\ \end{bmatrix}} = \underset{|V|\times k}{\begin{bmatrix} \\ W \\ \\ \end{bmatrix}} \underset{k\times k}{\begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_k \end{bmatrix}} \underset{k\times|V|}{\begin{bmatrix} C \end{bmatrix}}$$



- Dense SVD embeddings vs. sparse PPMI matrices
- generally better at tasks requiring **word similarity**
  - Denoising: low-order dimensions represent noise
  - Truncation may help the models generalize better to unseen data.

Problems with SVD

Problems with SVD

- Computational cost scales quadratically, for w×c matrix:

  O**(wc²)** operations (when c<w)

  - → Bad for millions of words or documents!

- Hard to incorporate new words or documents
- Different learning regime than common ML models

Vector Semantics

- Dense embedding vectors via machine learning

Embeddings: Prediction-based models

Main idea: instead of capturing co-occurrence **counts**, **predict** the words in text

- Importantly, this is **self-supervised learning**
  - A word **c** that occurs near input word **w** in the corpus is the "correct label"
    - No need for human labels!
  - Inspired by neural net language models
    - Bengio et al. (2003); Collobert et al. (2011)

But we don't actually care about this task!

- we'll only extract the learned classifier weights to be the **word embeddings**

The most popular word embedding model: **word2vec** (Tomáš Mikolov!)

- Fast, easy to train (much faster than SVD)
- Pretrained embeddings available online

word2vec: Skip-Gram Training

Let's look at a word2vec variant: **skip-gram with negative sampling (SGNS)**

Idea: predict if a candidate word ***c*** is a neighbor of ***t***

1. The target word ***t*** and a neighboring context word ***c*** are **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

Let's look at the **Skip-Gram** training approach**:**

Assume a +/- 2 word context window, given training sentence:

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

c1 c2 [Wt] c3 c4

Skip-Gram Classifier

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

c1 c2 [Wt] c3 c4

Goal: train a classifier that is given a candidate (**w**ord, **c**ontext) pair

(apricot, jam)

(apricot, aardvark)

…

And assign each pair a probability:

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

Similarity, dot product, probability

Core intuition: base the classification on **embedding similarity** of **w** & **c**

- Remember: two vectors are **similar** if they have a **high dot product**
- Hence, **similarity(w,c) ∝ w · c**

But similarity is just a number…

- we need to normalize to get a “**probability**”!
- How? Well, just use the sigmoid fcn:

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

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

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 have lots of context words.
We'll **assume independence** and just multiply them:

remember the NB/unigrams?

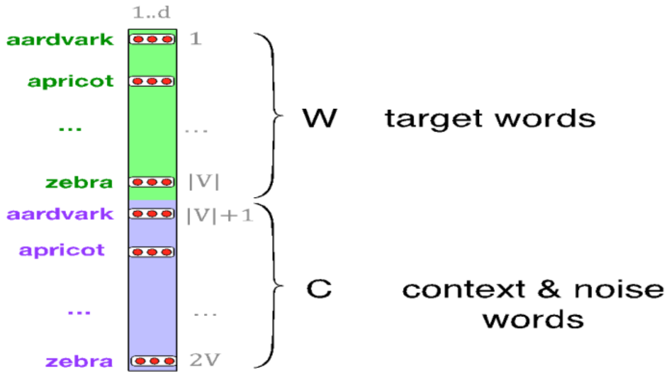
$$P(+|w,c_{1:L}) = \prod_{i=1}^{L} \sigma(c_i \cdot w)$$

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

Skip-gram classifier: Summary

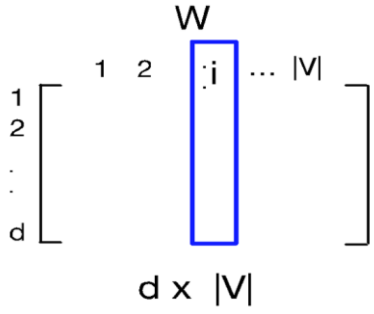
- We train a “probabilistic” classifier, given:
    1. a test target word ***w***
    2. its context window of ***L*** words $\boldsymbol{c}_{1:L}$
    - Estimate probability that **w** occurs in this window based on similarity of **w** (embeddings) to the $\boldsymbol{c}_{1:L}$ (embeddings).

We need to learn the embeddings: $\boldsymbol{\theta} =$



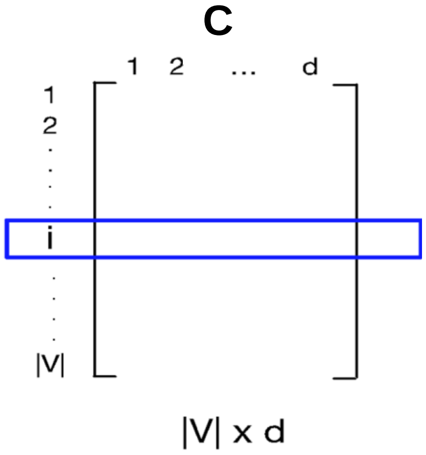
We learn 2 embeddings for each word

1) **input embedding (~word)** *w*, in the input matrix *W*



- Column *i* of the input matrix *W* is the 1×*d* vector embedding w_i for word *i* in the vocabulary.

1) **output embedding (~context)** *c*, in output matrix *C*



- Row *i* of the output matrix *C'* is a *d* × 1 vector embedding c_i for word *i* in the vocabulary.

Learning word2vec embeddings**: Skip-gram**

To obtain the embeddings, we first initialize them randomly, and start **training**

- iteratively shifting the word embeddings to be more like their neighbors

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

c1 c2 [Wt] 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

Learning word2vec embeddings: **Skip-gram**

To obtain the embeddings, we first initialize them randomly, and start **training**

- iteratively shifting the word embeddings to be more like their neighbors

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

c1 c2 [Wt] c3 c4

SGNS version of word2vec

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 vector embeddings

Given:

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

Goal:

- learn to adjust those word vectors such that we:
    - **Maximize** the similarity of the target & 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

- **Maximize** the similarity of the target & 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

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

For a ***w*** *with context* $\boldsymbol{c_{pos}}$, $\boldsymbol{c_{neg1}} \ldots \boldsymbol{c_{negk}}$

we assume independency of words in **c**

Training the classifier

Finally, we minimize the loss with **Stochastic Gradient Descent**

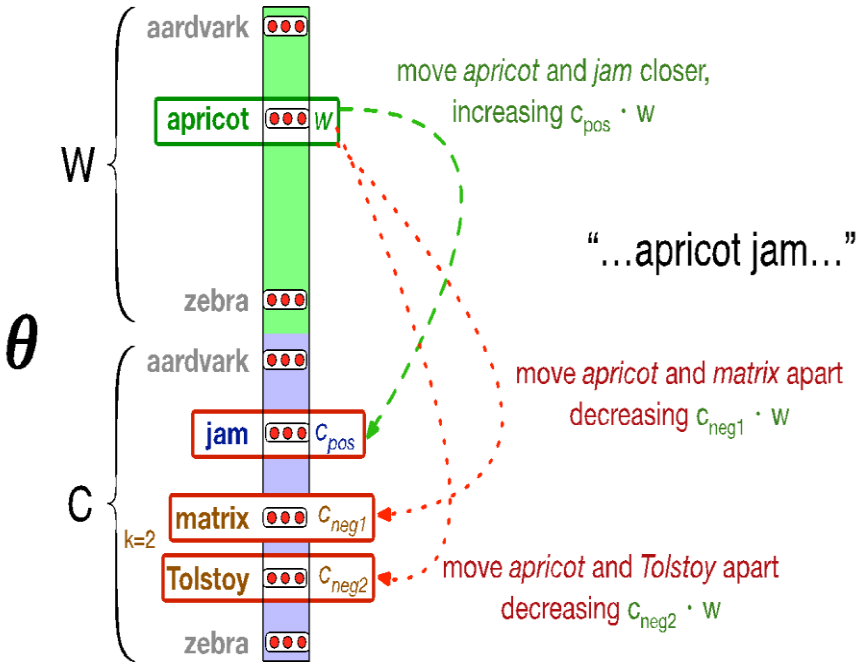
$$L_{CE} = -\left[\log \sigma(c_{pos} \cdot w) + \sum_{i=1}^{k} \log \sigma(-c_{neg_i} \cdot w)\right]$$

$$\frac{\partial L_{CE}}{\partial c_{pos}} = [\sigma(c_{pos} \cdot w) - 1]w$$

$$\frac{\partial L_{CE}}{\partial c_{neg}} = [\sigma(c_{neg} \cdot w)]w$$

$$\frac{\partial L_{CE}}{\partial w} = [\sigma(c_{pos} \cdot w) - 1]c_{pos} + \sum_{i=1}^{k} [\sigma(c_{neg_i} \cdot w)]c_{neg_i}$$

derivatives of sigmoids have elegant form…



Word2vec learning summary

How to learn word2vec (skip-gram) embeddings:

1. Start with **V random d**-dimensional vectors as initial embeddings
2. Train a classifier based on embedding **similarity** loss measure
3. From a corpus take **pairs** of words that **co-occur** as **positive** examples
4. Take pairs of words that **don't co-occur** as **negative** examples
5. **Train** the classifier to **distinguish** these by slowly adjusting all the embeddings to improve the classifier performance
6. Throw away the classifier code and keep the **embeddings**.

We actually end up with both target word **W and** context **C** embeddings!

- to represent a word *i* we commonly just **add** these as $\mathbf{w_i} + \mathbf{c_i}$

Relation between skip-grams and PMI!

- Note that if we multiply WC^T
  - We get a |V|x|V| matrix M , where each entry m_{ij} corresponds to some association between input word i and output word j
  - I can be shown that skip-gram reaches its optimum just when this matrix M is a shifted version of the PMI matrix:

$$WC^T = M^{PMI} - \log k$$

instead of truncating at 0 (PPMI), we shift by **k** (negative sampling)

  - So, skip-gram word2vec is implicitly factoring a shifted version of the PMI matrix into the two embedding matrices!

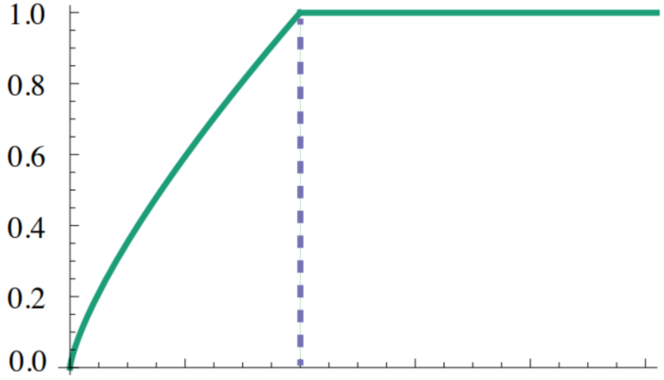
GloVe

- Can we combine these 2 approaches?
  - To make use of the co-occurrence counts
  - while avoiding the full matrix decomposition
- **GloVe** = Global Vectors
  - introduces a custom loss fcn **L**
- We iterate through all pairs of words **in X**
  - optimizing one co-occurrence **count** at a time
- No need to iterate the large text corpus
  - just through the aggregated counts
- Fast training, good even with small data

co-occurrence matrix

$$L = \frac{1}{2} \sum_{i,j=1}^{|V|} f(X_{ij})(w_i^T c_j - log\, X_{ij})^2$$

frequent pairs discounting



Vector Semantics

- Properties of learned embeddings

Nearest neighbors and window size

target:	Redmond	Havel	ninjutsu	graffiti	capitulate
	Redmond Wash.	Vaclav Havel	ninja	spray paint	capitulation
	Redmond Washington	president Vaclav Havel	martial arts	grafitti	capitulated
	Microsoft	Velvet Revolution	swordsmanship	taggers	capitulating

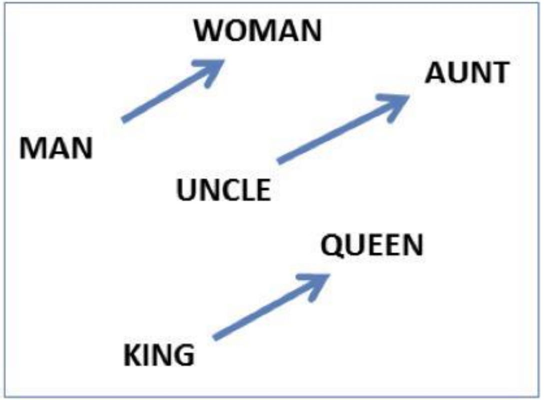
Small windows (C= +/- 2) : nearest words are syntactically similar words

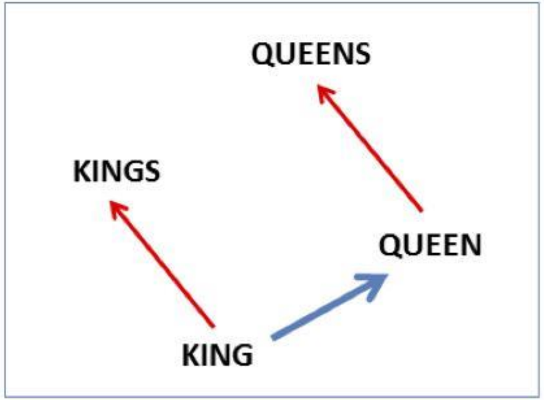
- *Hogwarts* nearest neighbors are other fictional schools:
- *Sunnydale, Evernight, Blandings*

Large windows (C= +/- 5) : nearest words are topically related words

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

Embedding space has neat geometrical relations





With that we can solve word analogies!:

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

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

Embedding space geometry: GloVe

Nearest words to
frog:

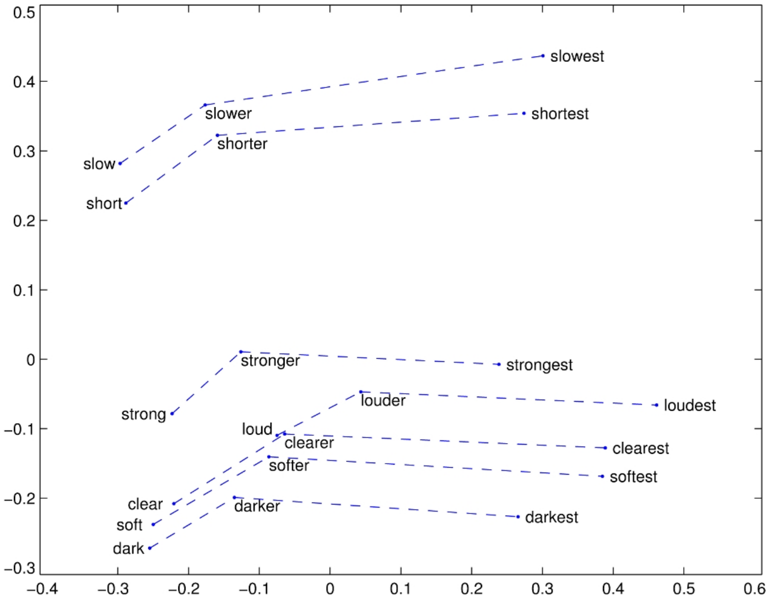
1. frogs
2. toad
3. litoria
4. leptodactylidae
5. rana
6. lizard
7. eleutherodactylus

litoria

leptodactylidae

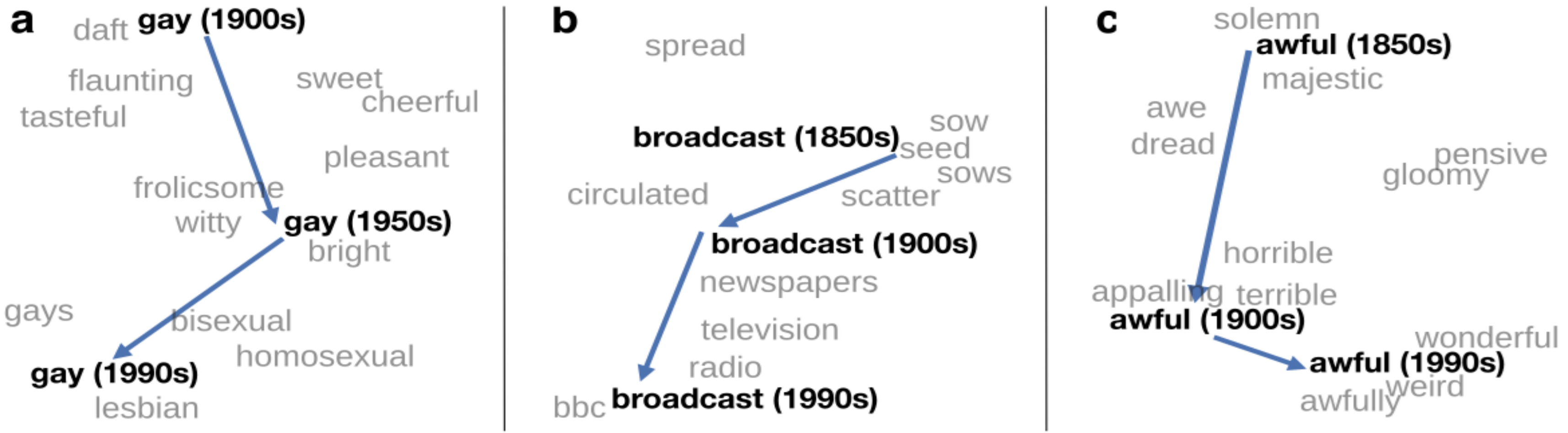
rana

eleutherodactylus



Embeddings as a window into **historical** semantics

~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**!

Ask “Paris : France :: Tokyo : x”

- x = Japan

Ask “father : doctor :: mother : x”

- x = nurse

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

- x = homemaker

this can be a serious problem, why?

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.