

Lecture 7: Word Embeddings (il)

Instructor: Stephanie Schoch

CSCI 680: Natural Language Processing
Department of Computer Science, College of William & Mary
September 14, 2026



Lecture Outline

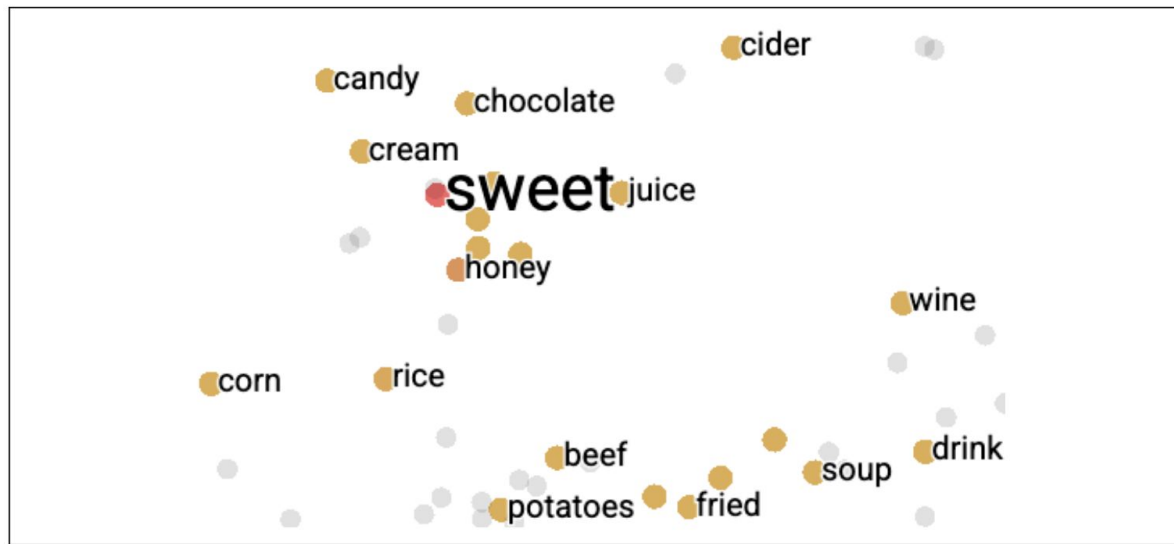
- Word Embeddings
 - Cosine Similarity
 - Count-Based Embeddings
 - Weighting: TF-IDF & PMI
 - Word2vec

Word Embeddings

Words as Vectors: Word Embeddings

- Represent a word as a point in a multidimensional semantic space
 - Space itself constructed from distribution of word neighbors
- Called an "embedding" because it's embedded into a space
- Fine-grained model of meaning for similarity

**Every modern
NLP algorithm
uses embeddings
as the
representations
of word meaning!**



Two Kinds of Word Embeddings

1. Simple count embeddings

- **Sparse** vectors
- Words are represented by the counts of nearby words
- tf-idf model of weighting those counts.

2. Word2vec

- **Dense** vectors
- Representation is created by training a classifier to **predict** whether a word is likely to appear nearby
- Later we'll discuss extensions called **contextual embeddings**

Word Co-Occurrence Matrix: A Larger Matrix

**Context
Window**
(± 4)

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

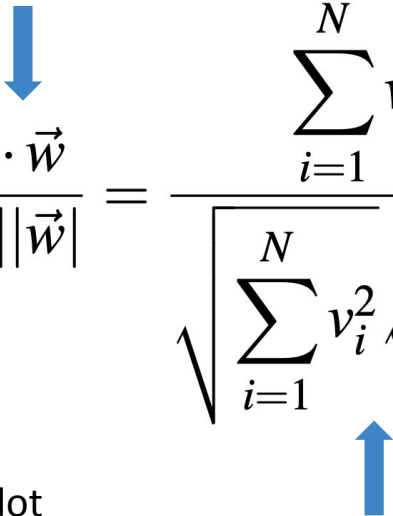
	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...

- Word context matrix is $|V| \times |V|$
- This could be 50,000 x 50,000

Most of these numbers are zero: **sparse vectors**

Computing Word Similarity: Cosine Similarity

- Dot product tends to be high when two vectors have large values in same dimension
- But: favors long vectors
 - Frequent words (e.g. functional words) have long vectors


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

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

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

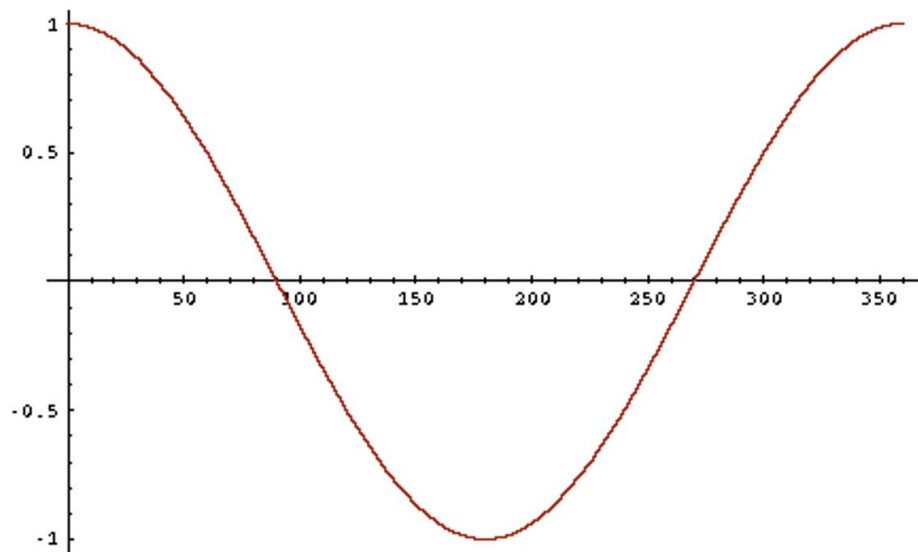
Based on the definition of dot product between two vectors.

Normalize by length

Computing Word Similarity: Cosine Similarity

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

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



Cosine Example

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

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

Cosine Example

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

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

$$\cos(\text{cherry}, \text{information}) =$$

$$\frac{442 * 5 + 8 * 3982 + 2 * 3325}{\sqrt{442^2 + 8^2 + 2^2} \sqrt{5^2 + 3982^2 + 3325^2}} = .017$$

$$\cos(\text{digital}, \text{information}) =$$

$$\frac{5 * 5 + 1683 * 3982 + 1670 * 3325}{\sqrt{5^2 + 1683^2 + 1670^2} \sqrt{5^2 + 3982^2 + 3325^2}} = .996$$

Raw Frequency

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

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

Need a form of weighting!

Weighting for Raw Frequencies

- tf-idf: Term Frequency - Inverse Document Frequency

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

- Downweighting words like “the” or “if”

- PMI: Pointwise Mutual Information

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

- Consider if words like "good" appear more often with "great" than we would expect by chance
 - More formally, the log of the ratio between the joint probability and the expected probability if the events were independent.

PMI: Example

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

	bark	meow	pet
dog	8	0	2
cat	0	6	4

Term Frequency (tf) in TF-IDF

We could imagine using the raw count:

$$\text{tf}_{t,d} = \text{count}(t,d)$$

$\text{count}(t,d)$ = # occurrences of word t in document d

But instead of using raw count, we usually squash it a bit

- Idea: word appearing 100 times in a document doesn't make that work 100 times more likely to be relevant to the meaning of the document

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

Inverse Document Frequency

- Document frequency df_t is the number of documents t occurs in
 - Note: this is not collection frequency: total count across all documents)
 - Romeo* is very distinctive for one Shakespeare play
- Inverse Document Frequency: idf_t

$$idf_t = \log_{10} \left(\frac{N}{df_t} \right)$$

N = total # of documents in the collection

	Collection Frequency	Document Frequency
Romeo	113	1
action	113	31

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

TF-IDF

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

**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	0.246	0	0.454	0.520
good	0	0	0	0
fool	0.030	0.033	0.0012	0.0019
wit	0.085	0.081	0.048	0.054

Sparse vs. Dense Vectors

- Count co-occurrence vectors (even if weighted by tf-idf)
 - Long (length $|V|$ = 20,000 to 50,000)
 - High-dimensional & sparse (most elements are zero)
- Alternative: learn vectors which are
 - Short (Number of dimensions d ranging from 50-1000)
 - Dense (most elements are non-zero)

Why dense vectors?

- Short vectors may be easier to use as features in ML (fewer weights to tune)
- Dense vectors may generalize better than explicit counts
- Dense vectors may do better at capturing synonymy:
 - *car* and *automobile* are synonyms; but are distinct dimensions
 - a word with *car* as a neighbor and a word with *automobile* as a neighbor should be similar, but aren't
- **In practice, they work better!**

Common Methods for Getting Short Dense Vectors

- **“Neural Language Model”-inspired models**
 - **Word2vec** (skipgram, CBOW), GloVe
- **Singular Value Decomposition (SVD)**
 - A special case of this is called LSA – Latent Semantic Analysis
- **Alternative to these "static embeddings":**
 - Contextual Embeddings (ELMo, BERT)
 - Compute distinct embeddings for a word in its context
 - Separate embeddings for each token of a word

Word2Vec

- Popular embedding method
 - Very fast to train
 - Code available on the web
- Idea: predict rather than count
- Word2vec provides various options. We'll do:
skip-gram with negative sampling (SGNS)

Word2vec

- Instead of counting how often each word w occurs near "apricot"
 - Train a classifier on a binary prediction task:
 - Is w likely to show up near "apricot"?
- We don't actually care about this task
 - But we'll take the learned classifier weights as the word embeddings
- Big idea: **self-supervision**:
 - A word c that occurs near apricot in the corpus acts as the gold "correct answer" for supervised learning
 - No need for human labels
 - Bengio et al. (2003); Collobert et al. (2011)

Approach: Predict if Candidate Word c is a "Neighbor"

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

Skim-Gram Training Data & Classifier

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

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

c1 c2 [target] c3 c4

Goal: train a classifier that is given a candidate (word, context) pair

(apricot, jam)

(apricot, aardvark)

...

And assigns each pair a probability:

$$P(+ | w, c)$$

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

Similarity is computed from dot product

- Remember: two vectors are similar if they have a high dot product
 - Cosine is just a normalized dot product
- So:
 - $\text{Similarity}(w,c) \propto w \cdot c$
- We'll need to normalize to get a probability
 - (cosine isn't a probability either)

Turning dot products into probabilities

- $\text{Sim}(w, c) \approx w \cdot c$
- To turn this into a probability
- We'll use the sigmoid from logistic regression:

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

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

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:

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

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

Skip-gram Classifier: Summary

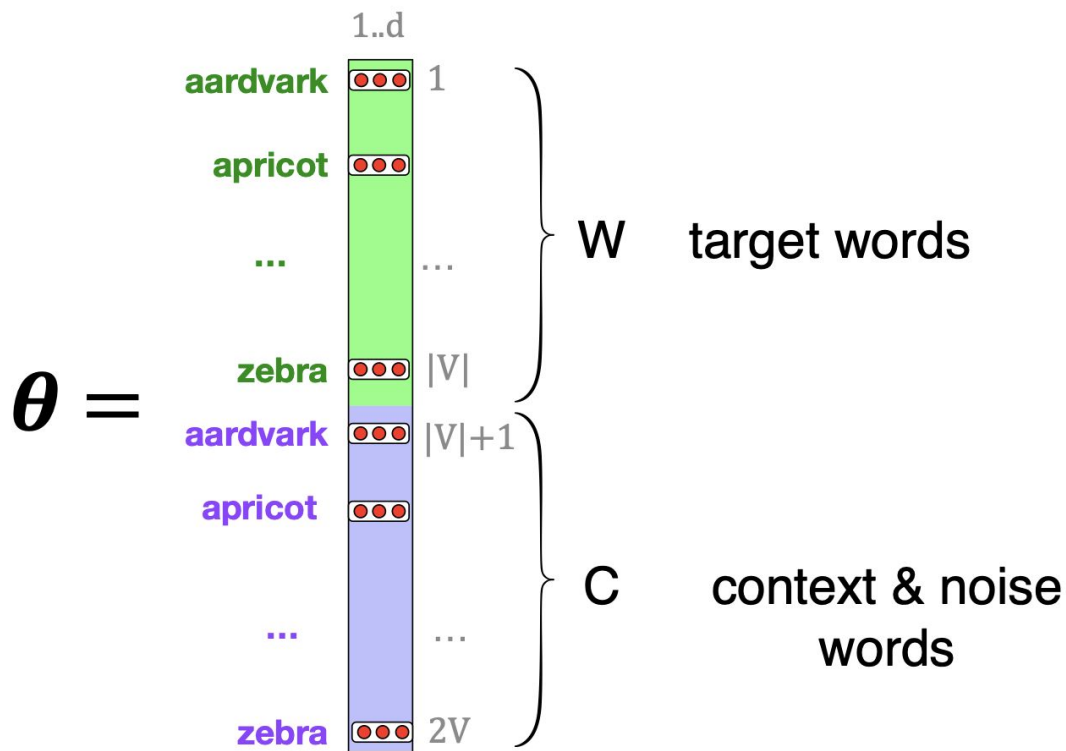
A probabilistic classifier, given

- a test target word w
- its context window of L words $c_{1:L}$

Estimates probability that w occurs in this window based on similarity of w (embeddings) to $c_{1:L}$ (embeddings).

To compute this, we just need embeddings for all the words.

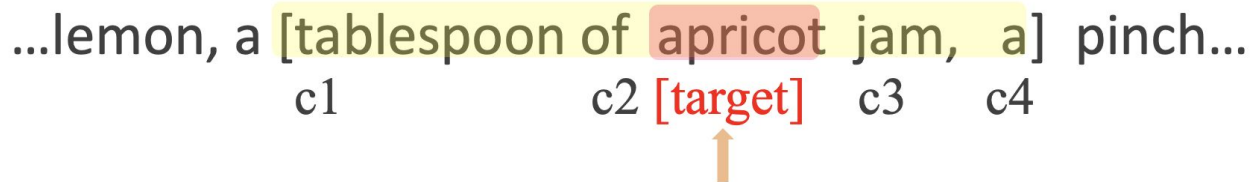
These embeddings we'll need: a set for w , a set for c



Learning the Embeddings: Skim-Gram Training Data

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

c1 c2 [target] c3 c4



positive examples +

t

c

apricot tablespoon

apricot of

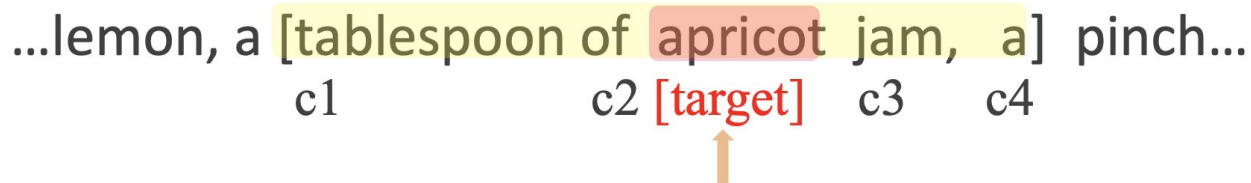
apricot jam

apricot a

Learning the Embeddings: Skim-Gram Training Data

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

c1 c2 [target] c3 c4



positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

For each positive example we'll grab k negative examples, sampling by frequency.

Learning the Embeddings: Skim-Gram Training Data

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

c1 c2 [target] c3 c4

↑

positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

t	c	t	c
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

Word2vec: how to learn vectors

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

The goal of learning is to adjust those word vectors such that we:

- **Maximize** the similarity of the **target word, context word** pairs (w, c_{pos}) drawn from the positive data
- **Minimize** the similarity of the (w, c_{neg}) pairs drawn from the negative data.

Loss function for one w with $c_{pos}, c_{neg1} \dots c_{negk}$

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

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

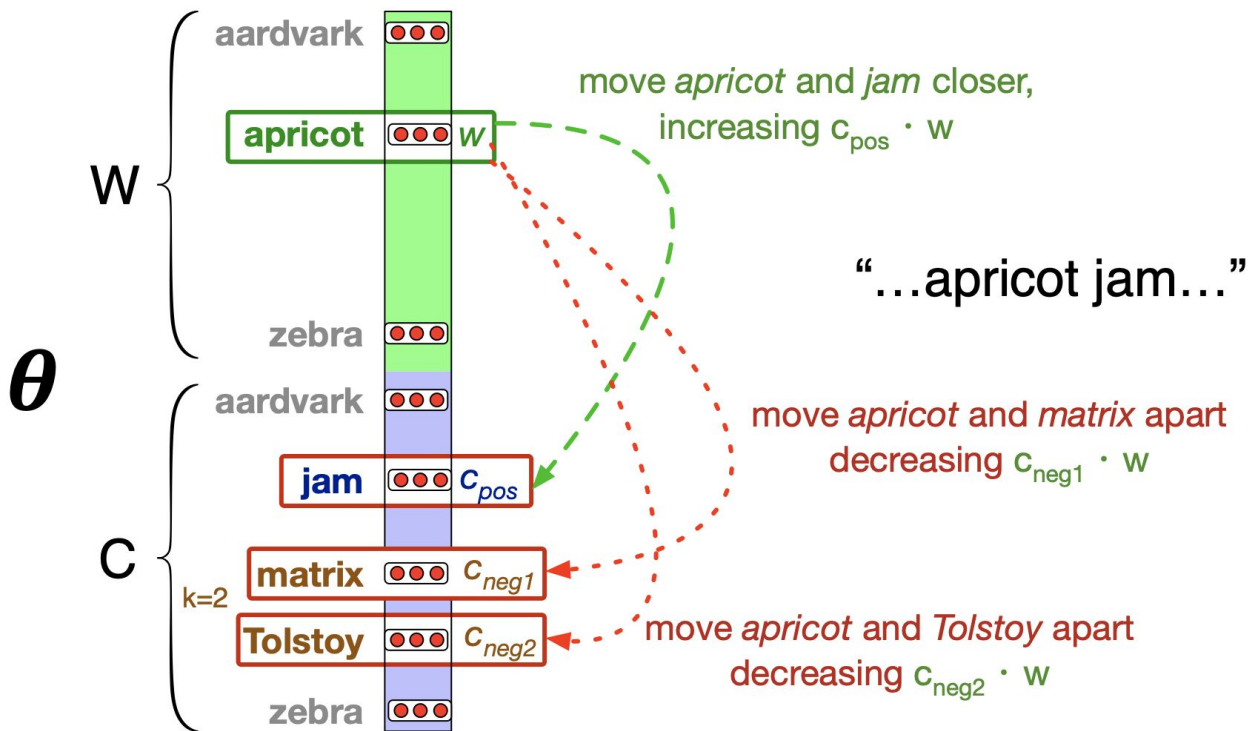
Learning the classifier: Intuition of One Step of Gradient Descent

How to learn?

- Stochastic gradient descent!

We'll adjust the word weights to

- make the positive pairs more likely
- and the negative pairs less likely,
- over the entire training set.



Reminder: Gradient Descent

- At each step:
 - Direction: We move in the reverse direction from the gradient of the loss function
 - Magnitude: we move the value of this gradient $d/dw L(f(x;w),y)$ weighted by a learning rate η
 - Higher learning rate means move w faster

$$w^{t+1} = w^t - \eta \frac{d}{dw} L(f(x;w), y)$$

The Derivatives of the Loss Function

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

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

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

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

Update Equation in SGD

Start with randomly initialized C and W matrices, then incrementally do updates

$$c_{pos}^{t+1} = c_{pos}^t - \eta [\sigma(c_{pos}^t \cdot w^t) - 1] w^t$$

$$c_{neg}^{t+1} = c_{neg}^t - \eta [\sigma(c_{neg}^t \cdot w^t)] w^t$$

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

Update Equation in SGD

Start with randomly initialized C and W matrices, then incrementally do updates

$$c_{pos}^{t+1} = c_{pos}^t - \eta [\sigma(c_{pos}^t \cdot w^t) - 1] w^t$$

$$c_{neg}^{t+1} = c_{neg}^t - \eta [\sigma(c_{neg}^t \cdot w^t)] w^t$$

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

Two Sets of Embeddings:

- SGNS learns two sets of embeddings:
 - Target embeddings matrix W
 - Context embedding matrix C
- It's common to just add them together, representing word i as the vector $w_i + c_i$

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

Start with V random d -dimensional vectors as initial embeddings

Train a classifier based on embedding similarity

- Take a corpus and take pairs of words that co-occur as positive examples
- Take pairs of words that don't co-occur as negative examples
- Train the classifier to distinguish these by slowly adjusting all the embeddings to improve the classifier performance
- Throw away the classifier code and keep the embeddings.

Properties of Embeddings

The kinds of neighbors depend on window size

Small windows ($C = \pm 2$) : nearest words are syntactically similar words in same taxonomy

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

Large windows ($C = \pm 5$) : nearest words are related words in same semantic field

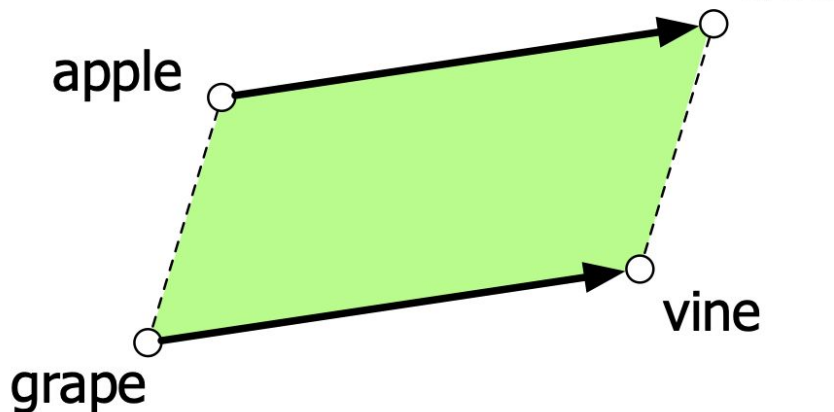
- *Hogwarts* nearest neighbors are Harry Potter world:
- *Dumbledore, half-blood, Malfoy*

Analogical Relations

The classic parallelogram model of analogical reasoning (Rumelhart and Abrahamson 1973)

To solve: "apple is to tree as grape is to _____"

Add $\overrightarrow{\text{tree}} - \overrightarrow{\text{apple}}$ to $\overrightarrow{\text{grape}}$ to get **vine**



Analogical relations via parallelogram

The parallelogram method can solve analogies with both sparse and dense embeddings (Turney and Littman 2005, Mikolov et al. 2013b)

$\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ is close to $\overrightarrow{\text{queen}}$
 $\overrightarrow{\text{Paris}} - \overrightarrow{\text{France}} + \overrightarrow{\text{Italy}}$ is close to $\overrightarrow{\text{Rome}}$

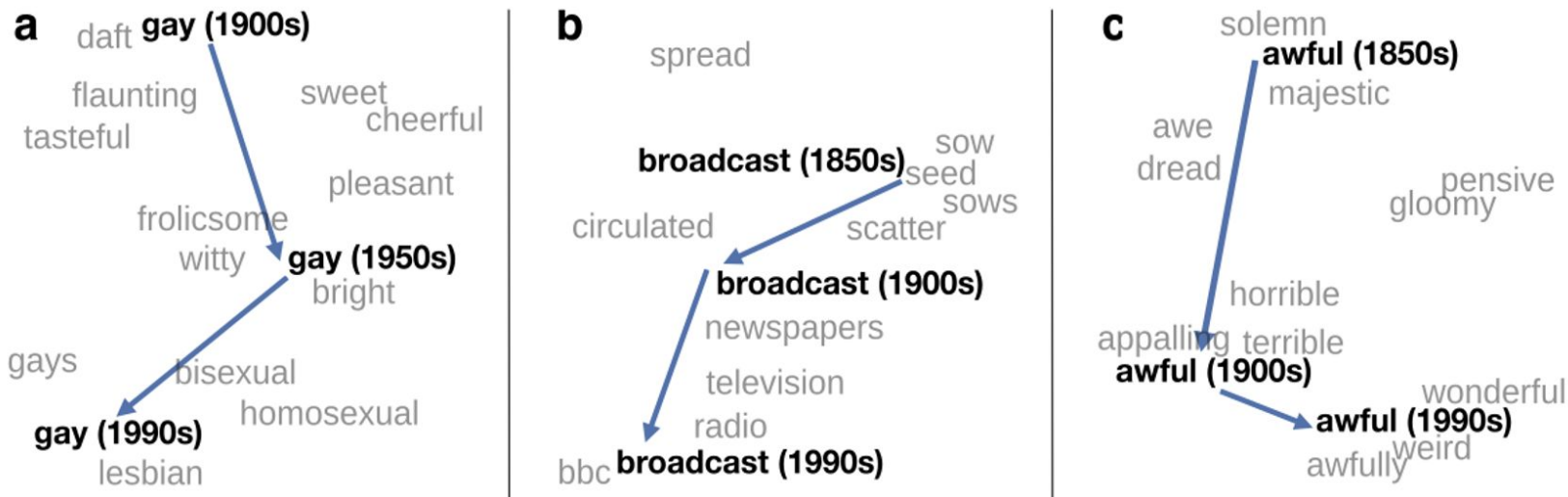
For a problem $a:a^*:b:b^*$, the parallelogram method is:

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

Embeddings as a Window into Historical Semantics

Train embeddings on different decades of historical text to see meanings shift
Visualizing semantic change over time.

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



TODOs:

- **Project Pitch Slide Due: 9/14 (Due 11:59PM)**
- **Homework 1 Released 5PM: Due 9/30**

Next Class: Project Pitches (I)

<https://www.cs.cmu.edu/~dst/WordEmbeddingDemo/>