BBM 495
WORD EMBEDDINGS

LECTURER: BURCU CAN

2019-2020 SPRING

Word Similarity & Relatedness

- How similar is **pizza** to **pasta**?
- How related is **pizza** to **Italy**?

- **Representing words as vectors** allows easy computation of similarity

Window-based co-occurrence matrix

- Window length 1 (more common: 5 - 10)
- Symmetric (irrelevant whether left or right context)
- Example corpus:
  - I like deep learning.
  - I like NLP.
  - I enjoy flying.

Window-based co-occurrence matrix

- Example corpus:
  - I like deep learning.
  - I like NLP.
  - I enjoy flying.

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Problems with simple co-occurrence vectors

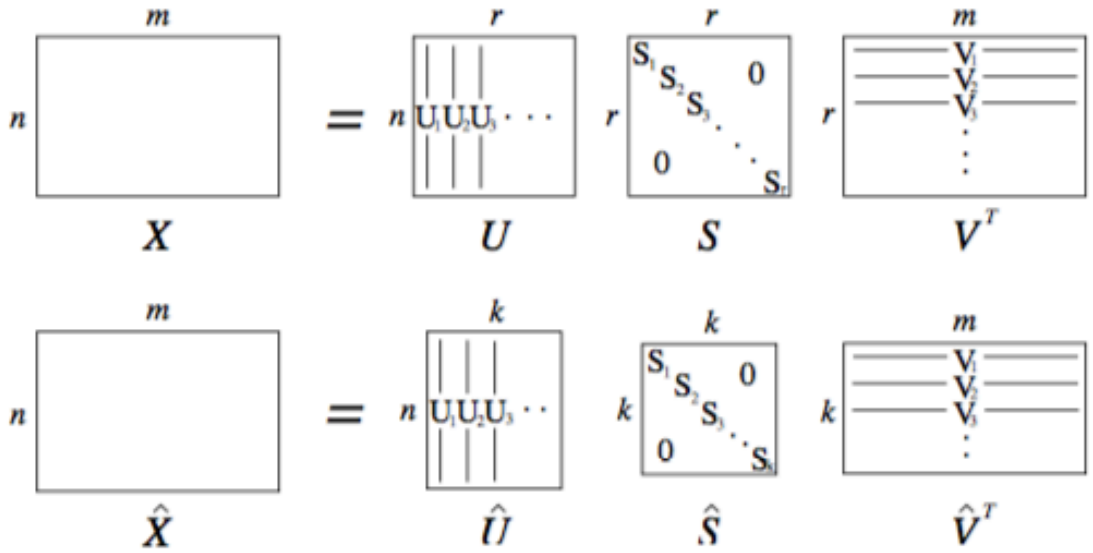
- Increase in size with vocabulary
- Very high dimensional: require a lot of storage
- Subsequent classification models have sparsity issues
- Models are less robust

Solution: low dimensional vectors

- Idea: store "most" of the important information in a fixed, small number of dimensions: a dense vector
- Usually 25 – 1000 dimensions, similar to word2vec
- How to reduce the dimensionality?

Method 1: Dimensionality Reduction on X

Singular Value Decomposition of co-occurrence matrix *X*.



$\hat{X}$ is the best rank *k* approximation to *X*, in terms of least squares.

Simple SVD word vectors in Python

Corpus:
I like deep learning. I like NLP. I enjoy flying.

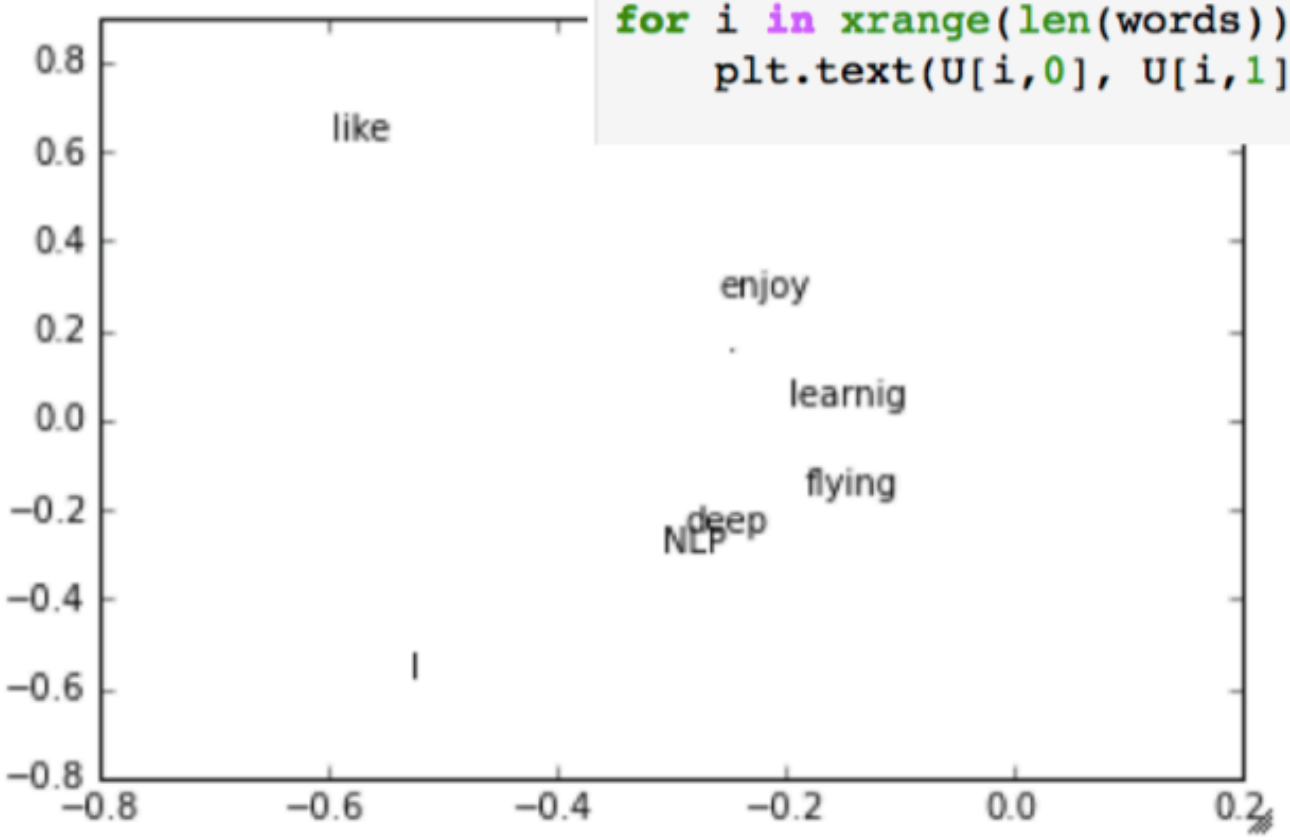
```
import numpy as np
la = np.linalg
words = ["I", "like", "enjoy",
         "deep","learnig","NLP","flying","."]
X = np.array([[0,2,1,0,0,0,0,0],
              [2,0,0,1,0,1,0,0],
              [1,0,0,0,0,0,1,0],
              [0,1,0,0,1,0,0,0],
              [0,0,0,1,0,0,0,1],
              [0,1,0,0,0,0,0,1],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

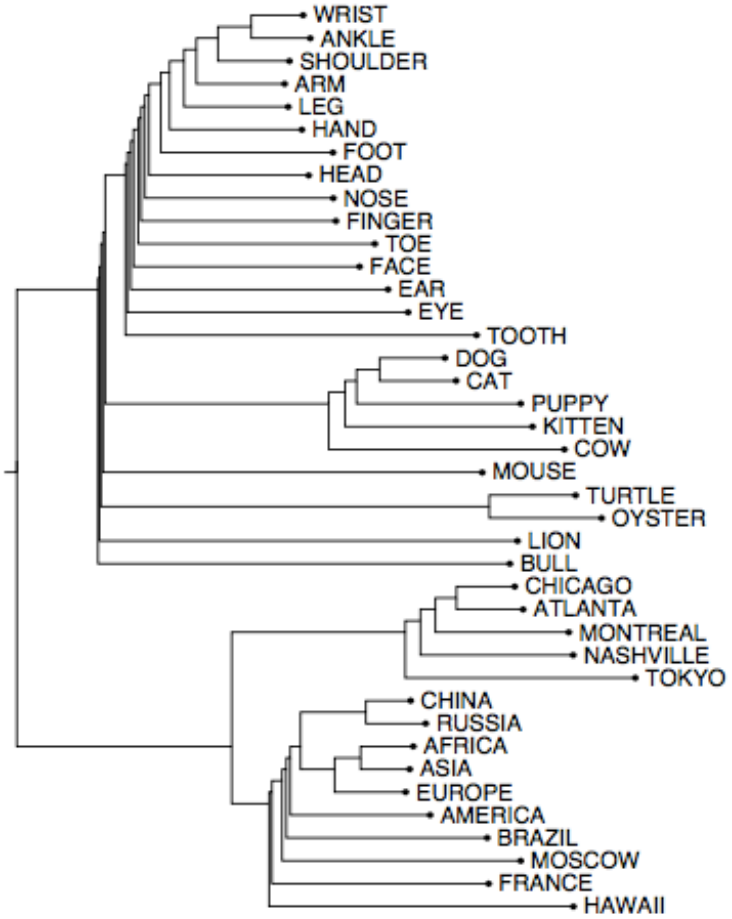
Simple SVD word vectors in Python

Corpus: I like deep learning. I like NLP. I enjoy flying.
Printing first two columns of U corresponding to the 2 biggest singular values

```
for i in xrange(len(words)):
    plt.text(U[i,0], U[i,1], words[i])
```

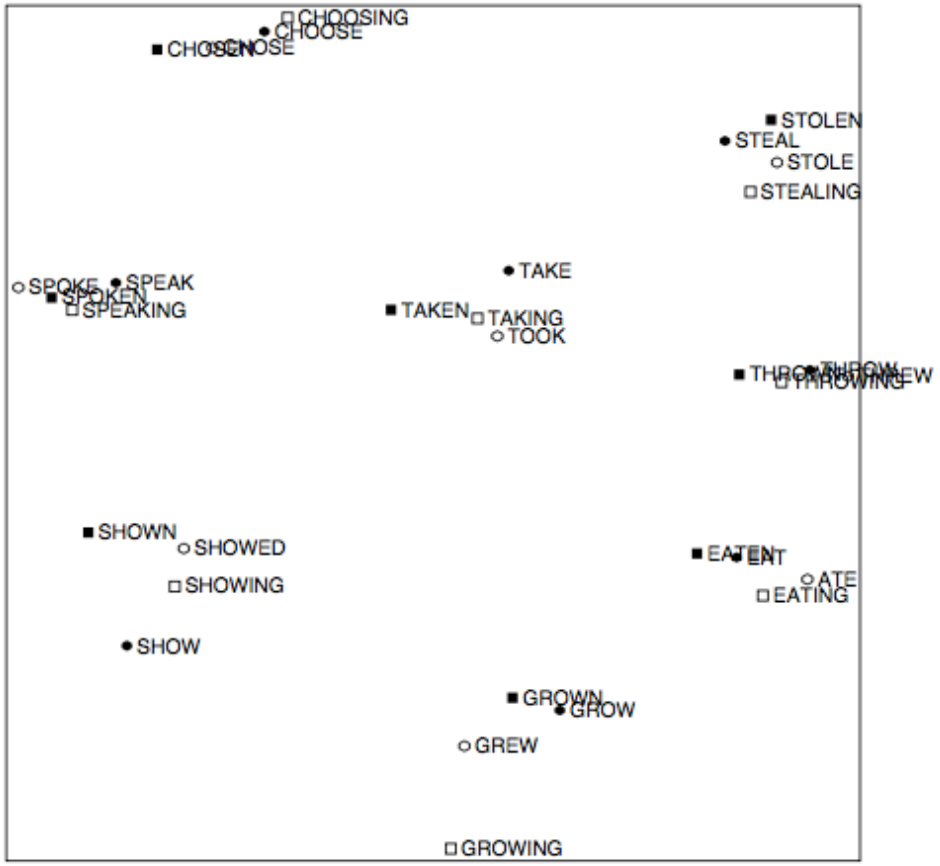


Interesting semantic patterns emerge in the vectors



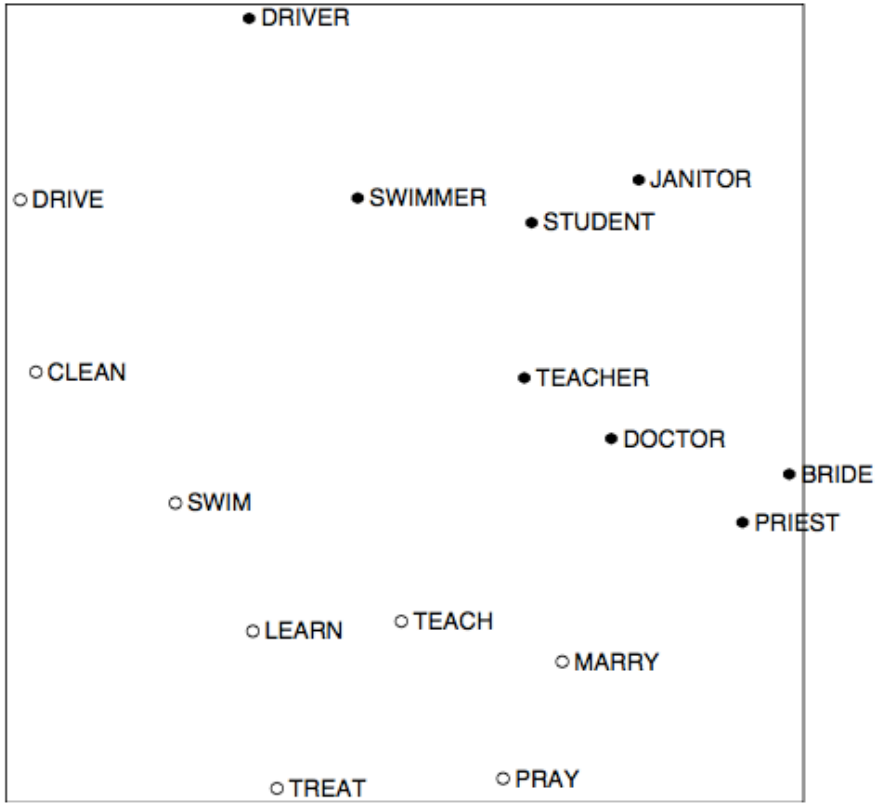
An Improved Model of Semantic Similarity Based on Lexical Co-Occurrence
Rohde et al. 2005

Interesting syntactic patterns emerge in the vectors



An Improved Model of Semantic Similarity Based on Lexical Co-Occurrence
Rohde et al. 2005

Interesting syntactic patterns emerge in the vectors



An Improved Model of Semantic Similarity Based on Lexical Co-Occurrence
Rohde et al. 2005

Problems with SVD

Computational cost scales quadratically for n x m matrix:

$O(mn^2)$ flops (when n<m)

→ Bad for millions of words or documents

Hard to incorporate new words or documents

Problems with resources like WordNet

- Great as a resource but missing nuance
  - e.g. "proficient" is listed as a synonym for "good". This is only correct in some contexts.
- Missing new meanings of words
  - e.g. wicked, badass, nifty, wizard, genius, ninja, bombest
  - Impossible to keep up-to-date!
- Subjective
- Requires human labor to create and adapt
- Hard to compute accurate word similarity →

Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols:
hotel, conference, motel

Means one 1, the rest 0s

Words can be represented by one-hot vectors:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

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

Problems with words as discrete symbols

Example: in web search, if user searches for "Seattle motel", we would like to match documents containing "Seattle hotel".

But:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

These two vectors are orthogonal.

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

Solution:

- Could rely on WordNet's list of synonyms to get similarity?
- **Instead: learn to encode similarity in the vectors themselves**

Representing words by their context

- Core idea: **A word's meaning is given by the words that frequently appear close-by**
  - *"You shall know a word by the company it keeps"* (J. R. Firth 1957: 11)
  - One of the most successful ideas of modern statistical NLP!
- When a word *w* appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).
- Use the many contexts of *w* to build up a representation of *w*

…government debt problems turning into ***banking*** *crises as happened in 2009…*
…saying that Europe needs unified ***banking*** *regulation to replace the hodgepodge…*
…India has just given its ***banking*** *system a shot in the arm…*

These context words will represent ***banking***

Word vectors

We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts.

$$\textit{linguistics} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

Note: word vectors are sometimes called word embeddings or word representations.

Sparse vs dense vectors

- Why dense vectors?
  - 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; this fails to capture similarity between a word with *car* as a neighbor and a word with *automobile* as a neighbor

Word vector

=

word representation

=

word embeddings

Prediction-based models: An alternative way to get dense vectors

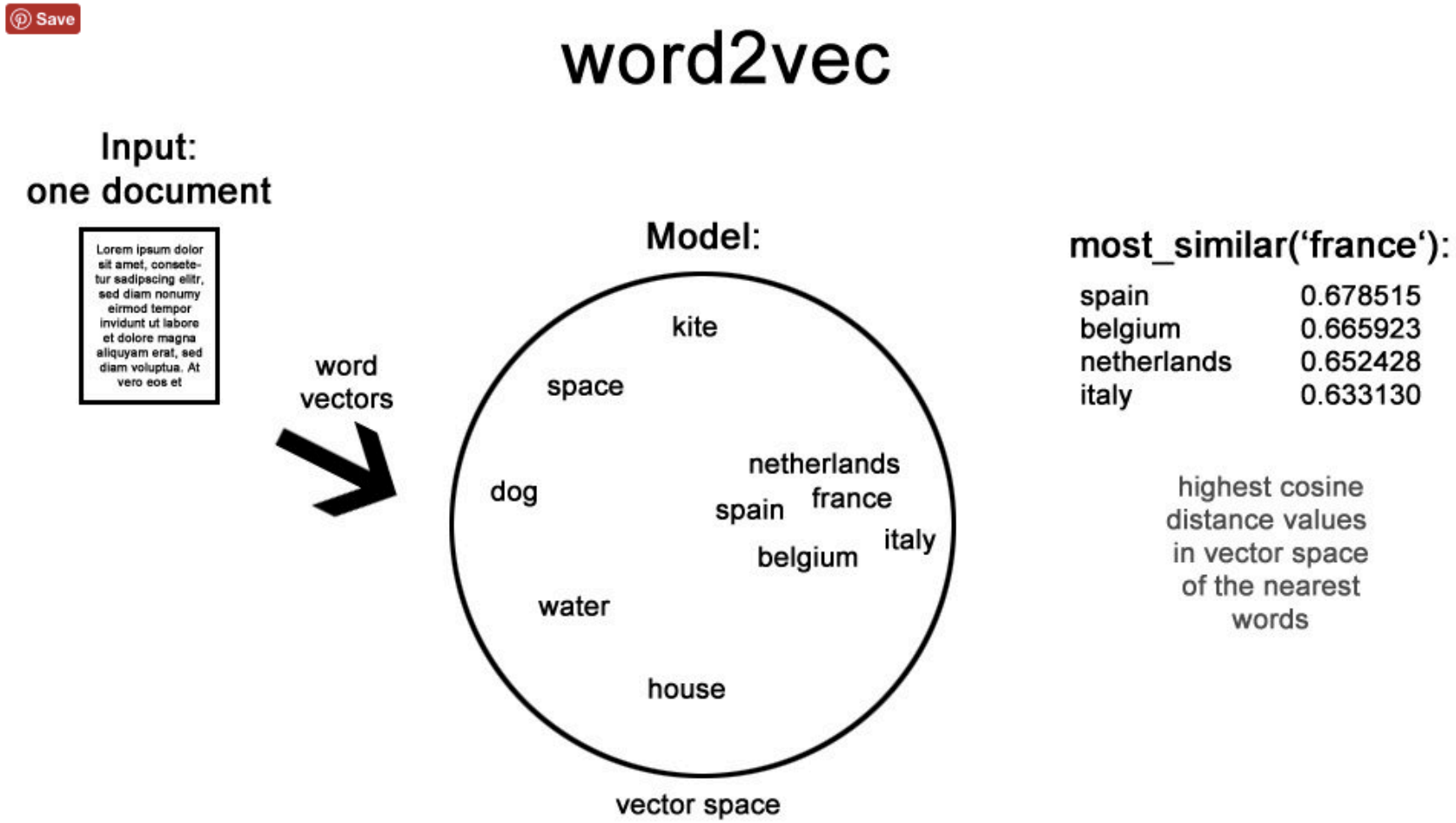
- **Skip-gram** (Mikolov et al. 2013a) **CBOW** (Mikolov et al. 2013b)
- Learn embeddings as part of the process of word prediction.
- Train a neural network to predict neighboring words
  - Inspired by **neural net language models**.
  - In so doing, learn dense embeddings for the words in the training corpus.
- Advantages:
  - Fast, easy to train (much faster than SVD)
  - Available online in the `word2vec` package
  - Including sets of pretrained embeddings!

Word2vec: Overview

Word2vec (Mikolov et al. 2013) is a framework for learning word vectors. Idea:

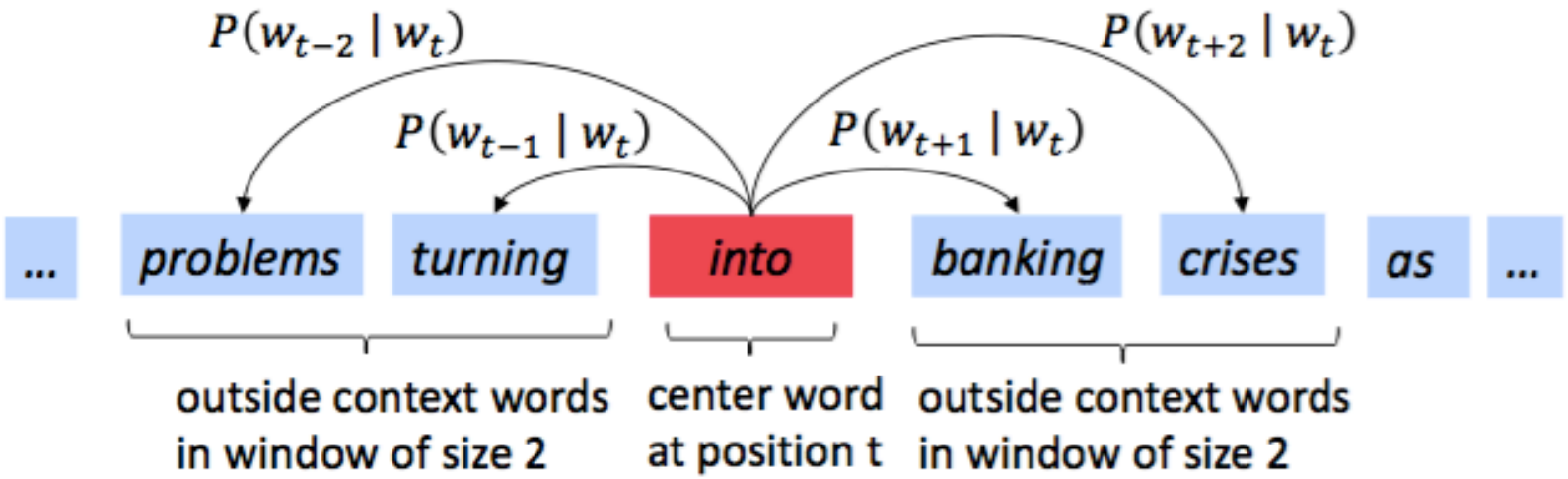
- We have a large corpus of text
- Every word in a fixed vocabulary is represented by a vector
- Go through each position *t* in the text, which has a center word *c* and context ("outside") words *o*
- Use the similarity of the word vectors for *c* and *o* to calculate the probability of *o* given *c* (or vice versa)
- Keep adjusting the word vectors to maximize this probability

Word2vec: Overview



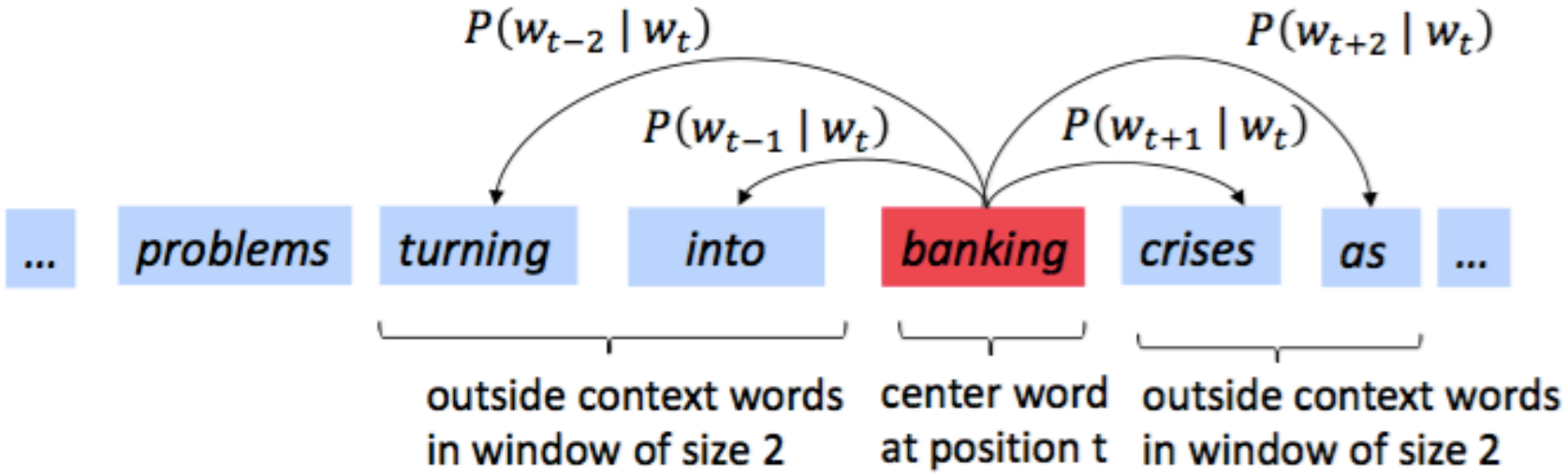
Word2vec: Overview

- Example windows and process for computing $P(w_{t+j} \mid w_t)$



Word2vec: Overview

- Example windows and process for computing $P(w_{t+j} \mid w_t)$

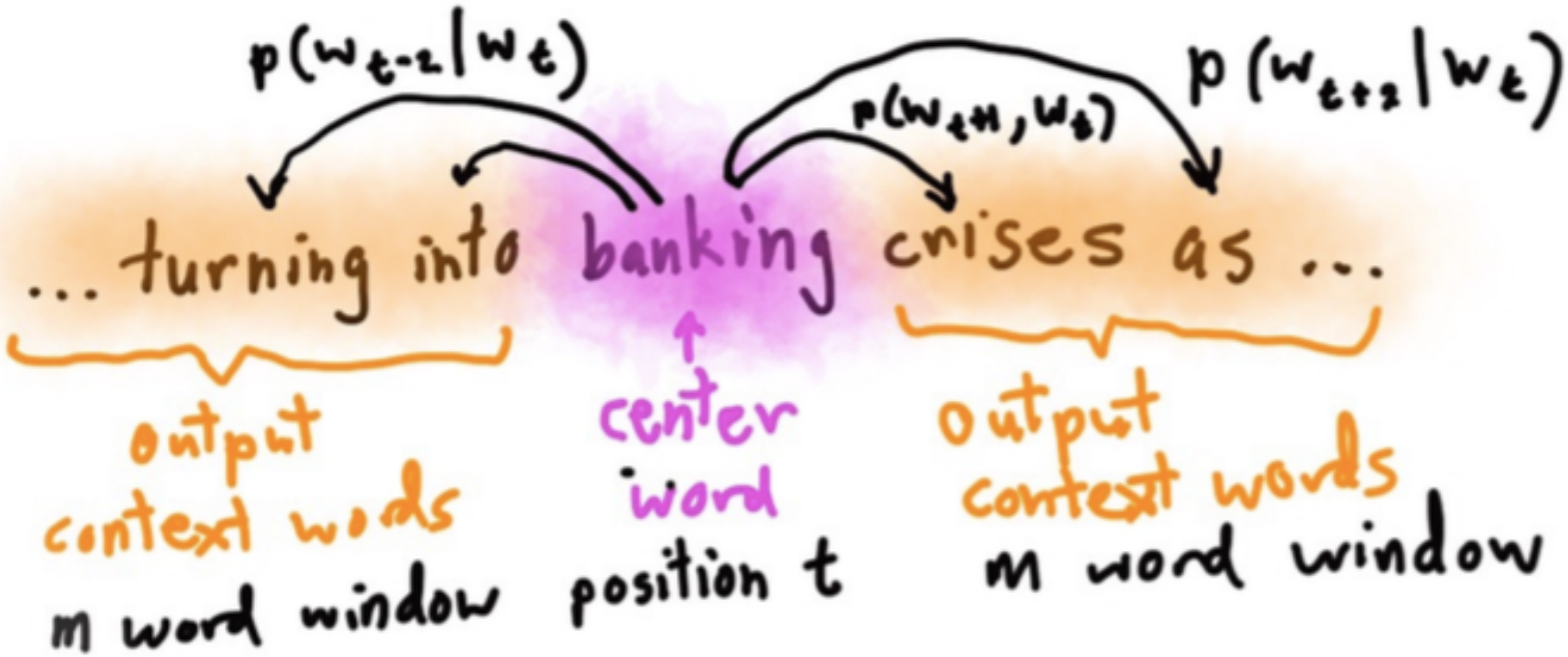


Skip-grams

- Predict each neighboring word
  - in a context window of 2*C* words
  - from the current word.
- So for C=2, we are given word w_t and predicting these 4 words:

$$[w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}]$$

Word2vec skip-gram prediction



Basic idea of learning neural network word embeddings (Predict!)

We define a model that predicts between a center word w_t and context words in terms of word vectors, e.g.,

$$p(\text{context}|w_t) = \ldots$$

which has a loss function, e.g.,

$$J = 1 - p(w_{-t}|w_t)$$

We look at **many** positions t in a big language corpus

We keep adjusting the vector representations of words to minimize this loss

Word2vec: objective function

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size m, given center word w_j.

Likelihood = $$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$$

θ is all variables to be optimized

sometimes called *cost* or *loss* function

The objective function $J(\theta)$ is the (average) negative log likelihood:

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

Minimizing objective function $\Leftrightarrow$ Maximizing predictive accuracy

Word2vec: objective function

- We want to minimize the objective function:

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

- Question: How to calculate $P(w_{t+j} \mid w_t; \theta)$?
- Answer: We will *use two* vectors per word *w*:
  - v_w when *w* is a center word
  - u_w when *w* is a context word
- Then for a center word *c* and a context word *o*:

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w\in V}\exp(u_w^T v_c)}$$

Similarity is computed from dot product

- Remember: two vectors are similar if they have a high dot product
  - Cosine is just a normalized dot product
- So:
  - Similarity (o,c) $\propto u_o \cdot V_c$
- We will need to normalize to get a probability

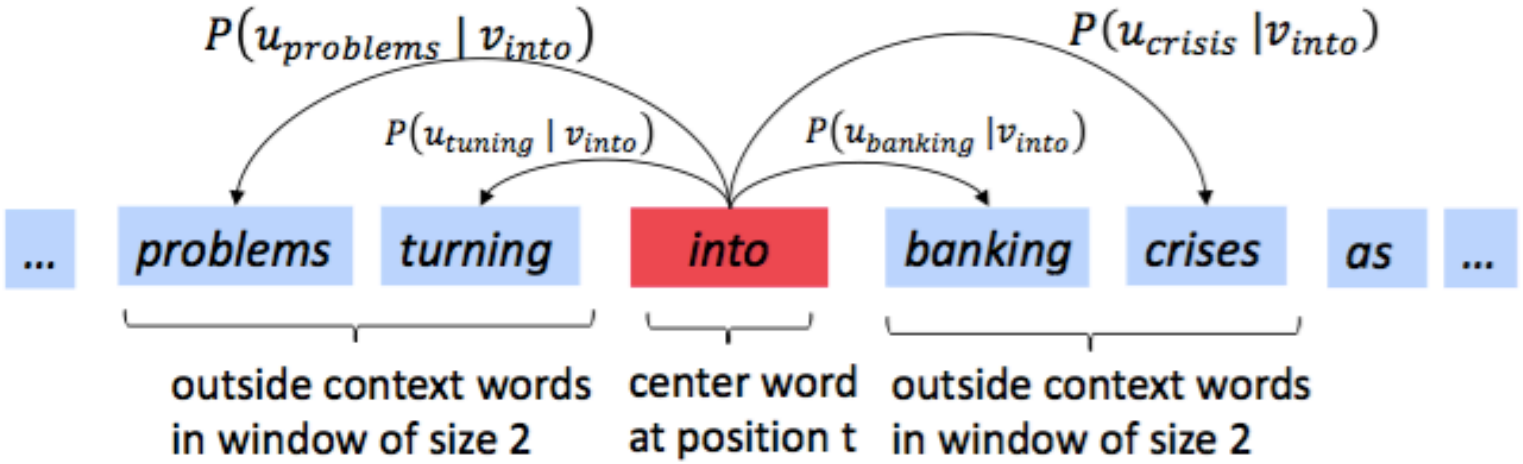
Turning dot products into probabilities

- We use softmax to turn into probabilities:

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{V} \exp(u_w^T v_c)}$$

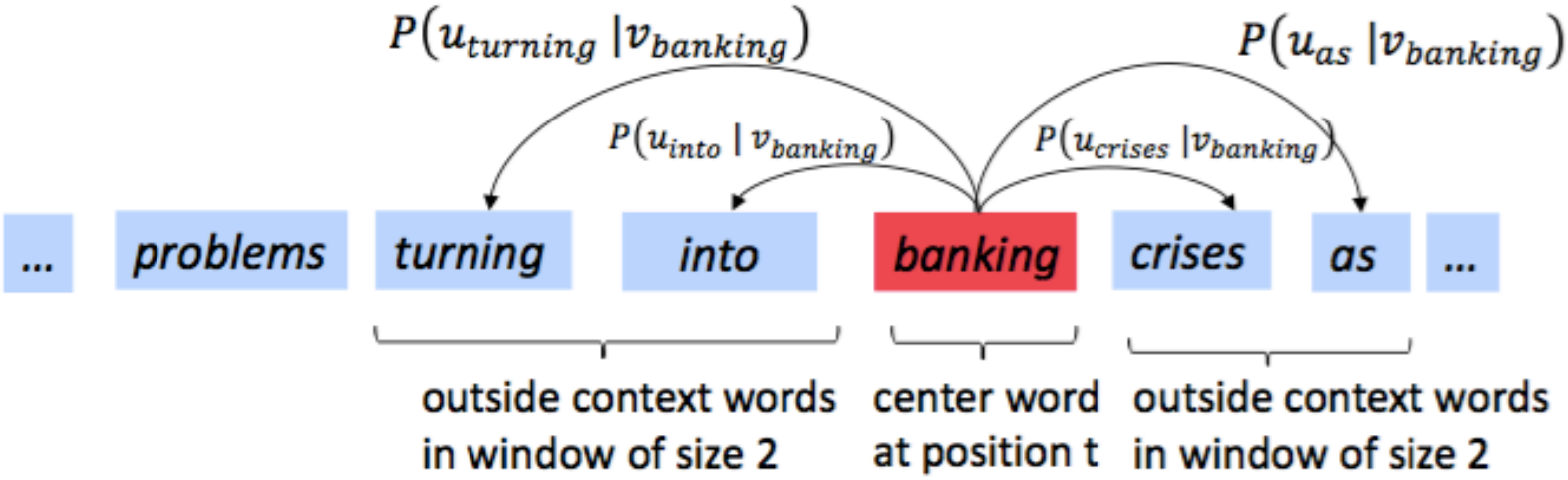
Word2Vec Overview with Vectors

- Example windows and process for computing $P(w_{t+j} \mid w_t)$
- $P(u_{problems} \mid v_{into})$ short for $P(problems \mid into\ ; u_{problems}, v_{into}, \theta)$



Word2Vec Overview with Vectors

- Example windows and process for computing $P(w_{t+j} \mid w_t)$



Details of Word2Vec

For $p(w_{t+j}|w_t)$ we choose:

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{V} \exp(u_w^T v_c)}$$

where *o* is the outside (or output) word index, *c* is the center word index, v_c and u_o are the "center" and "outside" vectors for word indices *c* and *o*

Softmax using word *c* to obtain probability of word *o*

Co-occurring words are driven to have similar vectors

Word2vec: prediction function

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Dot product compares similarity of *o* and *c*.
Larger dot product = larger probability

After taking exponent,
normalize over entire vocabulary

- This is an example of the **softmax function** $\mathbb{R}^n \to \mathbb{R}^n$

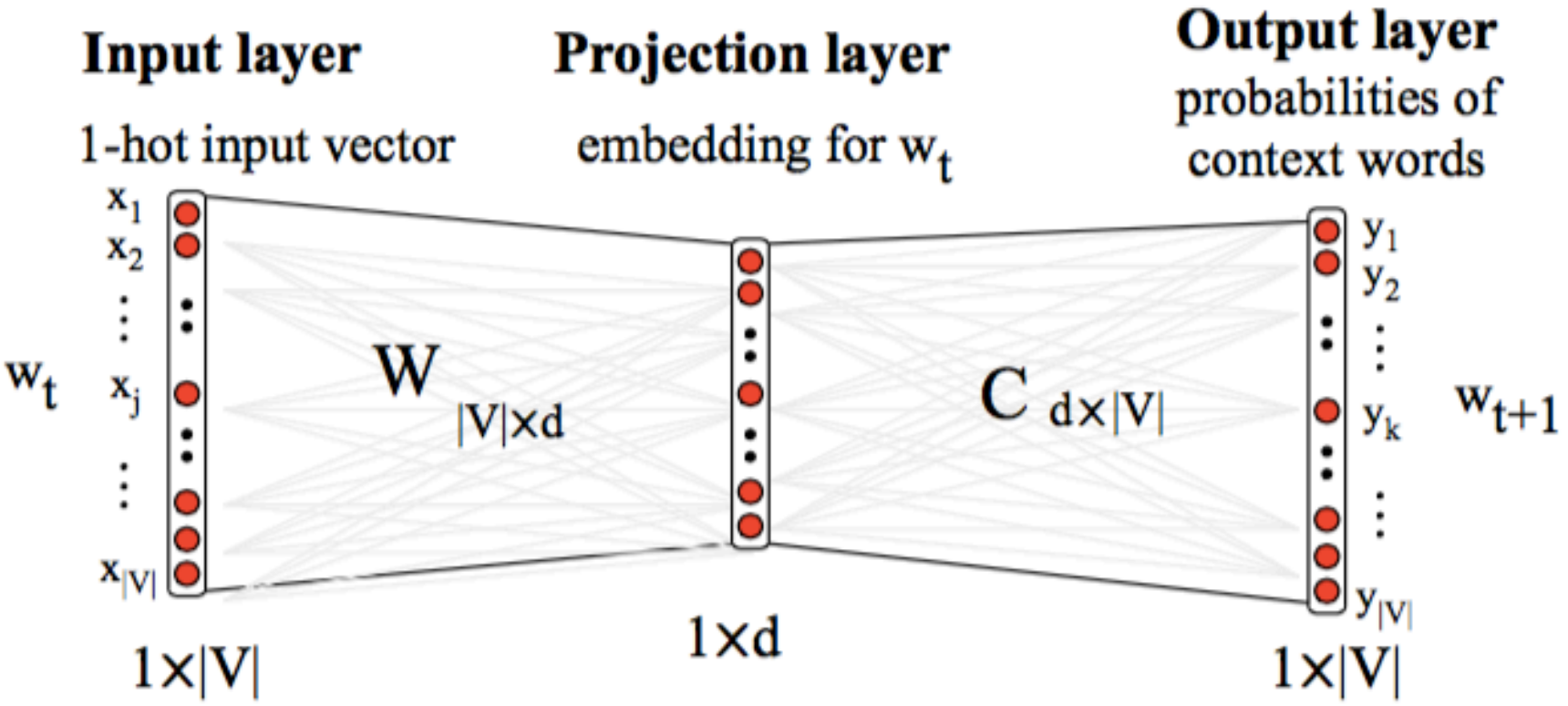
$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i$$

- The softmax function maps arbitrary values x_i to a probability distribution p_i
  - "max" because amplifies probability of largest x_i
  - "soft" because still assigns some probability to smaller x_i
  - Frequently used in Deep Learning

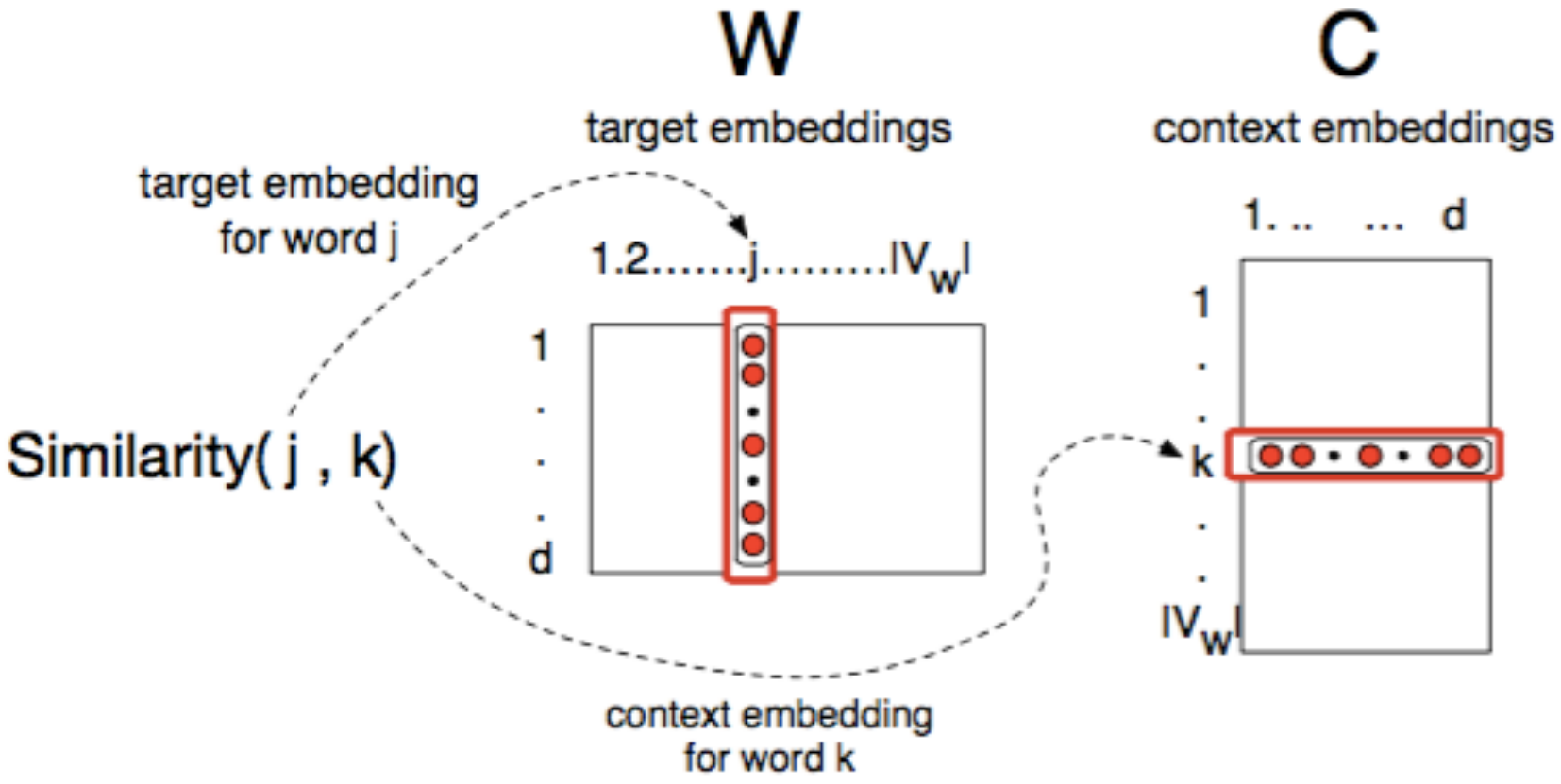
Learning

- Start with some initial embeddings (e.g., random)
- iteratively make the embeddings for a word
  - more like the embeddings of its neighbors
  - less like the embeddings of other words.

Visualizing error backpropagation

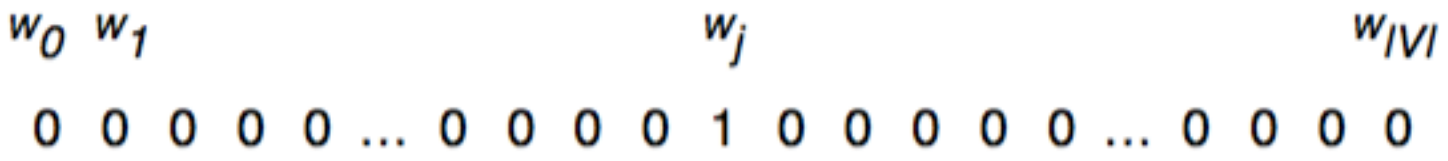


Similarity as dot product



One-hot vectors

- A vector of length |V|
- 1 for the target word and 0 for other words
- So if “popsicle” is vocabulary word 5
- The **one-hot vector** is
- [0,0,0,0,1,0,0,0,0…….0]



To train the model: Compute all vector gradients!

- Recall: θ represents all model parameters, in one long vector
- In our case with *d*-dimensional vectors and *V*-many words:

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

- Remember: every word has two vectors
- We then optimize these parameters

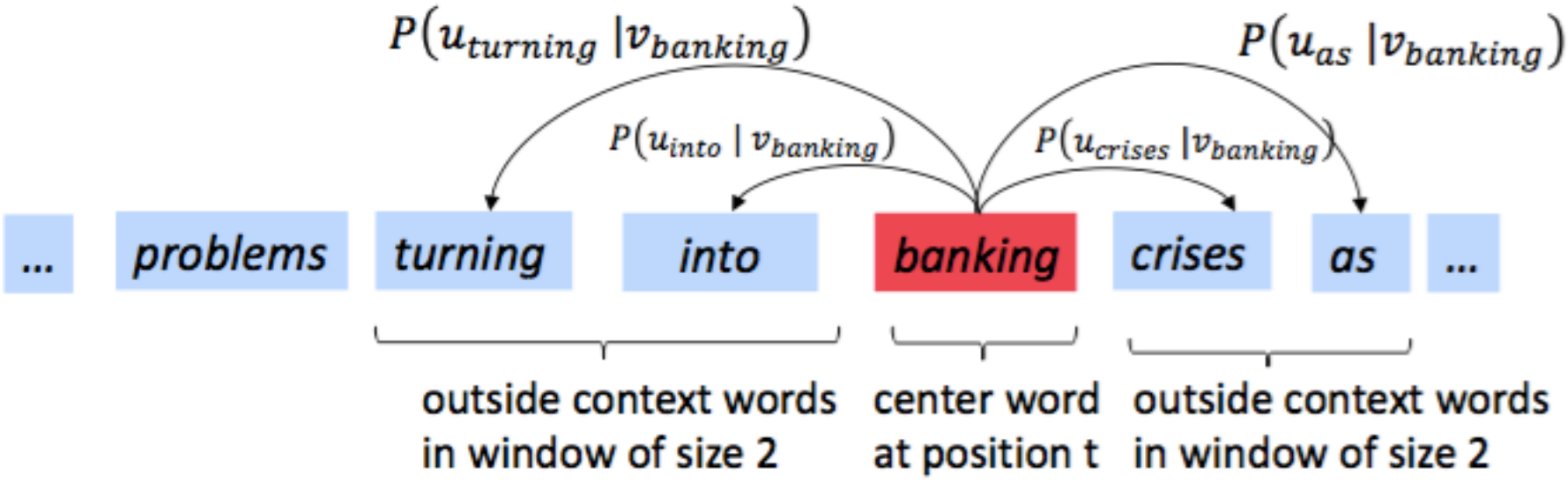
Vector gradients

- Take gradients at each window

$$\nabla_\theta J_t(\theta) = \begin{bmatrix} 0 \\ \vdots \\ \nabla_{v_{like}} \\ \vdots \\ 0 \\ \nabla_{u_I} \\ \vdots \\ \nabla_{u_{learning}} \\ \vdots \end{bmatrix} \in \mathbb{R}^{2dV}$$

Calculating all gradients

- Go through gradient for each center vector v in a window
- In each window, we will compute updates for all parameters that are being used in that window. For example:

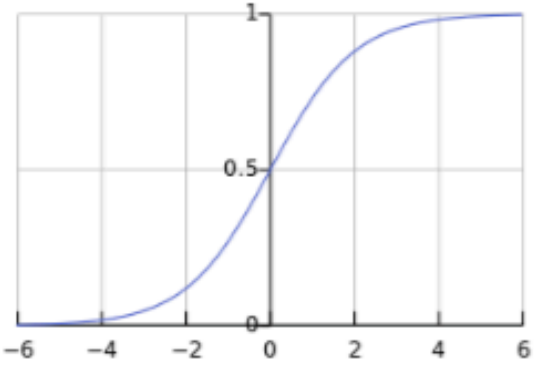


Gradient

- From paper: "Distributed Representations of Words and Phrases and their Compositionality" (Mikolov et al. 2013)
- Overall objective function (they maximize): $J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J_t(\theta)$

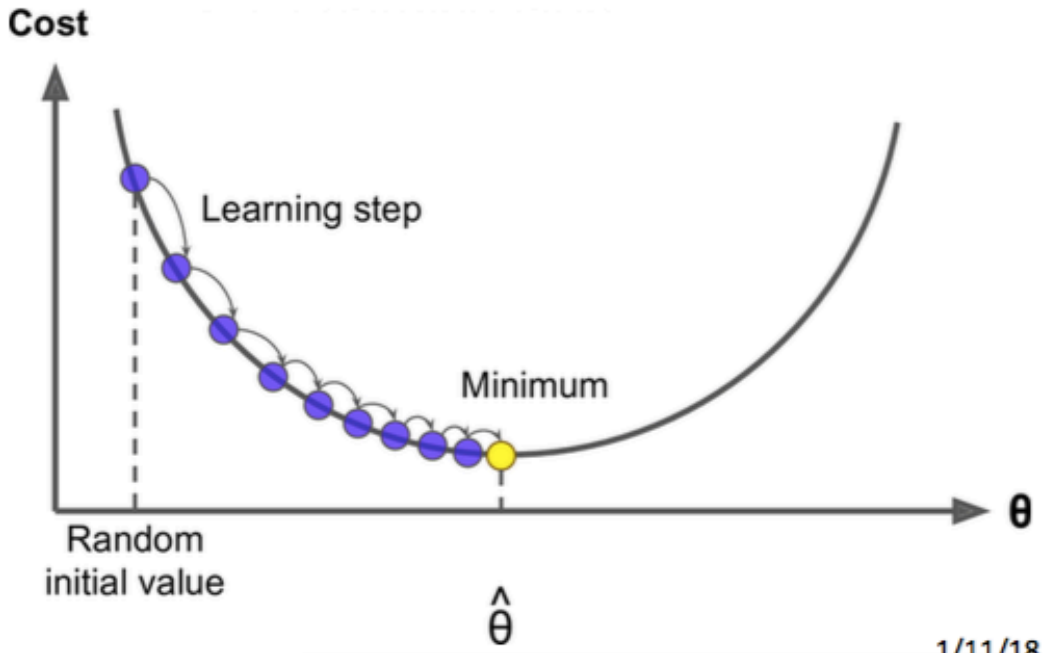
$$J_t(\theta) = \log \sigma\left(u_o^T v_c\right) + \sum_{i=1}^{k} \mathbb{E}_{j\sim P(w)}\left[\log \sigma\left(-u_j^T v_c\right)\right]$$

- The sigmoid function! $\sigma(x) = \frac{1}{1+e^{-x}}$ (we'll become good friends soon)
- So we maximize the probability of two words co-occurring in first log →



Gradient Descent

- We have a cost function $J(\theta)$ we want to minimize
- **Gradient Descent** is an algorithm to minimize $J(\theta)$
- Idea: for current value of θ, calculate gradient of $J(\theta)$, then take small step in direction of negative gradient. Repeat.



Note: Our objectives are not convex like this :(

Gradient Descent

- Update equation (in matrix notation):

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$$

α = *step size* or *learning rate*

- Update equation (for single parameter):

$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial}{\partial \theta_j^{old}} J(\theta)$$

- Algorithm:

```
while True:
    theta_grad = evaluate_gradient(J,corpus,theta)
    theta = theta - alpha * theta_grad
```

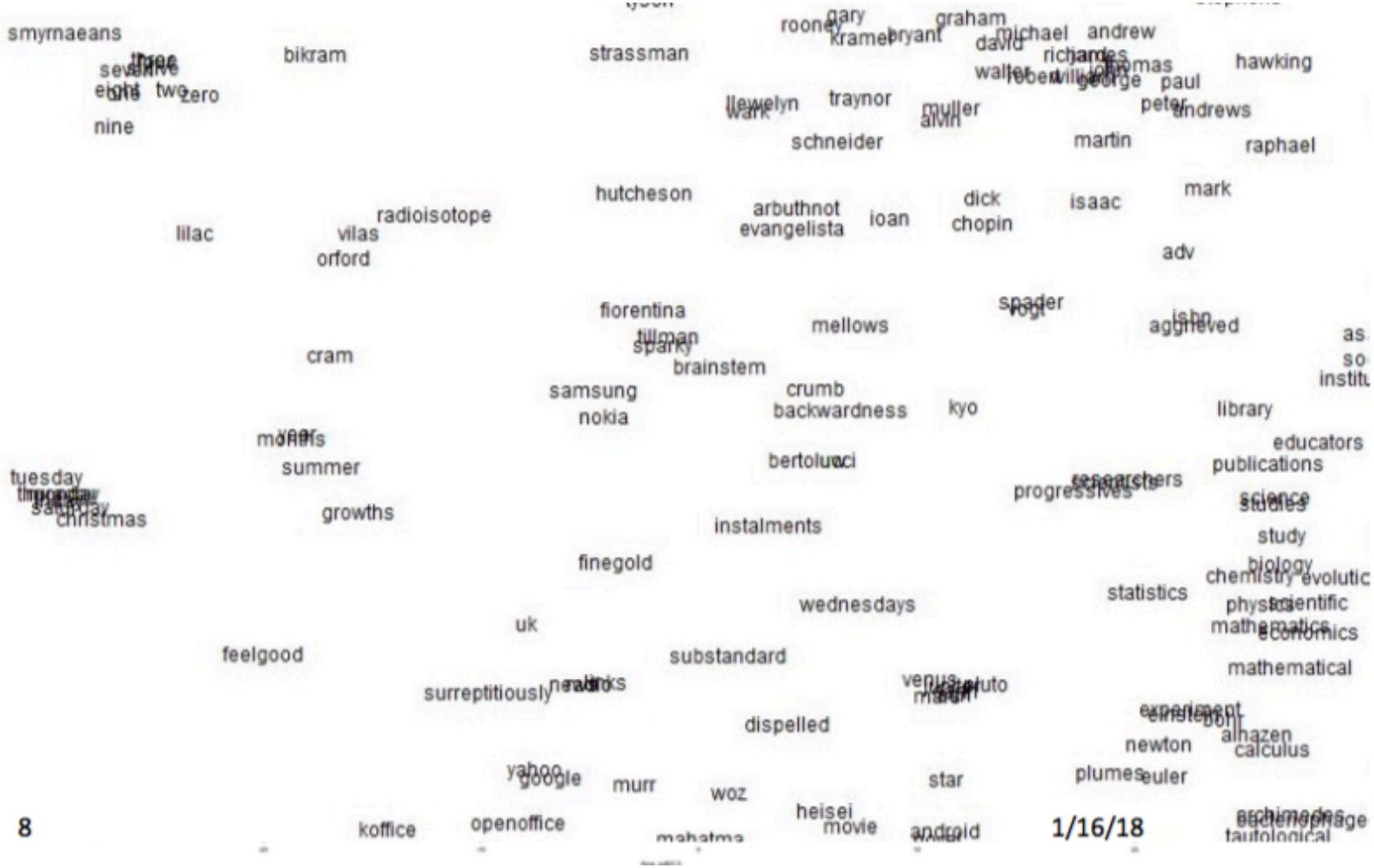
Word meaning as a vector

The result is a dense vector for each word type, chosen so that it is good at predicting other words appearing in its context

… those other words also being represented by vectors

$$\textit{currency} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

Word2vec improves objective function by putting similar words nearby in space



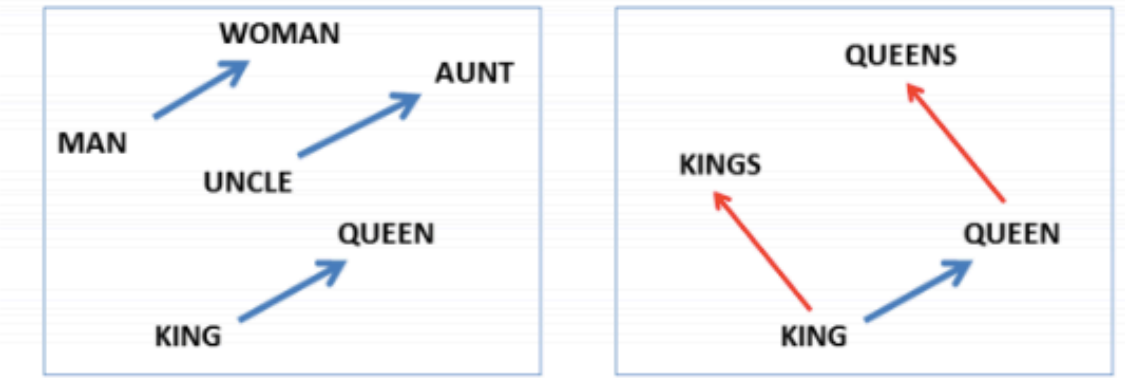
Properties of embeddings

- Nearest words to some embeddings (Mikolov et al. 20131)

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

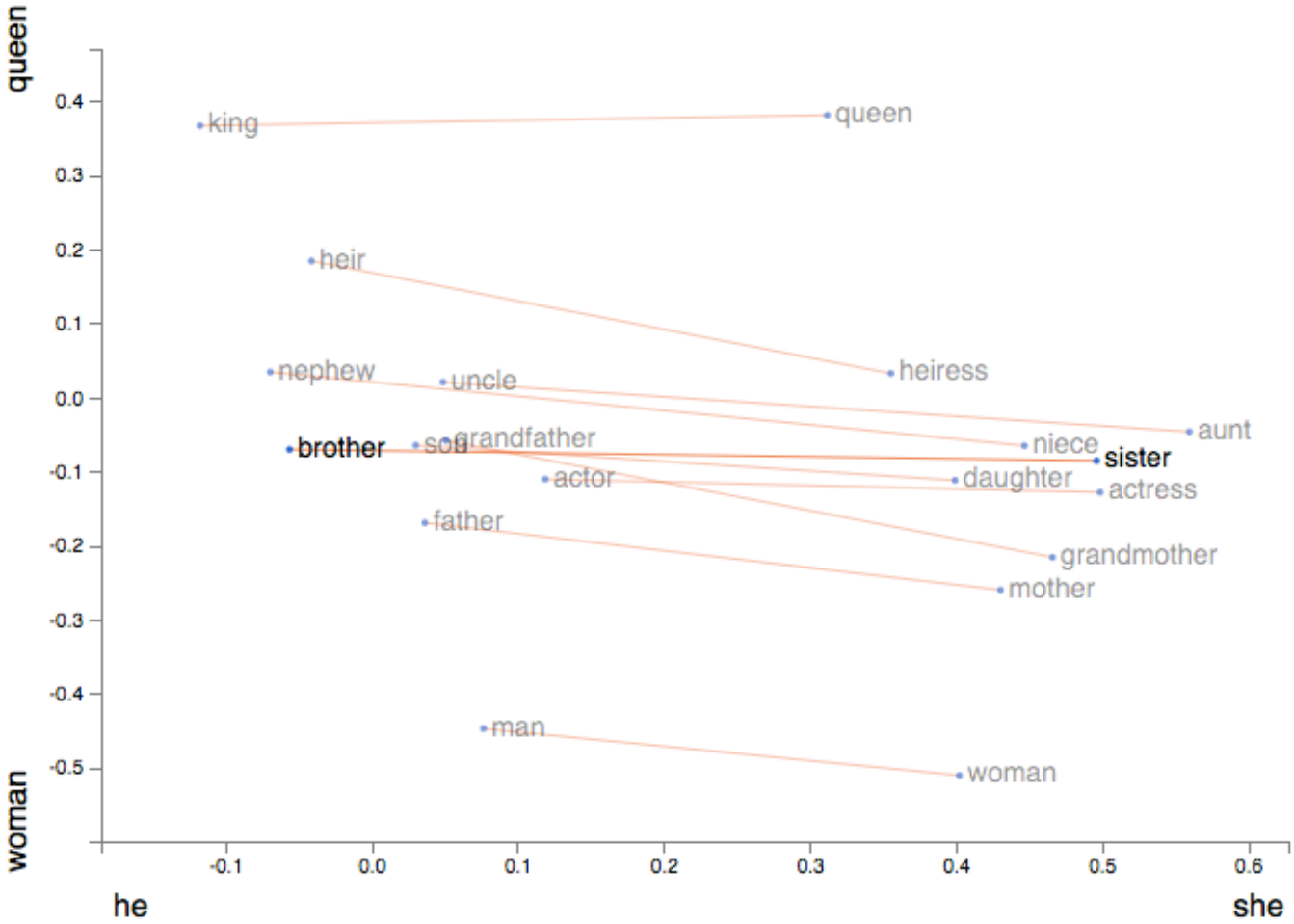
Embeddings capture relational meaning

- We can do nearest neighbor search around result of vector operation "king – man + woman" and obtain "queen"

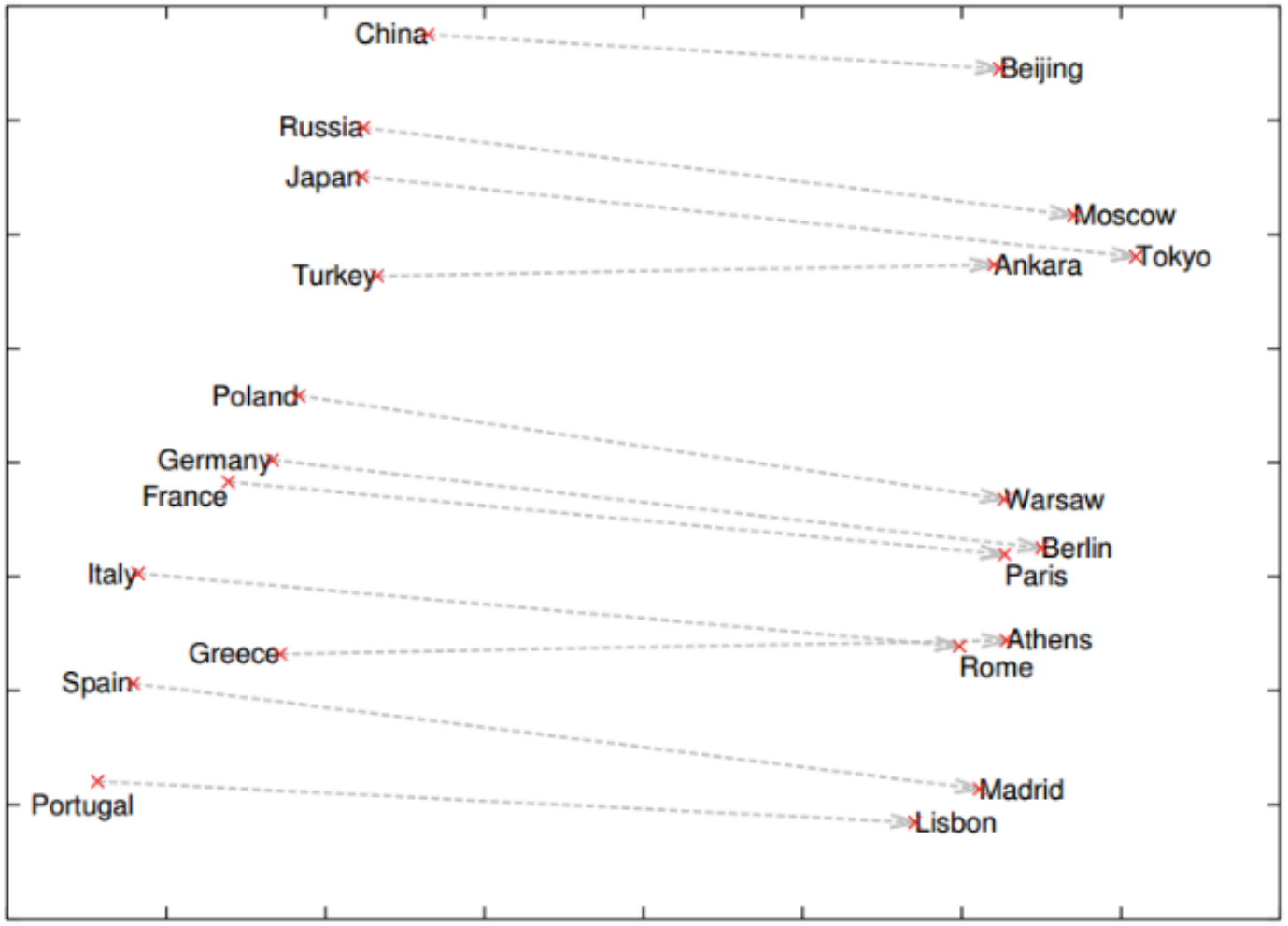


Linguistic regularities in continuous space word representations (Mikolov et al, 2013)

Trained word vectors



Trained word vectors



Neural Networks for NLP, Tomas Mikolov

Meaning in vectors

About word2vec

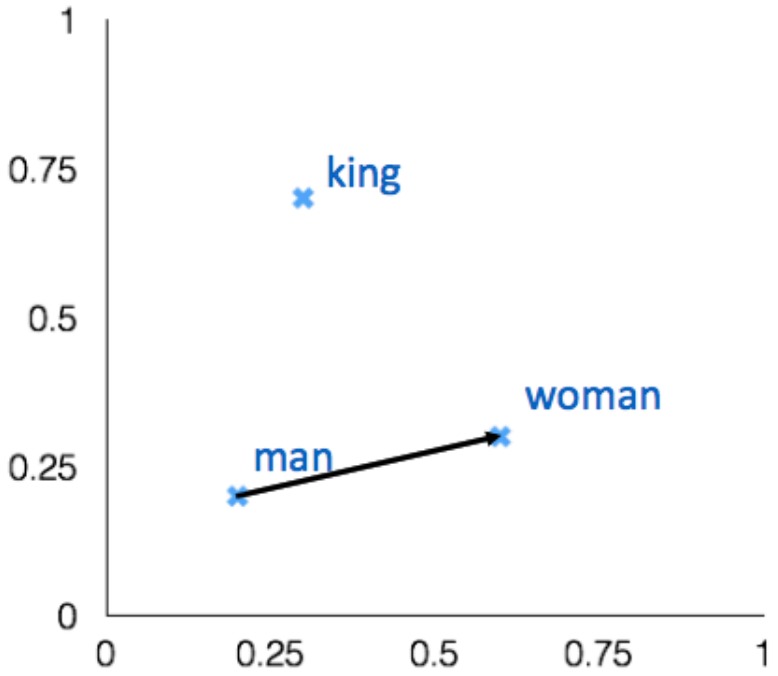
- Word Vector Analogies

a:b :: c:?

man:woman :: king:?

$$d = \arg\max_i \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||}$$

- Evaluate word vectors by how well their cosine distance after addition captures intuitive semantic and syntactic analogy questions
- Discarding the input words from the search!
- Problem: What if the information is there but not linear?



Intrinsic word vector evaluation

- Word vector distances and their correlation with human judgments
- Example dataset: WordSim353
  http://www.cs.technion.ac.il/~gabr/resources/data/wordsim353/

Word 1	Word 2	Human (mean)
tiger	cat	7.35
tiger	tiger	10.00
book	paper	7.46
computer	internet	7.58
plane	car	5.77
professor	doctor	6.62
stock	phone	1.62
stock	CD	1.31
stock	jaguar	0.92

Intrinsic word vector evaluation

- Word Vector Analogies: Syntactic and **Semantic** examples from

: capital-world
Abuja Nigeria Accra Ghana
Abuja Nigeria Algiers Algeria
Abuja Nigeria Amman Jordan
Abuja Nigeria Ankara Turkey
Abuja Nigeria Antananarivo Madagascar
Abuja Nigeria Apia Samoa
Abuja Nigeria Ashgabat Turkmenistan
Abuja Nigeria Asmara Eritrea
Abuja Nigeria Astana Kazakhstan

problem: can change

Intrinsic word vector evaluation

- Word Vector Analogies: **Syntactic** and Semantic examples from

: gram4-superlative
bad worst big biggest
bad worst bright brightest
bad worst cold coldest
bad worst cool coolest
bad worst dark darkest
bad worst easy easiest
bad worst fast fastest
bad worst good best
bad worst great greatest

Intrinsic word vector evaluation

- Word Vector Analogies: **Syntactic** and Semantic examples from

: gram7-past-tense
dancing danced decreasing decreased
dancing danced describing described
dancing danced enhancing enhanced
dancing danced falling fell
dancing danced feeding fed
dancing danced flying flew
dancing danced generating generated
dancing danced going went
dancing danced hiding hid
dancing danced hitting hit

References

- Dense Vectors, Dan Jurafsky
- Representation for Language: from Word Embeddings to Sentence Meanings, Christopher Manning, Stanford University, 2017
- Natural Language Processing with Deep Learning, Richard Socher, Stanford University
- More Word Vectors, Richard Socher, Stanford University
- Improving Distributional Similarity with Lessons Learned from Word Embeddings, Omer Levy,