自然语言处理技术

语言表示与相似度计算

孙承杰
计算机科学与技术学院

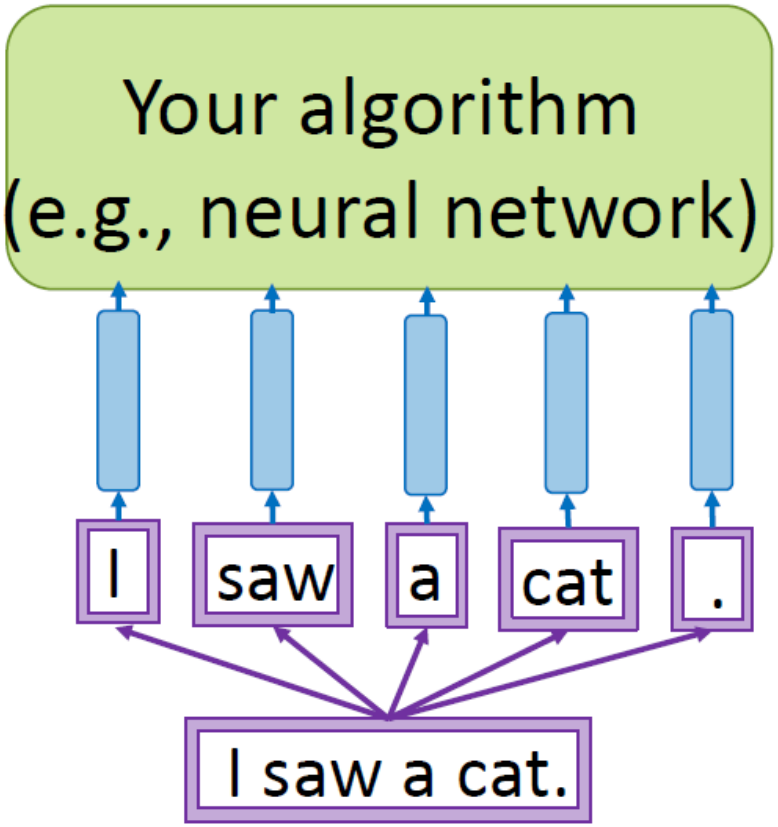
主要内容

- 词语和文档表示方法
- 距离度量与相似度计算

词语和文档表示方法

- One-hot representations
- Bag of Words representations
- Distributional representations
  - Latent Semantic Indexing
  - Topic model
- Distributed representation

词的表示方法是NLP的基础



Any algorithm for solving any task

Word representation - vector
(word embedding)

Sequence of tokens

Text

One-hot representations

- Simple way to encode discrete concepts

  Example:

  vocabulary = (Monday, Tuesday, is, a, today)

  Monday = [1 0 0 0 0]

  Tuesday = [0 1 0 0 0]

  is = [0 0 1 0 0]

  a = [0 0 0 1 0]

  Today = [0 0 0 0 1]

  – The vector dimension is the size of the vocabulary

Bag of words representations

- Sum of one-hot codes
- Ignores order of words

  Example:

  vocabulary = (Monday, Tuesday, is, a, today)

  Monday Monday= [2 0 0 0 0]

  today is a Monday = [1 0 1 1 1]

  today is a Tuesday = [0 1 1 1 1]

  is a Monday today = [1 0 1 1 1]

Bag of words

- Words are features/index
- Weight of features/index
  - Binary value
    - 1: occurrence 0:otherwise
  - Real value
    - TF*IDF (TF: term frequency, IDF: inverse document frequency)
- Vectorization
  - document->vector

TF·IDF

f_{ij} = frequency of term t_i in document d_j

$$TF_{ij} = \frac{f_{ij}}{\max_k f_{kj}}$$

n_i = number of docs that mention term i

N = total number of docs

$$IDF_i = \log \frac{N}{n_i}$$

TF·IDF score $w_{ij} = TF_{ij} * IDF_i$

Doc profile = set of words with highest TF·IDF scores, together with their scores

Tips: **Karen Spärck Jones**

- 26 August 1935 – 4 April 2007
- she introduced IDF in 1972. IDF is used in most search engines today, usually as part of the tf-idf weighting scheme.
- Spärck Jones, Karen (1972). "A statistical interpretation of term specificity and its application in retrieval". *Journal of Documentation* **28** (1): 11–21. doi:10.1108/eb026526.
- the ACL Lifetime Achievement Award (2004)

http://en.wikipedia.org/wiki/Karen_Sp%C3%A4rck_Jones

Problems with discrete representation

- Semantic Gap
  - motel [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0] AND

  hotel [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0] = 0
- High dimension
  - Vocabulary size
- Sparsity issues

Distributional representations

- You shall know a word by the company it keeps (Firth, J. R. 1957:11)
  - Representing a word by means of its neighbors
  - One of the most successful ideas of modern statistical NLP

sugar, a sliced lemon, a tablespoonful of	**apricot**	jam, a pinch each of,
their enjoyment. Cautiously she sampled her first	**pineapple**	and another fruit whose taste she likened
well suited to programming on the digital	**computer**.	In finding the optimal R-stage policy from
for the purpose of gathering data and	**information**	necessary for the study authorized in the

	aardvark	computer	data	pinch	result	sugar	…
apricot	0	0	0	1	0	1	
pineapple	0	0	0	1	0	1	
digital	0	2	1	0	1	0	
information	0	1	6	0	4	0	

Distributional representations

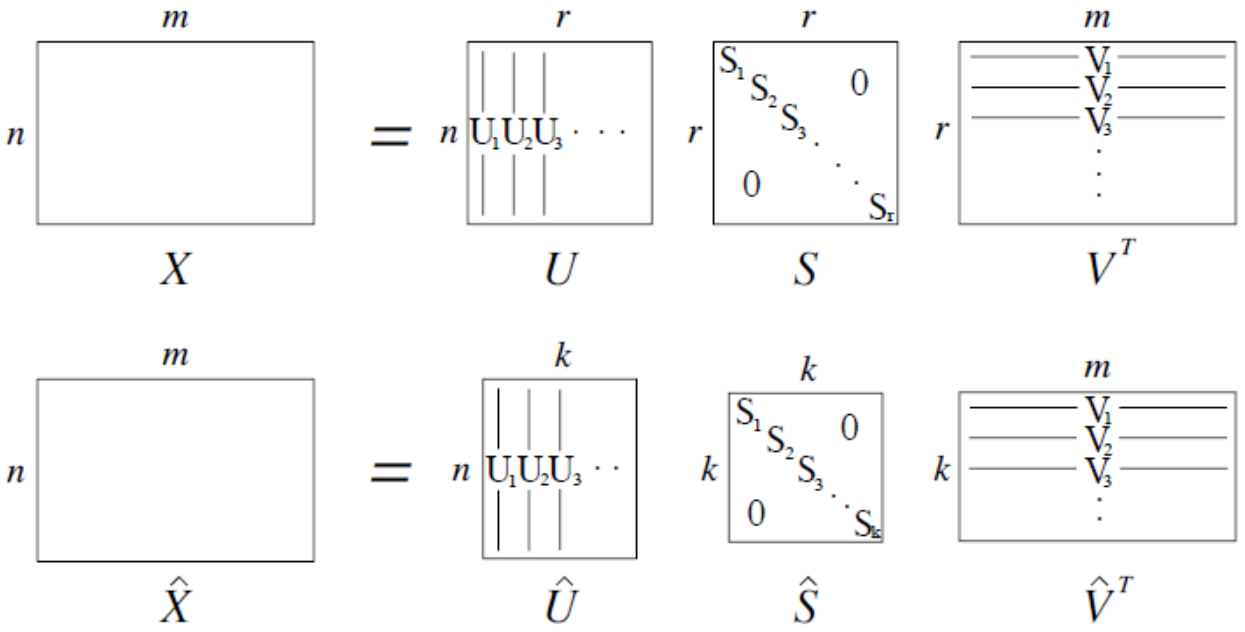
- Idea: store “most” of the important information in a fixed, small number of dimensions: a dense vector
- Usually around 25 – 1000 dimensions
- How to reduce the dimensionality?

Latent Semantic Indexing

- Representing words in a vector space
- The feature vectors of words is learned based on the word-document co-occurrence
  - Indexing by latent semantic analysis. S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, R. Harshman. JASIS. 1990.

Latent Semantic Indexing

- Latent semantic indexing (LSI) is an indexing and retrieval method
  - **Uses singular value decomposition (SVD)** to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of text.



Simple SVD word vectors in Python

corpus = "I like deep learning. I like NLP. I enjoy flying."

```
import numpy as np
la = np.linalg
words = ["I", "like", "enjoy",
         "deep","learnig","NLP","flying","."]
X = np.array([[0,2,1,0,0,0,0,0],
              [2,0,0,1,0,1,0,0],
              [1,0,0,0,0,0,1,0],
              [0,1,0,0,1,0,0,0],
              [0,0,0,1,0,0,0,1],
              [0,1,0,0,0,0,0,1],
              [0,0,1,0,0,0,0,1],
              [0,0,0,0,1,1,1,0]])

U, s, Vh = la.svd(X, full_matrices=False)
```

Problems with SVD

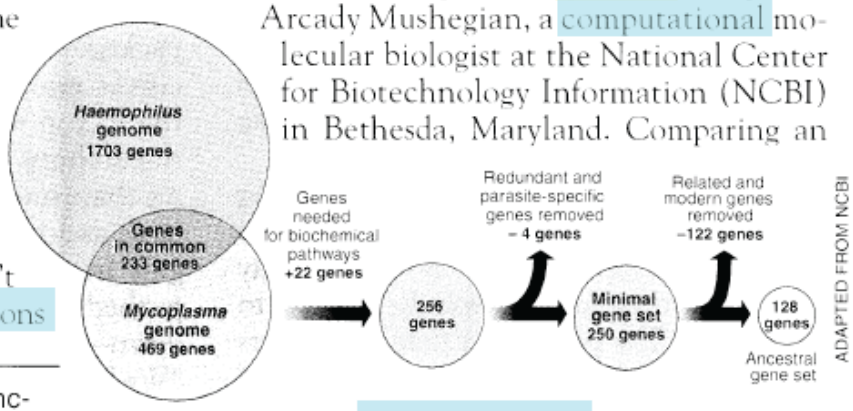
- Computational cost scales quadratically for nxm matrix:
  - O(mn^2) flops (when n<m)
  - Bad for millions of words or documents
- Hard to incorporate new words or documents

Topic model

Seeking Life's Bare (Genetic) Necessities

COLD SPRING HARBOR, NEW YORK—How many genes does an organism need to survive? Last week at the genome meeting here,* two genome researchers with radically different approaches presented complementary views of the basic genes needed for life. One research team, using computer analyses to compare known genomes, concluded that today's organisms can be sustained with just 250 genes, and that the earliest life forms required a mere 128 genes. The other researcher mapped genes in a simple parasite and estimated that for this organism, 800 genes are plenty to do the job—but that anything short of 100 wouldn't be enough.

Although the numbers don't match precisely, those predictions "are not all that far apart," especially in comparison to the 75,000 genes in the human genome, notes Siv Andersson of Uppsala University in Sweden, who arrived at the 800 number. But coming up with a consensus answer may be more than just a genetic numbers game, particularly as more and more genomes are completely mapped and sequenced. "It may be a way of organizing any newly sequenced genome," explains Arcady Mushegian, a computational molecular biologist at the National Center for Biotechnology Information (NCBI) in Bethesda, Maryland. Comparing an



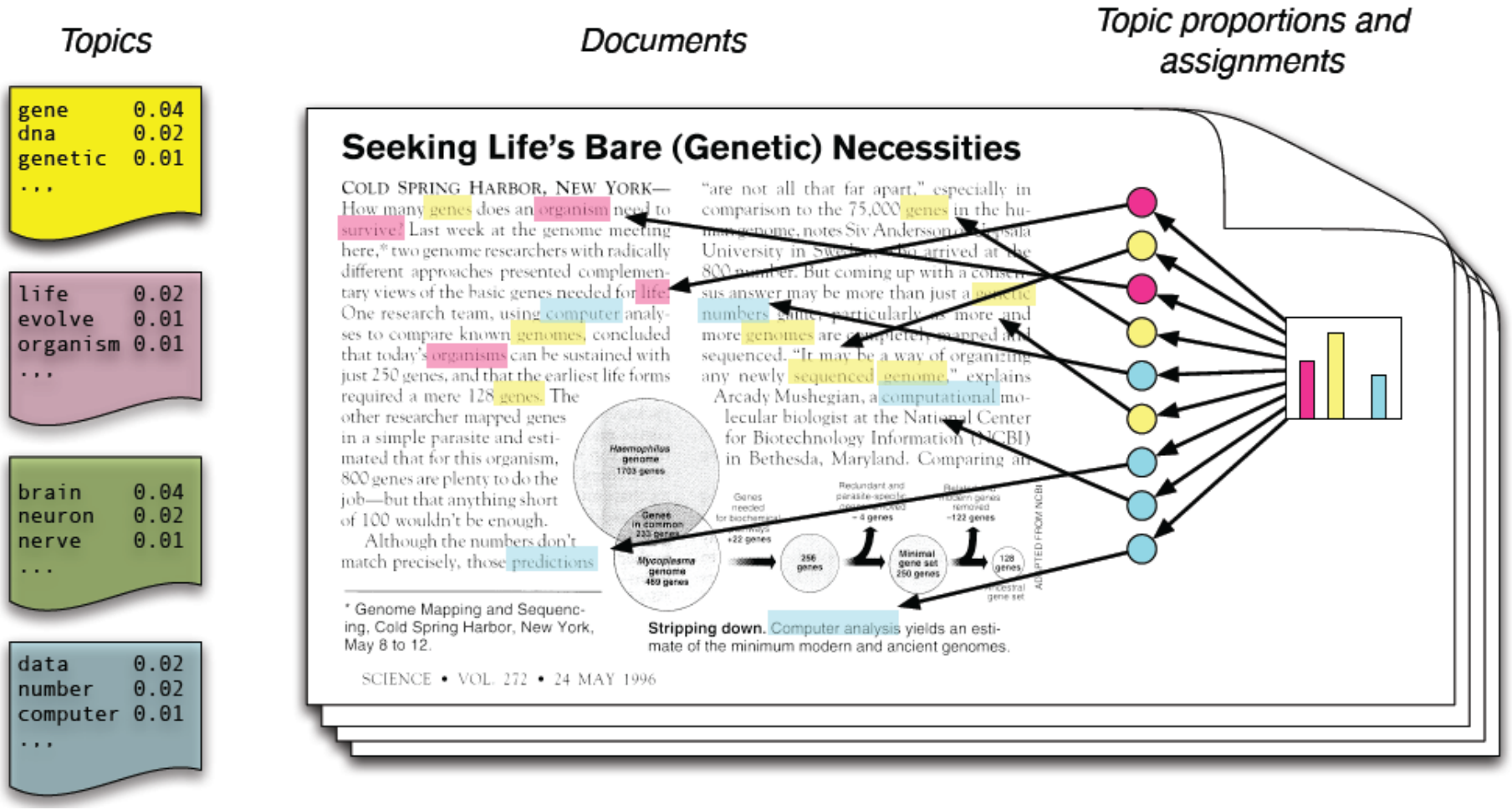
Stripping down. Computer analysis yields an estimate of the minimum modern and ancient genomes.

* Genome Mapping and Sequencing, Cold Spring Harbor, New York, May 8 to 12.

SCIENCE • VOL. 272 • 24 MAY 1996

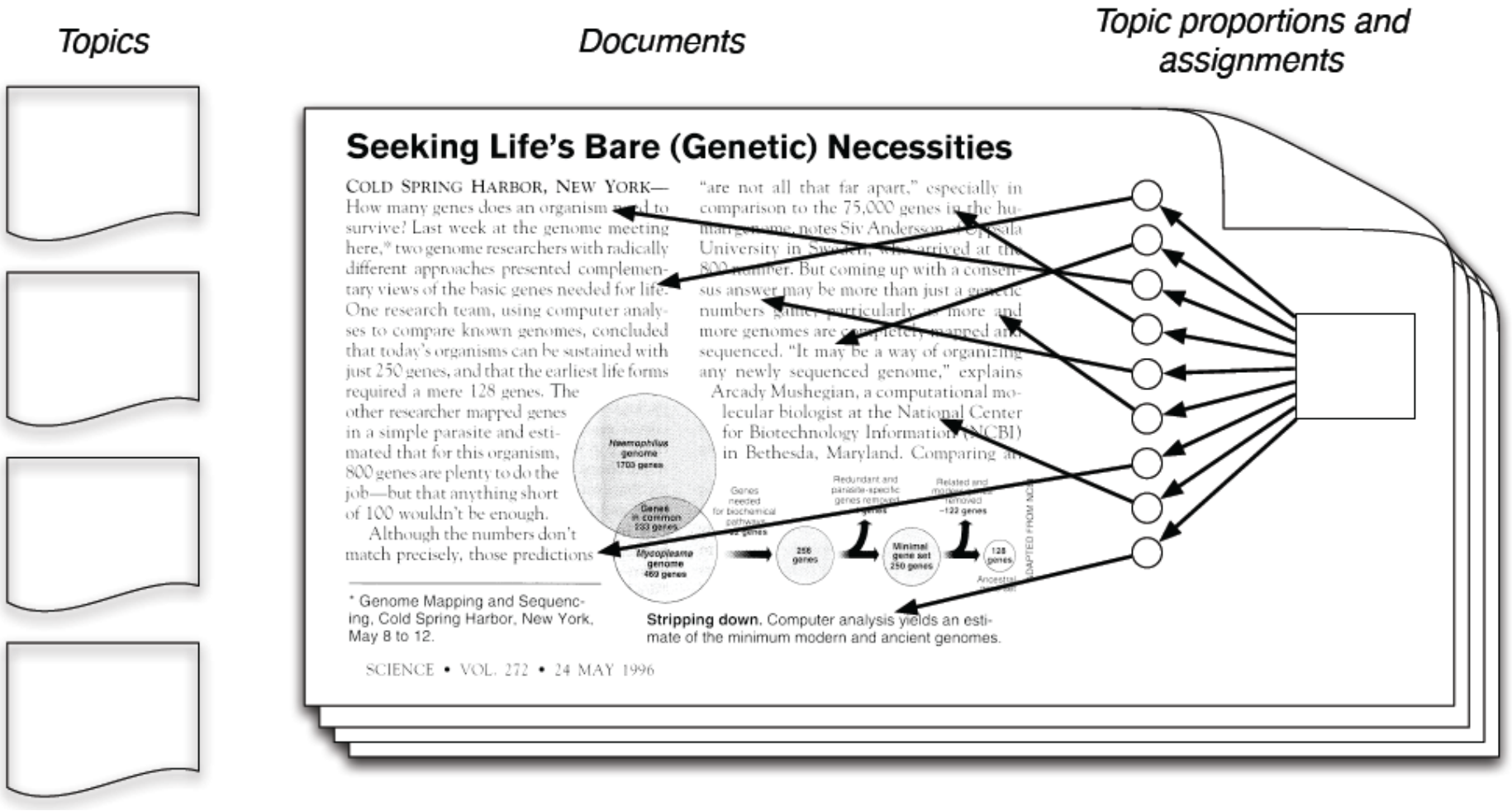
Simple intuition: Documents exhibit multiple topics.

Topic model



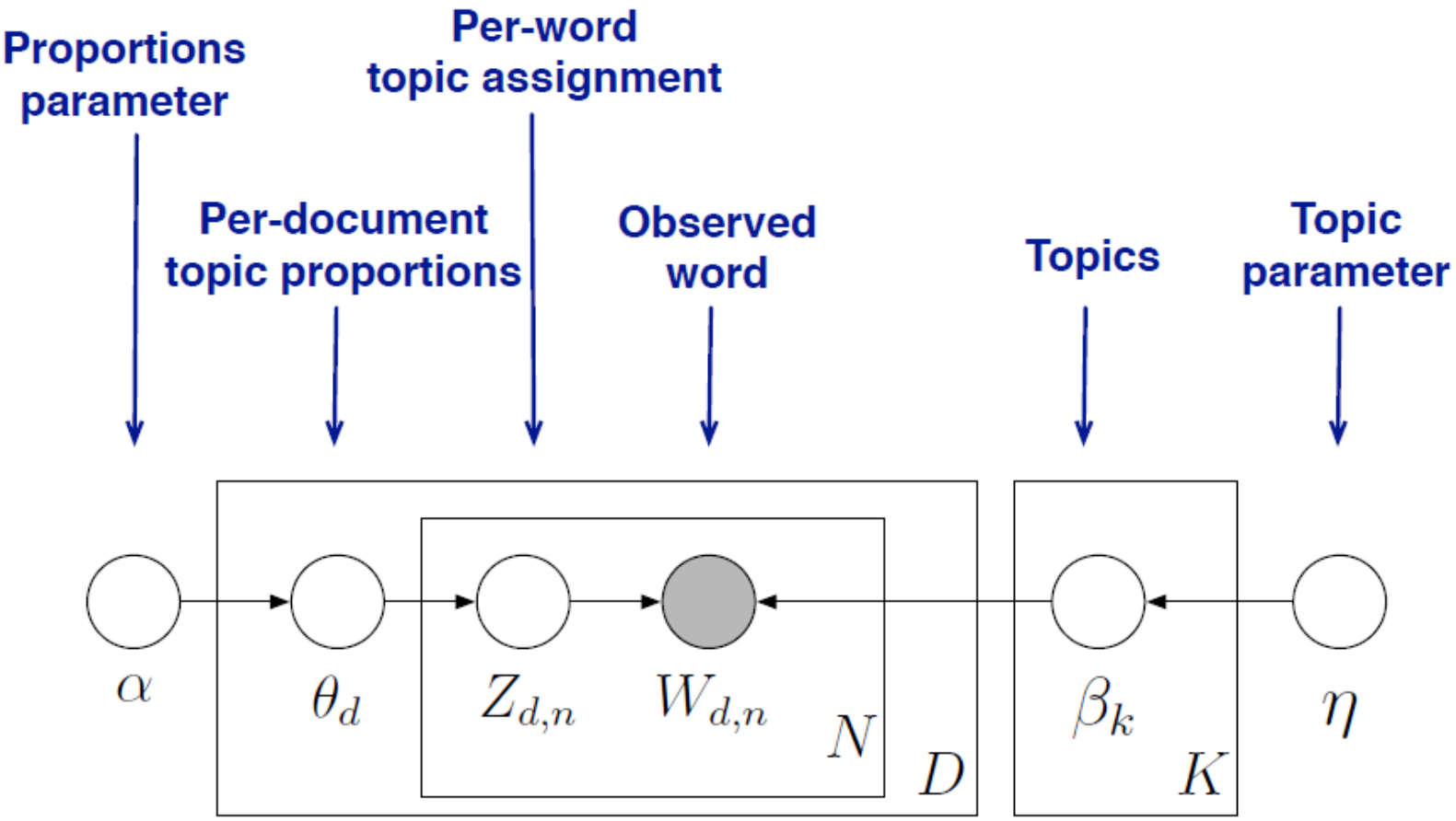
- Each topic is a distribution over words
- Each document is a mixture of corpus-wide topics
- Each word is drawn from one of those topics

Topic model



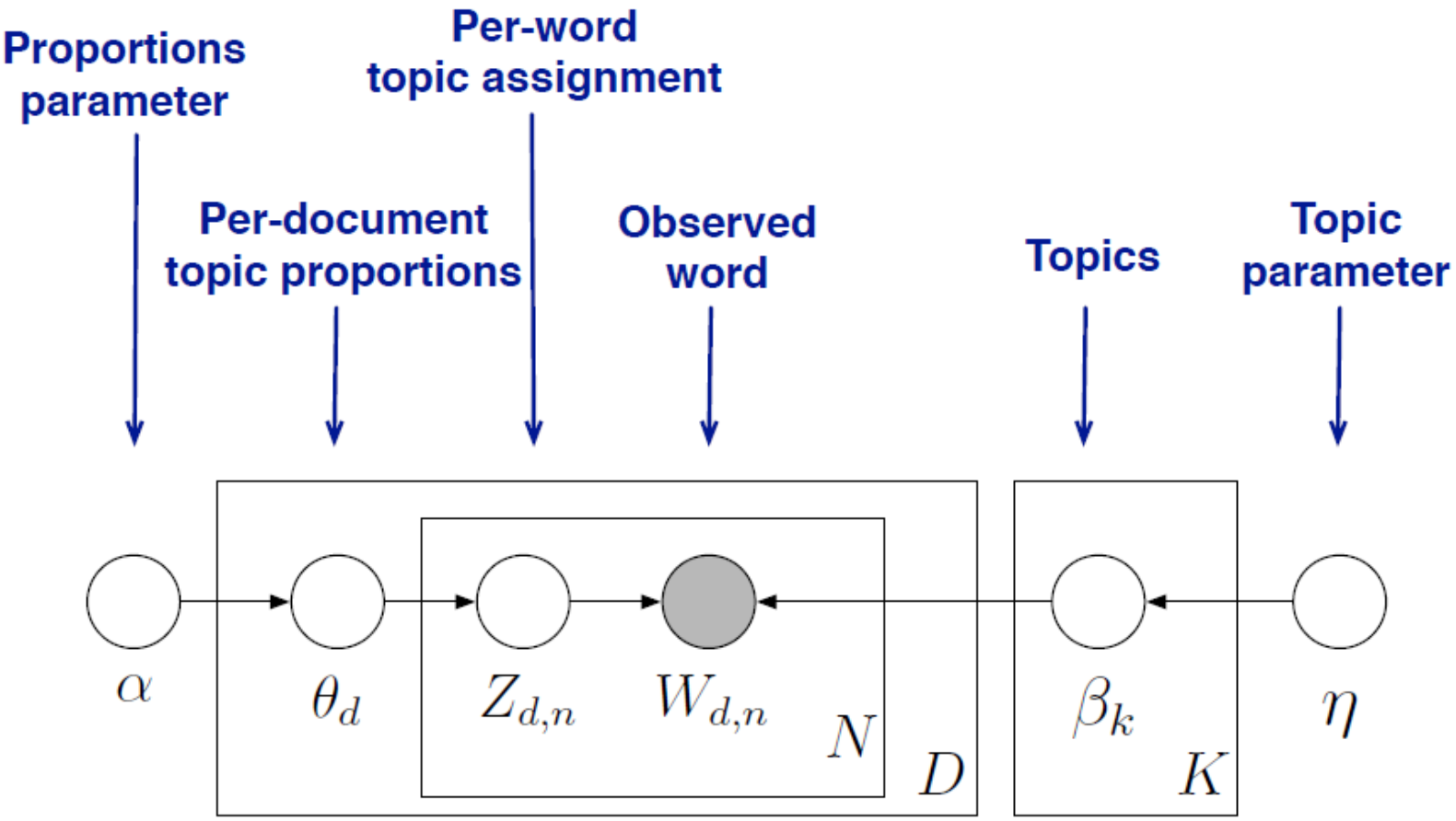
- Our goal is to **infer** the hidden variables
- I.e., compute their distribution conditioned on the documents

Latent Dirichlet allocation (LDA)



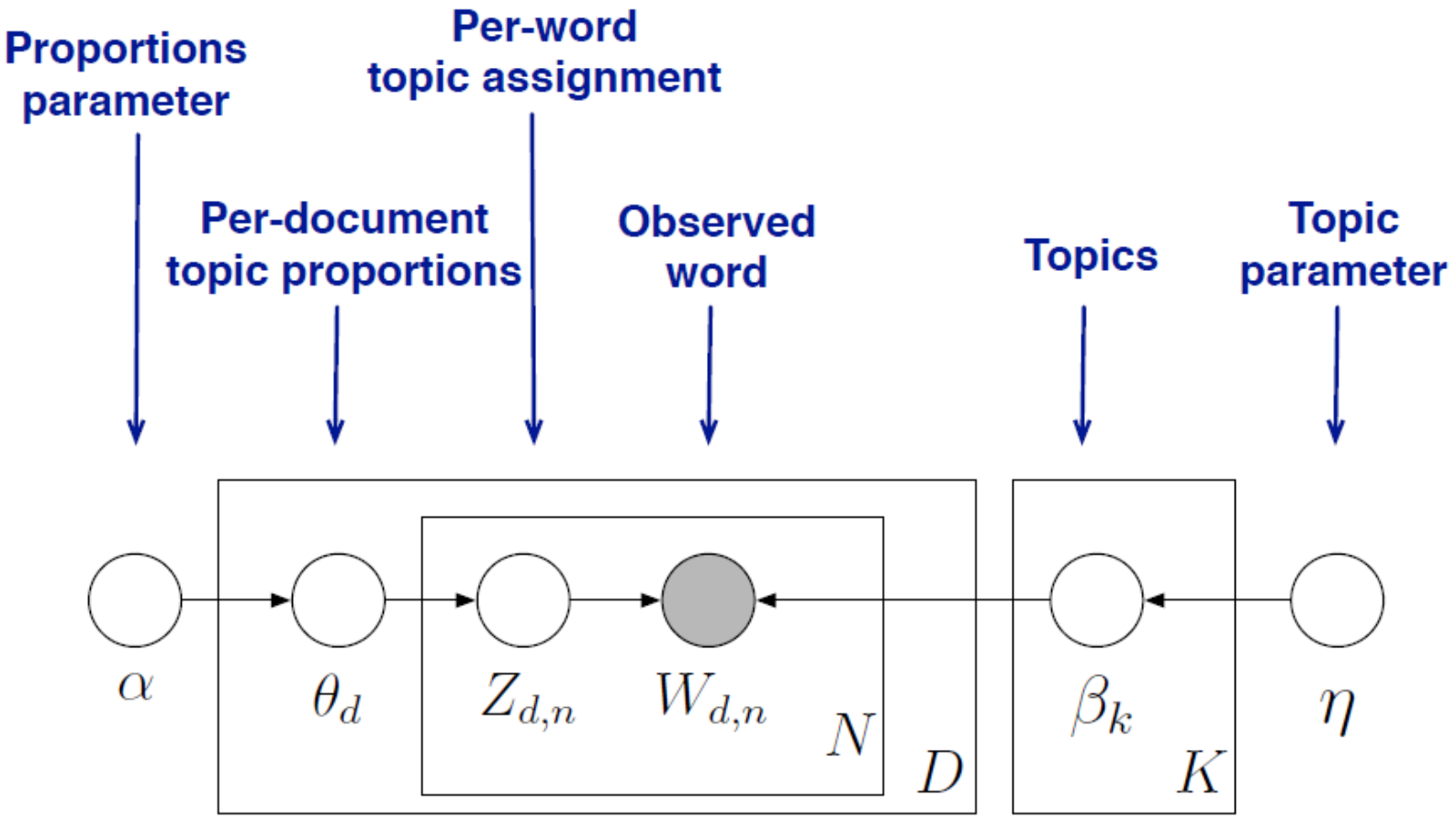
- Encodes our assumptions about the data
- Connects to algorithms for computing with data
- See Pattern Recognition and Machine Learning (Bishop, 2006).

Latent Dirichlet allocation (LDA)



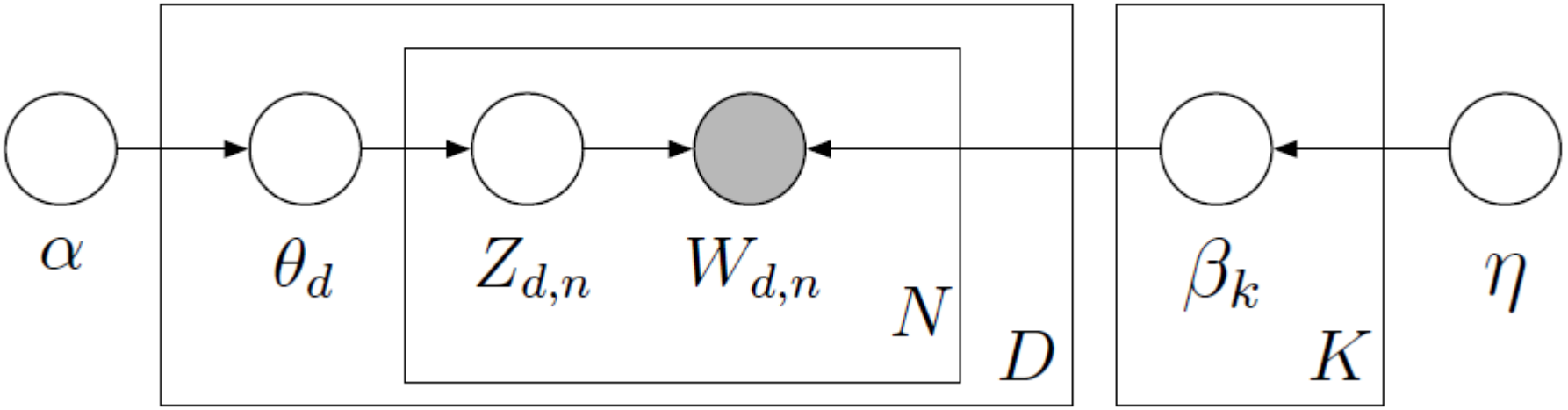
- Nodes are random variables; edges indicate dependence.
- Shaded nodes are observed.
- Plates indicate replicated variables.

Latent Dirichlet allocation (LDA)



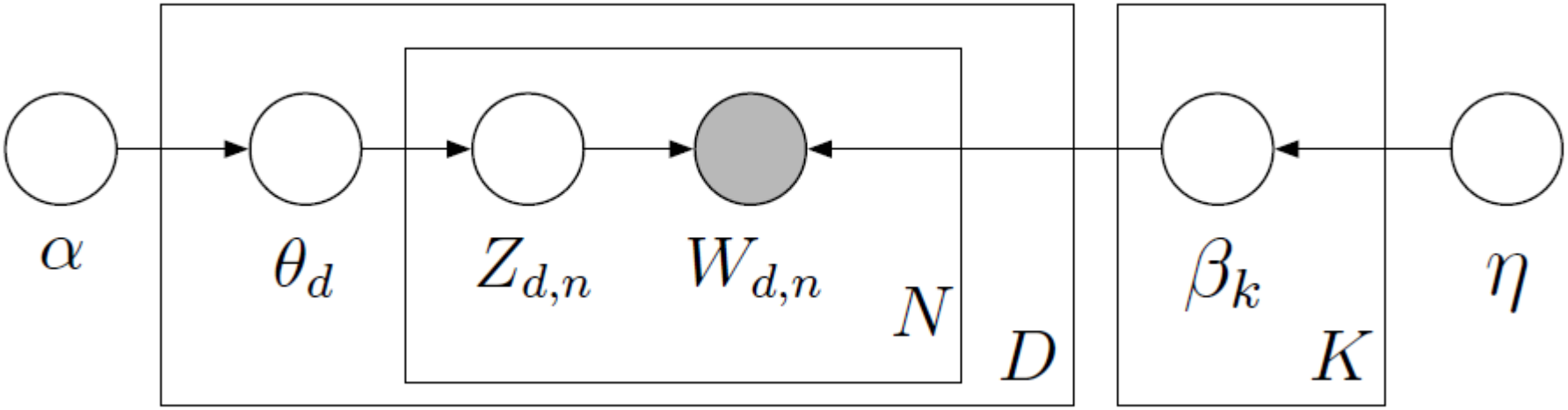
$$\prod_{i=1}^{K} p(\beta_i \mid \eta) \prod_{d=1}^{D} p(\theta_d \mid \alpha) \left(\prod_{n=1}^{N} p(z_{d,n} \mid \theta_d) p(w_{d,n} \mid \beta_{1:K}, z_{d,n}) \right)$$

LDA



- This joint defines a posterior.
- From a collection of documents, infer
  - Per-word topic assignment $z_{d,n}$
  - Per-document topic proportions θ_d
  - Per-corpus topic distributions β_k
- Then use posterior expectations to perform the task at hand, e.g., information retrieval, document similarity, exploration, ...

LDA



- Approximate posterior inference algorithms
- Mean field variational methods (Blei et al., 2001, 2003)
- Expectation propagation (Minka and Lafferty, 2002)
- Collapsed Gibbs sampling (Griffiths and Steyvers, 2002)
- Collapsed variational inference (Teh et al., 2006)
- Online variational inference (Hoffman et al., 2010)
- Also see Mukherjee and Blei (2009) and Asuncion et al. (2009).

Implementations of LDA

- There are many available implementations of topic modeling—
  - **LDA-C** * A C implementation of LDA
  - **HDP** * A C implementation of the HDP ("infinite LDA")
  - **Online LDA** * A python package for LDA on massive data
  - **LDA in R** * Package in R for many topic models
  - **LingPipe** Java toolkit for NLP and computational linguistics
  - **Mallet** Java toolkit for statistical NLP
  - **TMVE** * A python package to build browsers from topic models

* available at www.cs.princeton.edu/blei/

Distributed Representation

- Directly learn low-dimensional word vectors
  - A neural probabilistic language model (Bengio et al., 2003)
  - NLP from Scratch (Collobert & Weston, 2008)
  - Word2vec (Mikolov et al. 2013)

The definition of “distributed representation”

- “Distributed representation” means a many-to-many relationship between two types of representation (such as concepts and neurons).
  - Each concept is represented by many neurons
  - Each neuron participates in the representation of many concepts

Dense embeddings you can download!

- **Word2vec** (Mikolov et al.)
  - https://code.google.com/archive/p/word2vec/
- **Fasttext** http://www.fasttext.cc/
- **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

- 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*"?

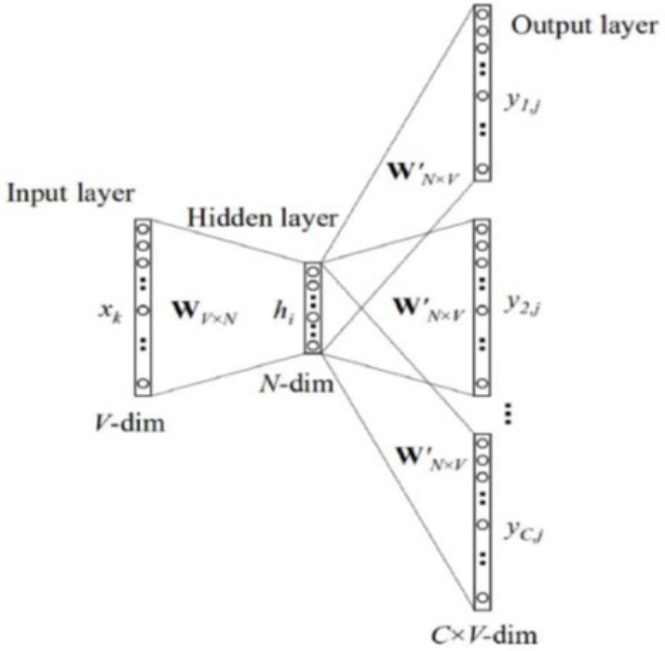
- We don’t actually care about this task
  - But we'll take the learned classifier weights as the word embeddings

Brilliant insight: Use running text as implicitly supervised training data!

- A word *s* near *apricot*
  - Acts as gold ‘correct answer’ to the question “Is word *w* likely to show up near *apricot*?”
- No need for hand-labeled supervision
- The idea comes from **neural language modeling**
  - Bengio et al. (2003)
  - Collobert et al. (2011)

Word2Vec: **Skip-Gram** Task

- Word2vec provides a variety of options. Let's do
  - "skip-gram with negative sampling" (SGNS)



Skip-gram algorithm

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

Skip-Gram Training Data

- Training sentence:
- ... lemon, a tablespoon of **apricot** jam a pinch ...
- c1 c2 target c3 c4

Asssume context words are those in +/- 2 word window

Skip-Gram Goal

- Given a tuple (t,c) = target, context
  - (*apricot*, *jam*)
  - (*apricot*, *aardvark*)

- Return probability that c is a real context word:
  - P(+|t,c)
  - $P(-|t,c) = 1-P(+|t,c)$

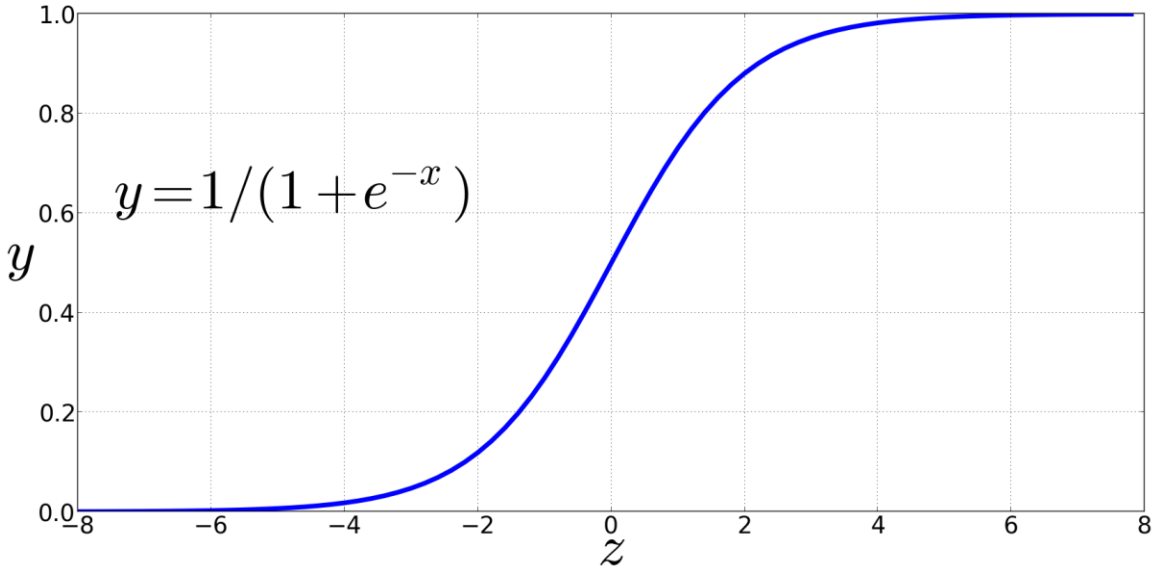
How to compute p(+|t,c)?

- Intuition:
  - Words are likely to appear near similar words
  - Model similarity with dot-product!
  - Similarity(t,c) $\propto t \cdot c$
- *Problem:*
  - *Dot product is not a probability!*
    - *(Neither is cosine)*

Turning dot product into a probability

- The sigmoid lies between 0 and 1:

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



Turning dot product into a probability

$$P(+|t,c) = \frac{1}{1+e^{-t\cdot c}}$$

$$\begin{aligned} P(-|t,c) &= 1 - P(+|t,c) \\ &= \frac{e^{-t\cdot c}}{1+e^{-t\cdot c}} \end{aligned}$$

For all the context words:

- Assume all context words are independent

$$P(+|t, c_{1:k}) = \prod_{i=1}^{\kappa} \frac{1}{1 + e^{-t \cdot c_i}}$$

$$\log P(+|t, c_{1:k}) = \sum_{i=1}^{k} \log \frac{1}{1 + e^{-t \cdot c_i}}$$

Skip-Gram Training Data

- Training sentence:
- ... lemon, a tablespoon of **apricot** jam a pinch ...
- c1 c2 t c3 c4

- Training data: input/output pairs centering on *apricot*
- Asssume a +/- 2 word window

Skip-Gram Training

- Training sentence:
- ... lemon, a tablespoon of **apricot** jam a pinch ...
- c1 c2 t c3 c4

positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	preserves
apricot	or

- For each positive example, we'll create *k* negative examples.
  - Using *noise* words
    - Any random word that isn't *t*

Skip-Gram Training

- Training sentence:
- ... lemon, a tablespoon of **apricot** jam a pinch ...
- c1 c2 t c3 c4

positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	preserves
apricot	or

negative examples - k=2

t	c	t	c
apricot	aardvark	apricot	twelve
apricot	puddle	apricot	hello
apricot	where	apricot	dear
apricot	coaxial	apricot	forever

Choosing noise words

- Could pick w according to their unigram frequency P(w)
- More common to chosen then according to $p_\alpha(w)$

$$P_\alpha(w) = \frac{count(w)^\alpha}{\sum_w count(w)^\alpha}$$

- α= ¾ works well because it gives rare noise words slightly higher probability
- To show this, imagine two events p(a)=.99 and p(b) = .01:

$$P_\alpha(a) = \frac{.99^{.75}}{.99^{.75} + .01^{.75}} = .97$$

$$P_\alpha(b) = \frac{.01^{.75}}{.99^{.75} + .01^{.75}} = .03$$

Setup

- Let's represent words as vectors of some length (say 300), randomly initialized.
- So we start with 300 * V random parameters
- Over the entire training set, we'd like to adjust those word vectors such that we
  - Maximize the similarity of the target word, context word pairs (t,c) drawn from the positive data
  - Minimize the similarity of the (t,c) pairs drawn from the negative data.

Learning the classifier

- Iterative process.
  - We'll start with 0 or random weights
  - Then adjust the word weights to
    - make the positive pairs more likely
    - and the negative pairs less likely

Objective Criteria

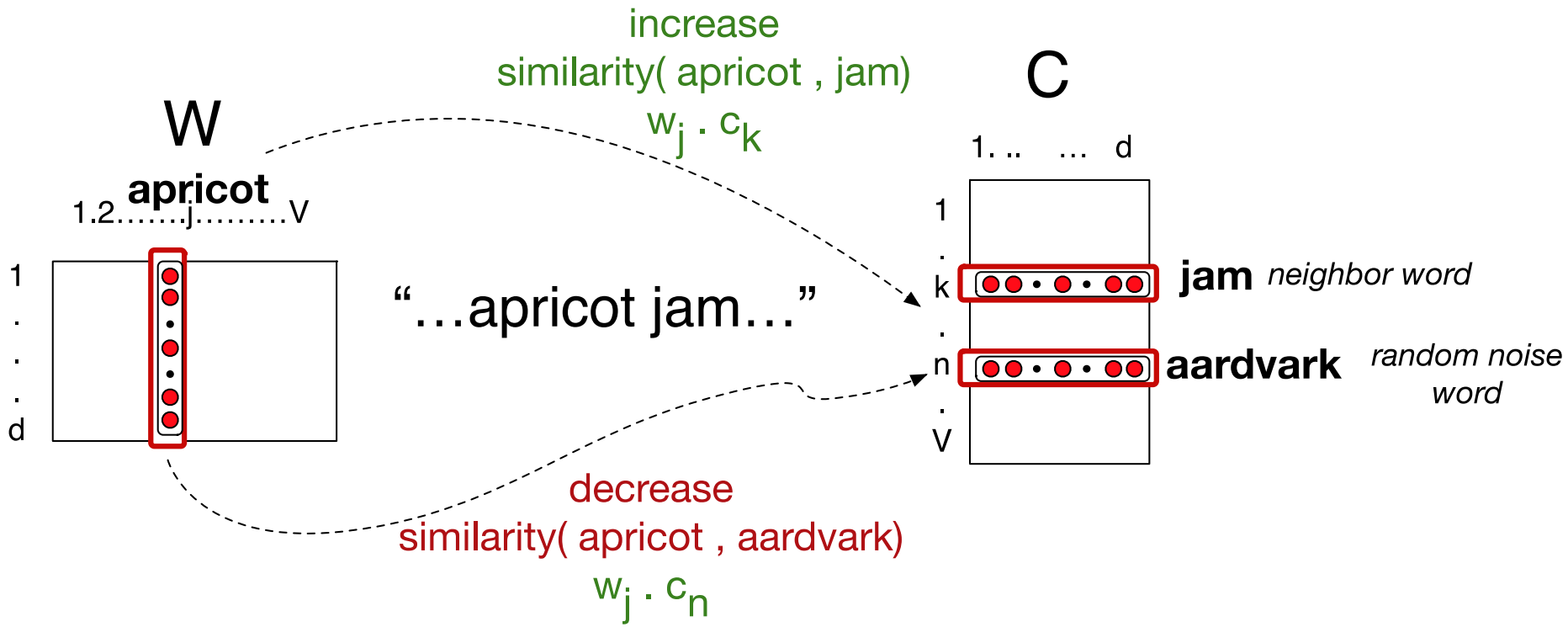
- over the entire training set:
  - We want to maximize…

$$\sum_{(t,c)\in+} logP(+|t,c) + \sum_{(t,c)\in-} logP(-|t,c)$$

  - Maximize the + label for the pairs from the positive training data, and the – label for the pairs sample from the negative data.

Focusing on one target word t:

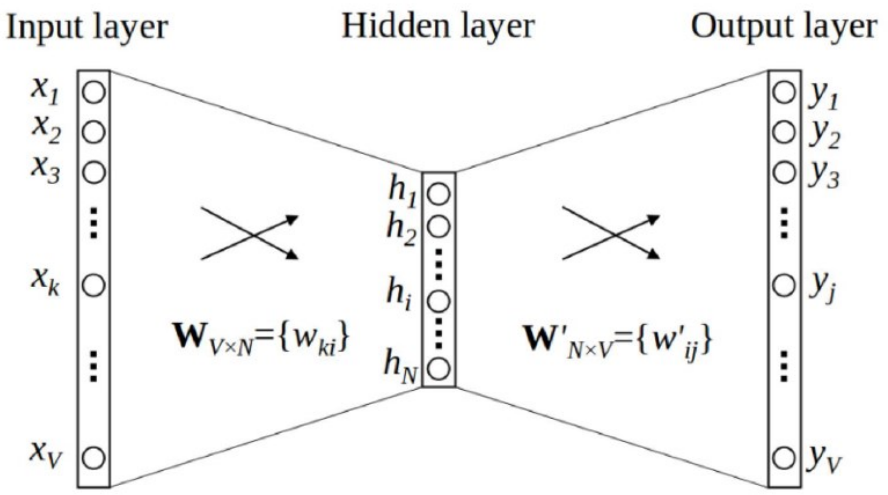
$$
\begin{aligned}
L(\theta) &= \log P(+|t,c) + \sum_{i=1}^{k} \log P(-|t,n_i) \\
&= \log \sigma(c \cdot t) + \sum_{i=1}^{k} \log \sigma(-n_i \cdot t) \\
&= \log \frac{1}{1+e^{-c \cdot t}} + \sum_{i=1}^{k} \log \frac{1}{1+e^{n_i \cdot t}}
\end{aligned}
$$


increase
similarity(apricot , jam)
$w_j \cdot c_k$
W
apricot
1.2.....j.....V
1
.
.
.
d
"…apricot jam…"
C
1. .. ... d
1
.
k
.
n
.
V
jam
neighbor word
aardvark
random noise word
decrease
similarity(apricot , aardvark)
$w_j \cdot c_n$

Train using gradient descent

- Actually learns two separate embedding matrices W and C
- Can use W and throw away C, or merge them somehow

$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \vdots \\ v_{zebra} \\ u_{aardvark} \\ u_a \\ \vdots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$



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

- Start with V random 300-dimensional vectors as initial embeddings
- Use logistic regression, the second most basic classifier used in machine learning after naïve bayes
  - 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.

Evaluating embeddings

- Compare to human scores on word similarity-type tasks:
  - WordSim-353 (Finkelstein et al., 2002)
  - SimLex-999 (Hill et al., 2015)
  - Stanford Contextual Word Similarity (SCWS) dataset (Huang et al., 2012)
  - TOEFL dataset: *Levied is closest in meaning to: imposed, believed, requested, correlated*

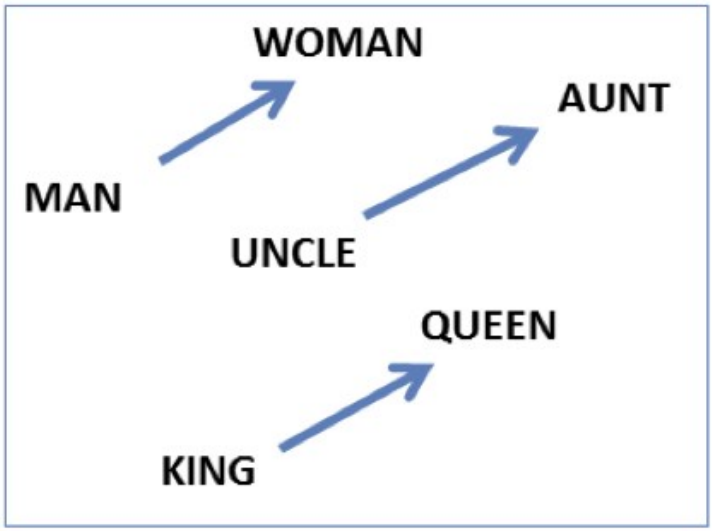
Properties of embeddings

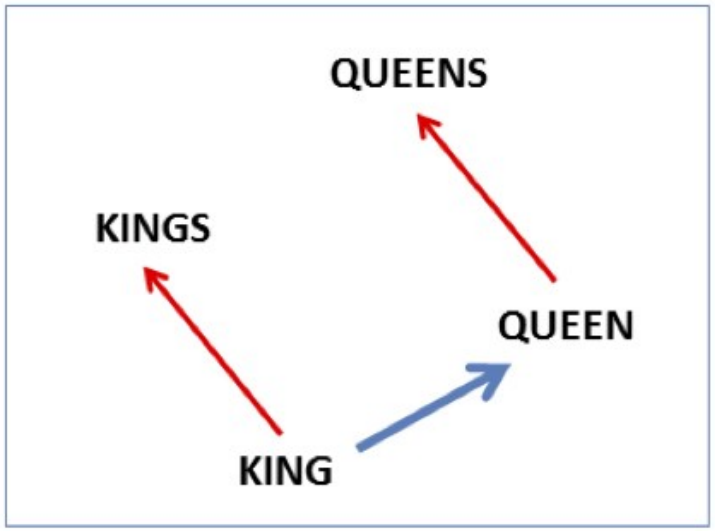
- Similarity depends on window size C
  - C = ±2 The nearest words to *Hogwarts:*
    - *Sunnydale*
    - *Evernight*
  - C = ±5 The nearest words to *Hogwarts:*
    - *Dumbledore*
    - *Malfoy*
    - *halfblood*

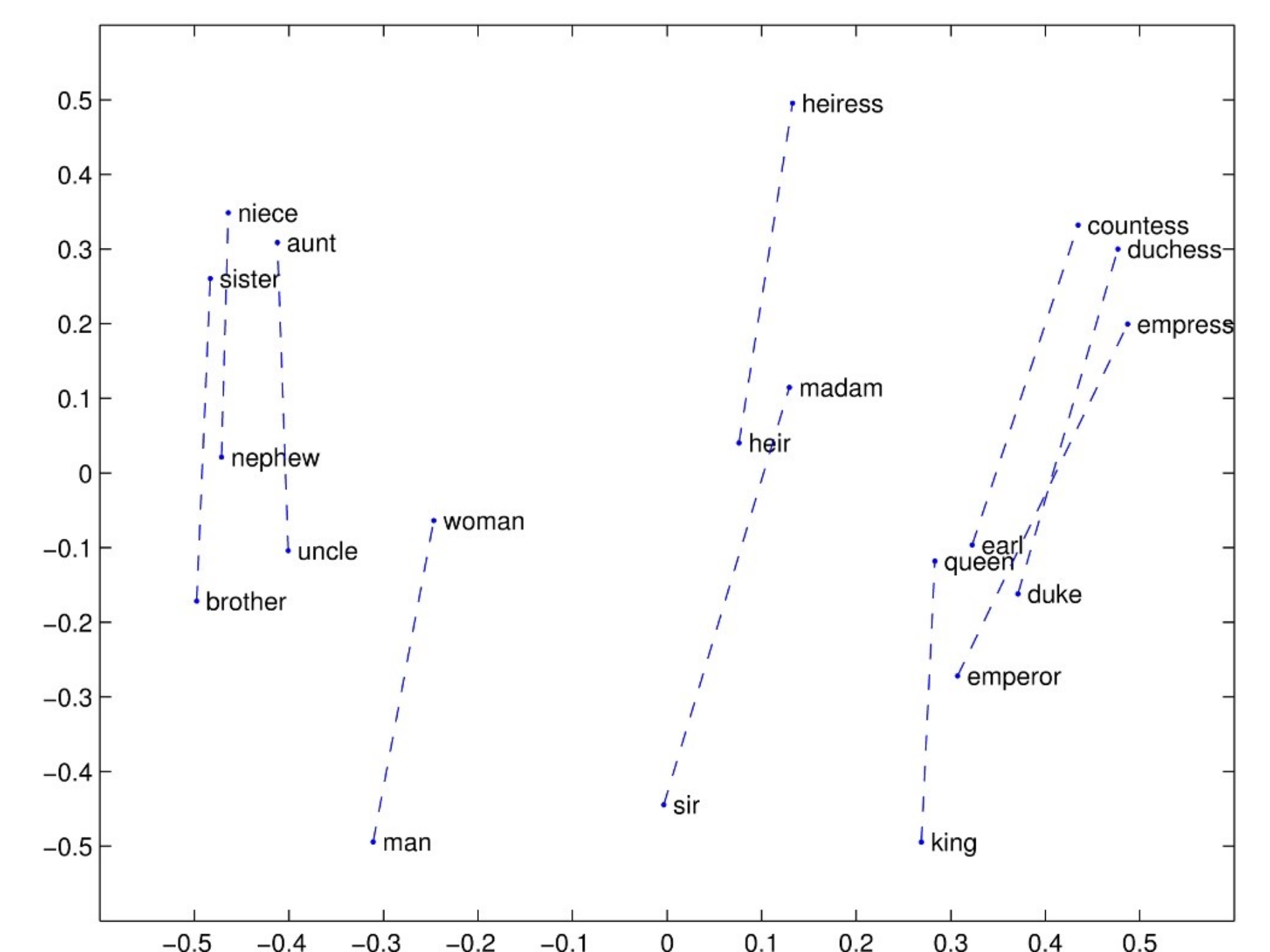
Analogy: Embeddings capture relational meaning!

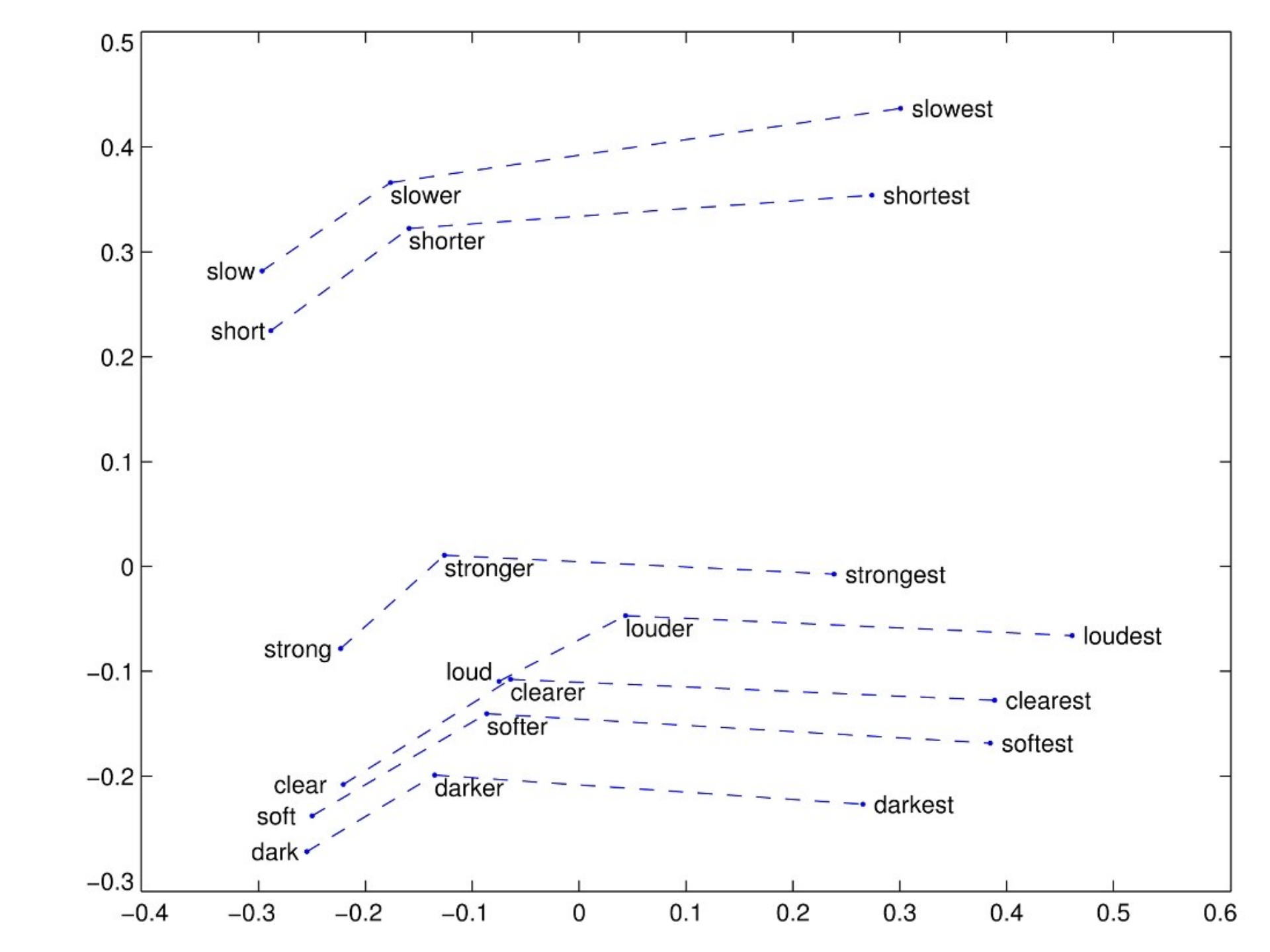
vector(*'king'*) - vector(*'man'*) + vector(*'woman'*) ≈ vector('queen')

vector(*'Paris'*) - vector(*'France'*) + vector(*'Italy'*) ≈ vector('Rome')






heiress
niece
aunt
sister
countess
duchess
empress
madam
heir
nephew
woman
uncle
earl
queen
duke
brother
emperor
sir
man
king
0.5
0.4
0.3
0.2
0.1
0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.5
−0.4
−0.3
−0.2
−0.1
0
0.1
0.2
0.3
0.4
0.5


slowest
slower
shortest
shorter
slow
short
stronger
strongest
louder
loudest
strong
loud
clearer
clearest
softer
softest
clear
darker
darkest
soft
dark
0.5
0.4
0.3
0.2
0.1
0
−0.1
−0.2
−0.3
−0.4
−0.3
−0.2
−0.1
0
0.1
0.2
0.3
0.4
0.5
0.6

Word Vectors Visualization

Word Vectors

- These word vectors can be subsequently used as features in many NLP tasks
- As word vectors can be trained on huge text datasets, they provide generalization for systems trained with limited amount of supervised data
- More complex model architectures can be used for obtaining the word vectors

Word Vectors

- Implementation
  - Word2vec
    - http://word2vec.googlecode.com/svn/trunk/
    - http://radimrehurek.com/gensim/
    - https://github.com/NLPchina/Word2VEC_java

主要内容

- 词语和文档表示方法
- 距离度量与相似度计算

Distance Measures

- Two major classes of distance measure:

1. *Euclidean*
2. *Non-Euclidean*

Euclidean Vs. Non-Euclidean

- A *Euclidean space* has some number of real-valued dimensions and points.
  - A *Euclidean distance* is based on the locations of points in such a space.
- A *Non-Euclidean distance* is based on properties of points, but not their "location" in a space.

Axioms of a Distance Measure

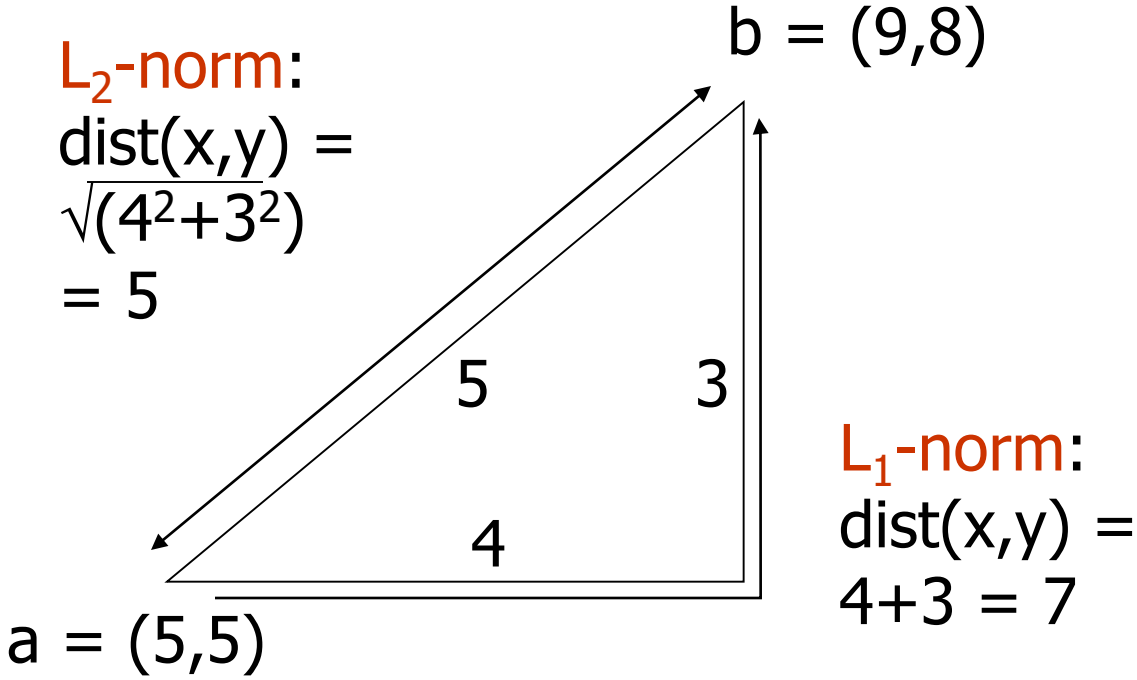
- *d* is a *distance measure* if it is a function from pairs of points to real numbers such that:

1. $d(x,y) \geq 0$.
2. $d(x,y) = 0$ iff $x = y$.
3. $d(x,y) = d(y,x)$.
4. $d(x,y) \leq d(x,z) + d(z,y)$ (*triangle inequality*).

Some Euclidean Distances

- *L_2 norm* : d(x,y) = square root of the sum of the squares of the differences between *x* and *y* in each dimension.
  - The most common notion of "distance."
- *L_1 norm* : sum of the differences in each dimension.
  - *Manhattan distance* = distance if you had to travel along coordinates only.

Examples of Euclidean Distances



Another Euclidean Distance

- L_∞ *norm* : d(x,y) = the maximum of the differences between x and y in any dimension.
- Note: the maximum is the limit as n goes to ∞ of the L_n norm: what you get by taking the n^{th} power of the differences, summing and taking the n^{th} root.

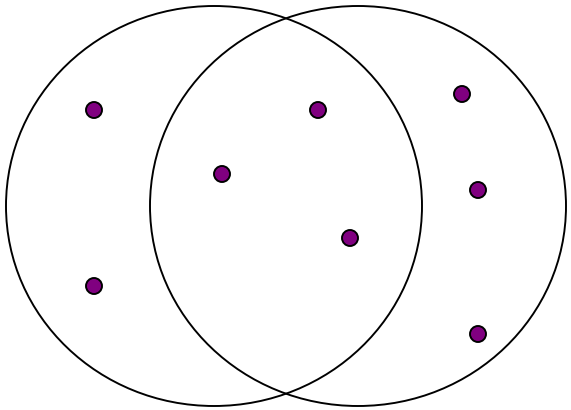
Non-Euclidean Distances

- *Jaccard distance* for sets = 1 minus Jaccard similarity.
- *Cosine distance* = angle between vectors from the origin to the points in question.
- *Edit distance* = number of inserts and deletes to change one string into another.
- *Hamming Distance* = number of positions in which bit vectors differ.

Jaccard Similarity of Sets

- The *Jaccard similarity* of two sets is the size of their intersection divided by the size of their union.
  - *Sim* $(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$.

Example: Jaccard Similarity

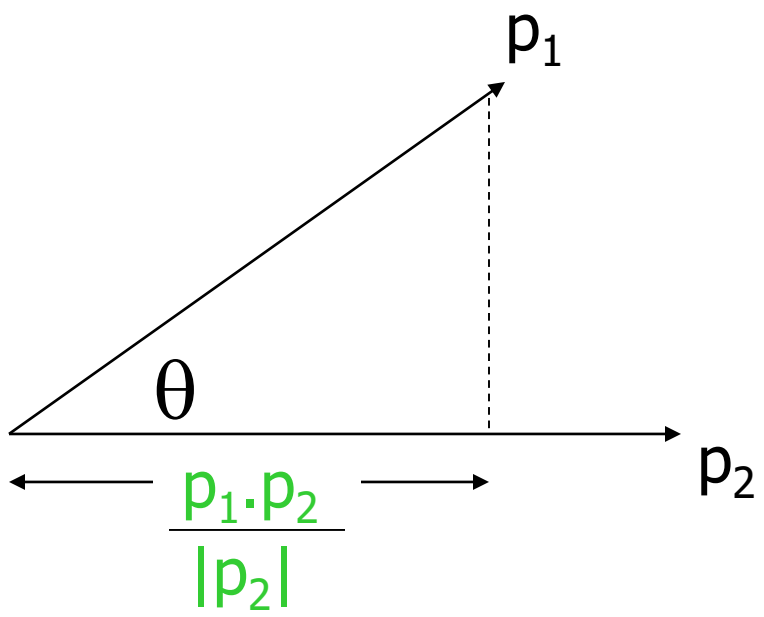


3 in intersection.
8 in union.
Jaccard similarity
= 3/8

Cosine Distance

- Think of a point as a vector from the origin (0,0,...,0) to its location.
- Two points' vectors make an angle, whose cosine is the normalized dot-product of the vectors: $p_1.p_2/|p_2||p_1|$.
  - Example: p_1 = 00111; p_2 = 10011.
  - $p_1.p_2 = 2$; $|p_1| = |p_2| = \sqrt{3}$.
  - $\cos(\theta) = 2/3$; θ is about 48 degrees.

Cosine-Measure Diagram



$$d(p_1, p_2) = \theta = \arccos(p_1.p_2/|p_2||p_1|)$$

Why C.D. Is a Distance Measure

- d(x,x) = 0 because arccos(1) = 0.
- d(x,y) = d(y,x) by symmetry.
- d(x,y) $\geq$ 0 because angles are chosen to be in the range 0 to 180 degrees.
- Triangle inequality: physical reasoning. If I rotate an angle from *x* to *z* and then from *z* to *y*, I can't rotate less than from *x* to *y*.

Edit Distance

- The *edit distance* of two strings is the number of inserts and deletes of characters needed to turn one into the other. Equivalently:
- $d(x,y) = |x| + |y| - 2|LCS(x,y)|$.
  - LCS = *longest common subsequence* = any longest string obtained both by deleting from *x* and deleting from *y*.

Example: LCS

- x = *abcde* ; y = *bcduve*.
- Turn x into y by deleting *a*, then inserting *u* and *v* after *d*.
  - Edit distance = 3.
- Or, LCS(x,y) = *bcde*.
- Note: $|x| + |y| - 2|LCS(x,y)| = 5 + 6 - 2*4 = 3$ = edit distance.

Why Edit Distance Is a Distance Measure

- d(x,x) = 0 because 0 edits suffice.
- d(x,y) = d(y,x) because insert/delete are inverses of each other.
- d(x,y) ≥ 0: no notion of negative edits.
- Triangle inequality: changing *x* to *z* and then to *y* is one way to change *x* to *y*.

Variant Edit Distances

- Allow insert, delete, and *mutate*.
  - Change one character into another.
- Minimum number of inserts, deletes, and mutates also forms a distance measure.
- Ditto for any set of operations on strings.
  - Example: substring reversal OK for DNA sequences

Hamming Distance

- *Hamming distance* is the number of positions in which bit-vectors differ.
- Example: $p_1 = 10101$; $p_2 = 10011$.
- $d(p_1, p_2) = 2$ because the bit-vectors differ in the 3^{rd} and 4^{th} positions.

Why Hamming Distance Is a Distance Measure

- d(x,x) = 0 since no positions differ.
- d(x,y) = d(y,x) by symmetry of "different from."
- d(x,y) $\geq$ 0 since strings cannot differ in a negative number of positions.
- Triangle inequality: changing *x* to *z* and then to *y* is one way to change *x* to *y*.

Acknowledgements

- Some slides borrowed from
  - Anand Rajaraman and Jeff Ullman
  - Richard Socher
  - Tomas Mikolov
  - Dan Jurafsky and James Martin