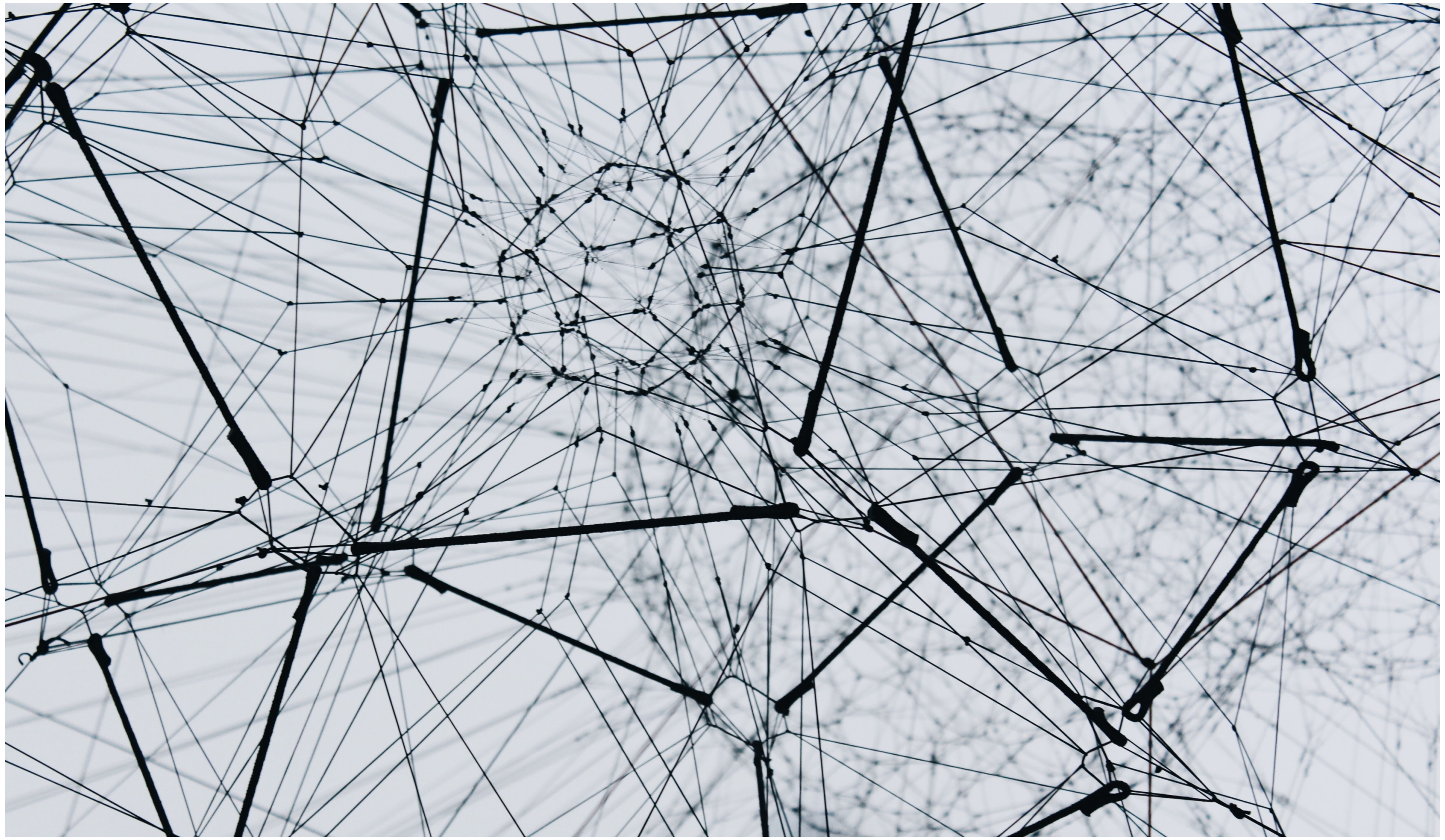


Word Embeddings

Natalie Parde, Ph.D.

Department of Computer Science

University of Illinois at Chicago

CS 421: Natural Language Processing

Fall 2019

Many slides adapted from Jurafsky and Martin (https://web.stanford.edu/~jurafsky/slp3/).

What are word embeddings?

- Vector representations of **word meaning**

pumpkin =

1	0	1	1	1	0	1	0	0	1

Distributional Hypothesis

- Words that occur in **similar contexts** tend to have **similar meanings**
  - It is time to harvest a **pumpkin**.
  - It is time to harvest a **squash**.
  - Add half a cup of canned **pumpkin** to the bread mix.
  - Add half a cup of canned **squash** to the bread mix.

Word embeddings make use of the distributional hypothesis to associate words with vectors.

Non-contextual Word Embeddings:

Every word has a single vector

- Word2Vec
- GloVe

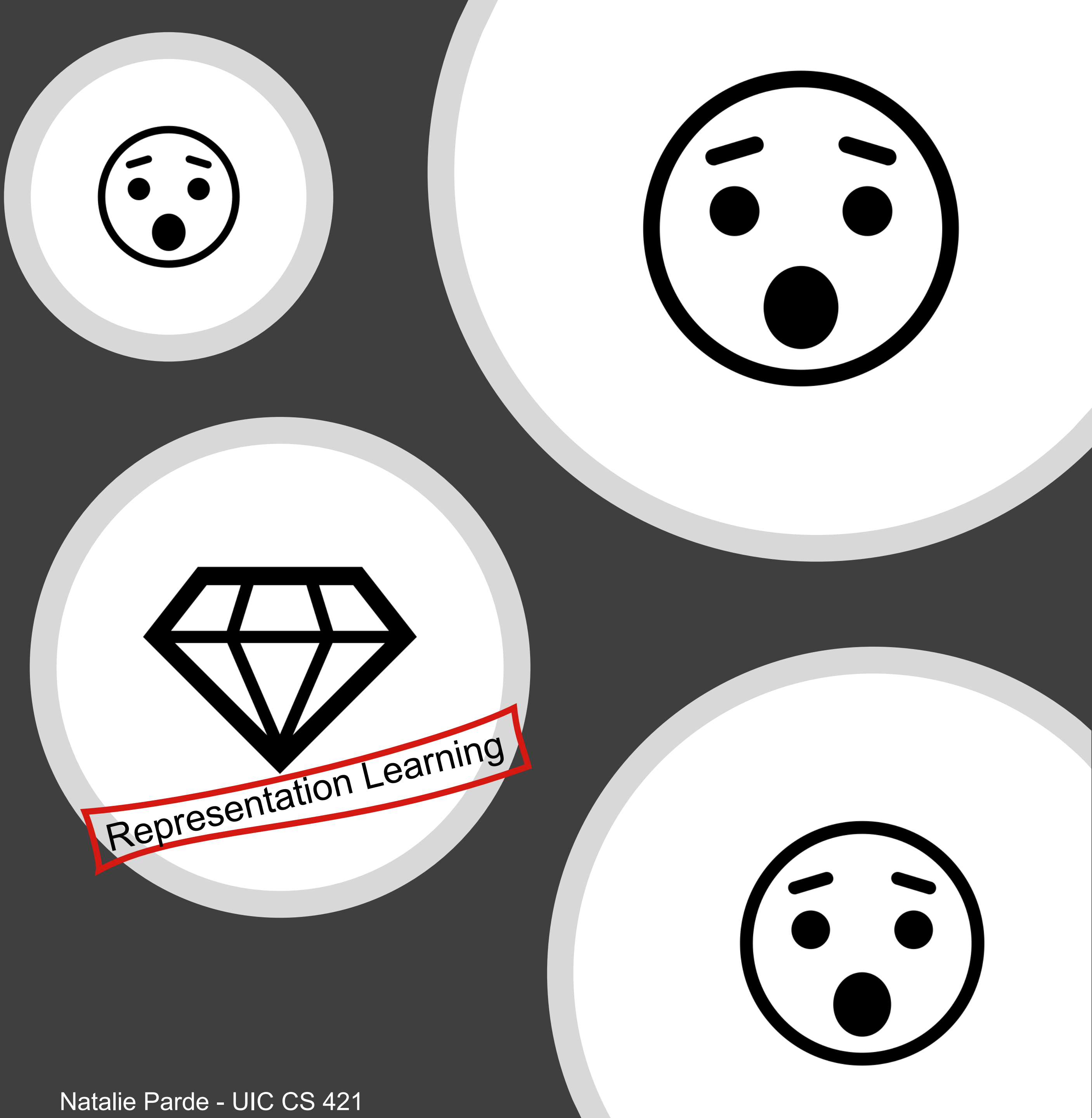
Contextual Word Embeddings:

A word can have multiple vectors

- ELMo
- BERT

Representation Learning

- Methods for learning word embeddings are examples of **representation learning**
    - **Representation Learning**: Automatically learning useful representations of input text
- Representation learning is **self-supervised**
- Recent trends in NLP have moved toward representation learning and away from creating representations by hand (i.e., **feature engineering**)



Lexical Semantics

- How should we represent the meaning of a word?
  - **N-gram models:** A word is a string of letters, or an index in a vocabulary list
  - **Logical representation:** A word defines its own meaning (“dog” = DOG)
- Problem?
  - These methods don’t allow us to infer anything deeper about a word
    - Similar meanings
    - Antonyms
    - Positive/negative connotations
    - Related contexts

Some Useful Properties of Meaning

- Lemmas and senses
- Synonymy
- Word similarity
- Word relatedness
- Frames and roles
- Connotation

Lemmas and Senses

- **Lemma:** The base form of a word
  - Pumpkins → pumpkin
  - Mice → mouse
- **Word Sense:** Different aspects of meaning for a word
  - Mouse (1): A small rodent
  - Mouse (2): A device to control a computer cursor
- Words with the same lemma should (hopefully!) reside near one another in vector space
- Different senses of words should be represented as different vectors in **contextual word representations**, but not in **non-contextual word representations**

Synonymy

- How do word senses relate to one another?
  - When a word sense for one word is (nearly) identical to the word sense for another word, the two words are **synonyms**
- **Synonymy:** Two words are synonymous if they are substitutable for one another in any sentence without changing the situations in which the sentence would be true
- Synonymy = **propositional meaning**

Take a seat on my brand new leather **couch**!

Take a seat on my brand new leather **sofa**!

Word Similarity

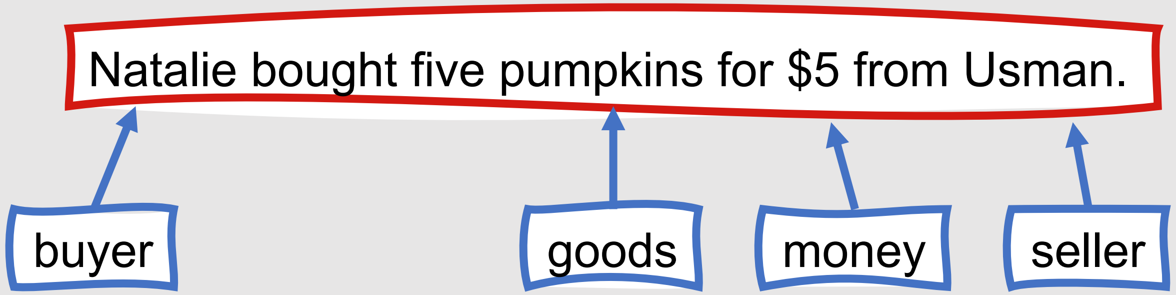
- Words don’t often have that many synonyms, but they do have a lot of **similar** words
  - Pumpkin ≈ squash
- Similarity deals with **relations between words**, not word senses
  - Could word Y be commonly used in the same context as word X?
    - Half a cup of squash 🙂
    - Harvest a squash 🙂
    - Squash spice latte 🤨
- Estimates of word similarity can help with tasks like:
  - Question answering
  - Paraphrasing
  - Summarizing

Word Relatedness

- Words can also be **related**, but not similar, to one another
- **Word Relatedness:** An association between words based on their shared participation in an event or **semantic field**
  - **Semantic Field:** A set of words covering a particular semantic domain
    - Restaurant: {waiter, menu, plate, food, …, chef}
- Related words can often be determined using **topic modeling** approaches

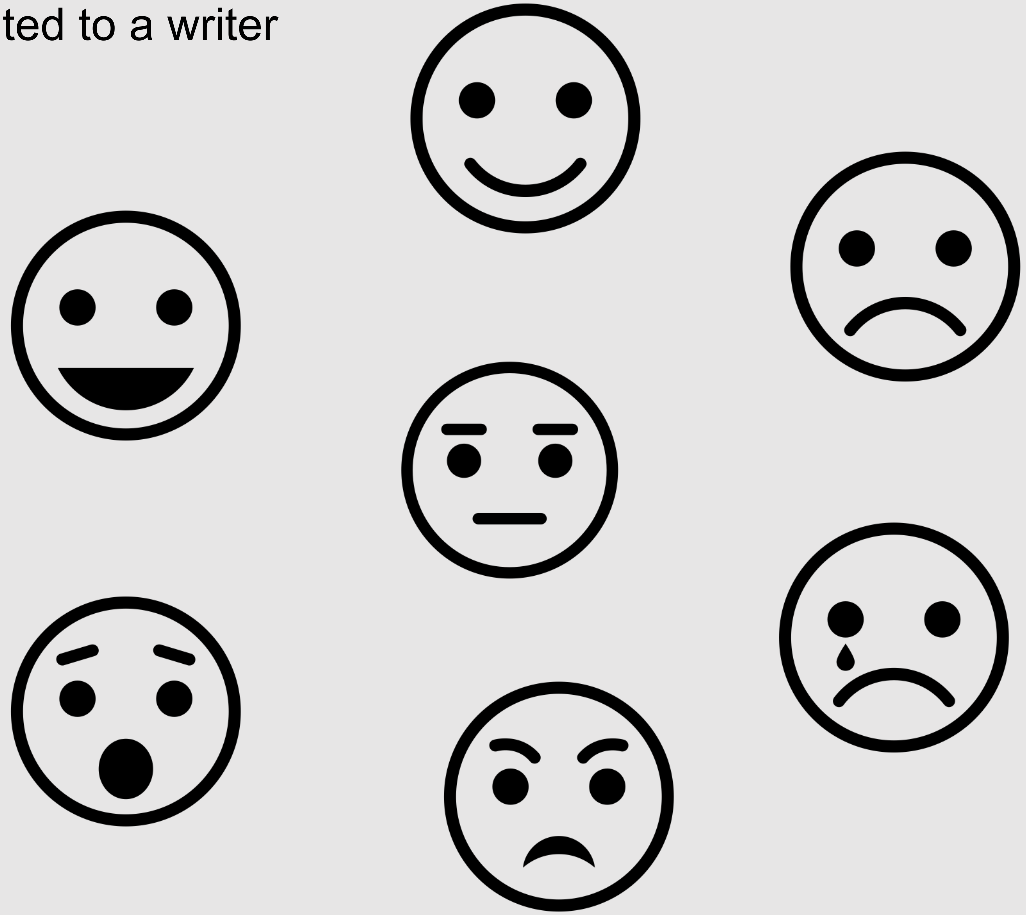
Semantic Frames and Roles

- **Semantic Frame:** A set of words that denote perspectives or participants in a particular type of event
  - Commercial Transaction = {buyer, seller, goods, money}
  - Closely related to semantic fields
- **Semantic Role:** A participant's underlying role with respect to the main verb in the sentence


Natalie bought five pumpkins for $5 from Usman.

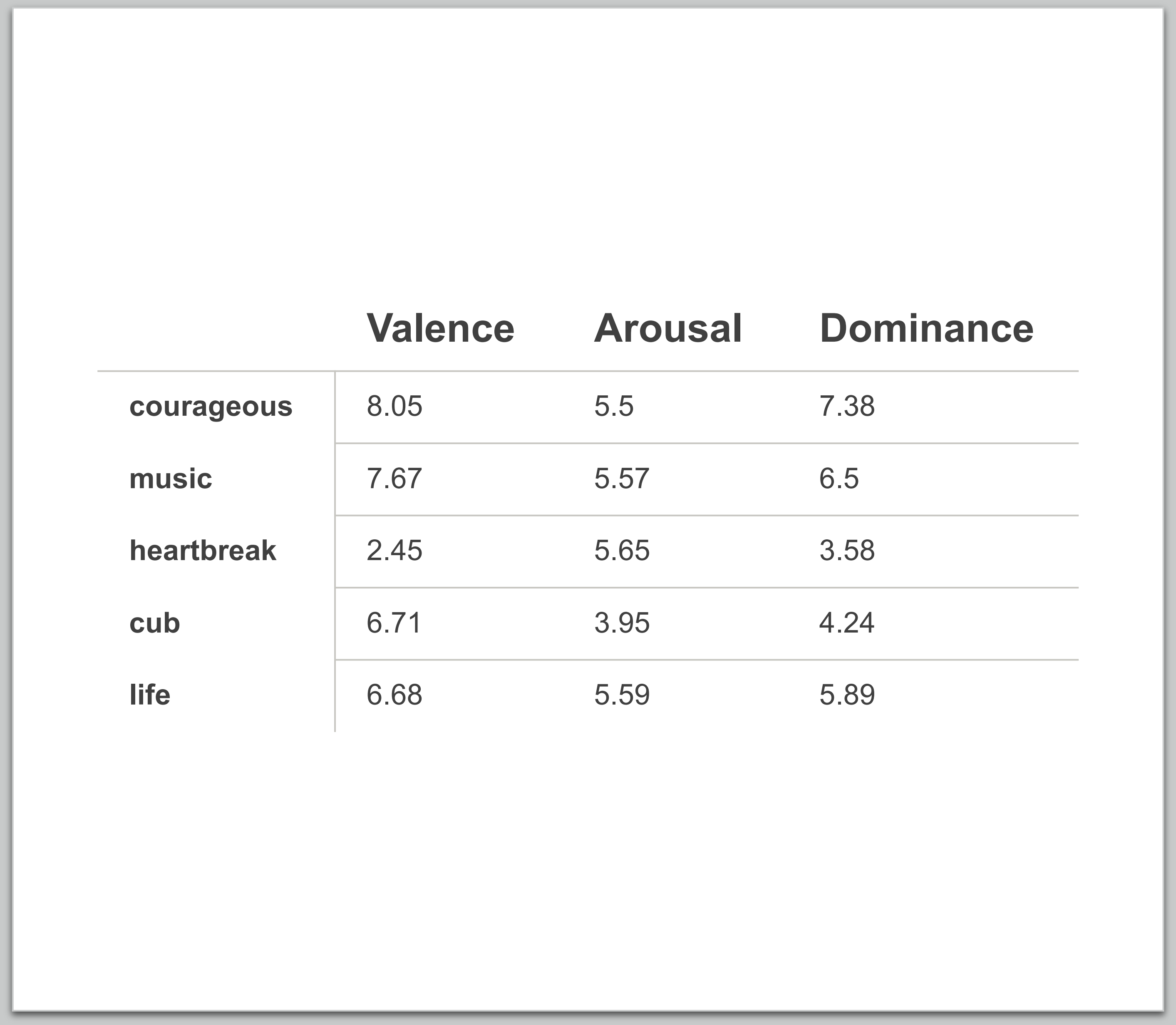
Connotation

- Also referred to as **affective meaning**
- **Connotation:** The aspects of a word's meaning that are related to a writer or reader's emotions, sentiment, opinions, or evaluations
- Generally three dimensions:
  - **Valence:** Positivity
    - High: Happy, satisfied
    - Low: Unhappy, annoyed
  - **Arousal:** Intensity of emotion
    - High: Excited, frenzied
    - Low: Relaxed, calm
  - **Dominance:** Degree of control
    - High: Important, controlling
    - Low: Awed, influenced



Connotation (Continued)

- Following this line of thought, each word can be represented by three numbers, corresponding to its value on each of the three affective dimensions
- When Osgood et al. (1957) did this, they ended up creating the first **word vectors**



	Valence	Arousal	Dominance
courageous	8.05	5.5	7.38
music	7.67	5.57	6.5
heartbreak	2.45	5.65	3.58
cub	6.71	3.95	4.24
life	6.68	5.59	5.89

Vector Semantics

- **Vector semantics** is a computational model that encodes different aspects of word meaning (e.g., word senses, word similarity and relatedness, lexical fields and frames, connotation, etc.) in vector form based on the word's usage in language
  - **A word is defined by its environment or distribution in language use**
- A word's distribution is the set of **contexts** in which it occurs
  - **Context:** Neighboring words or grammatical environments
- Two words that occur in very similar distributions are likely to have the same meaning

We do this all the time to infer meaning!

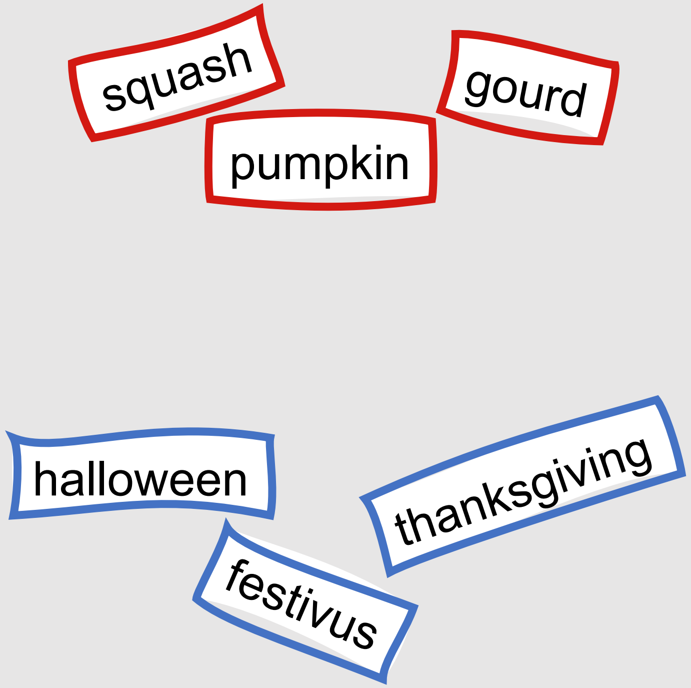
- Assume you don’t know what the Cantonese word *ongchoi* means
- However, you read the following sentences:
  - Ongchoi is delicious sautéed with garlic.
  - Ongchoi is superb over rice.
  - …ongchoi leaves with salty sauces…
- You’ve seen many of the other context words in these sentences previously:
  - …spinach sautéed with garlic over rice…
  - …chard stems and leaves are delicious…
  - …collard greens and other salty leafy greens…
- Your (correct!) conclusion?
  - Ongchoi is probably a leafy green similar to spinach, chard, or collard greens

We can do the same thing computationally.

- Count the words in the context of *ongchoi*
- See what other words occur in those same contexts

Vector semantics combines two intuitions.

- **Distributionalist**
  - Defining a word by counting what other words occur in its environment
- **Vector**
  - Defining a word as a vector point in an n-dimensional space
- Different versions of vector semantics define the numbers in the vector differently; however, they're always based in some way on counts of neighboring words



Advantages of Vector Semantics

- Easily compute word and phrase **similarity**
  - How similar are the vectors representing a set of words?
- Allows models learned for downstream tasks to **generalize** to new words
  - If a new word is close in vector space to one with a known sentiment value, we might be able to assume that the sentiment for the new word is similar as well
- Can be learned automatically from text **without labels or supervision**

How do we learn word embeddings?

Two commonly used models:

- TF*IDF
  - Long, sparse word vectors
- Word2Vec
  - Short, dense word vectors

Many, many other models!

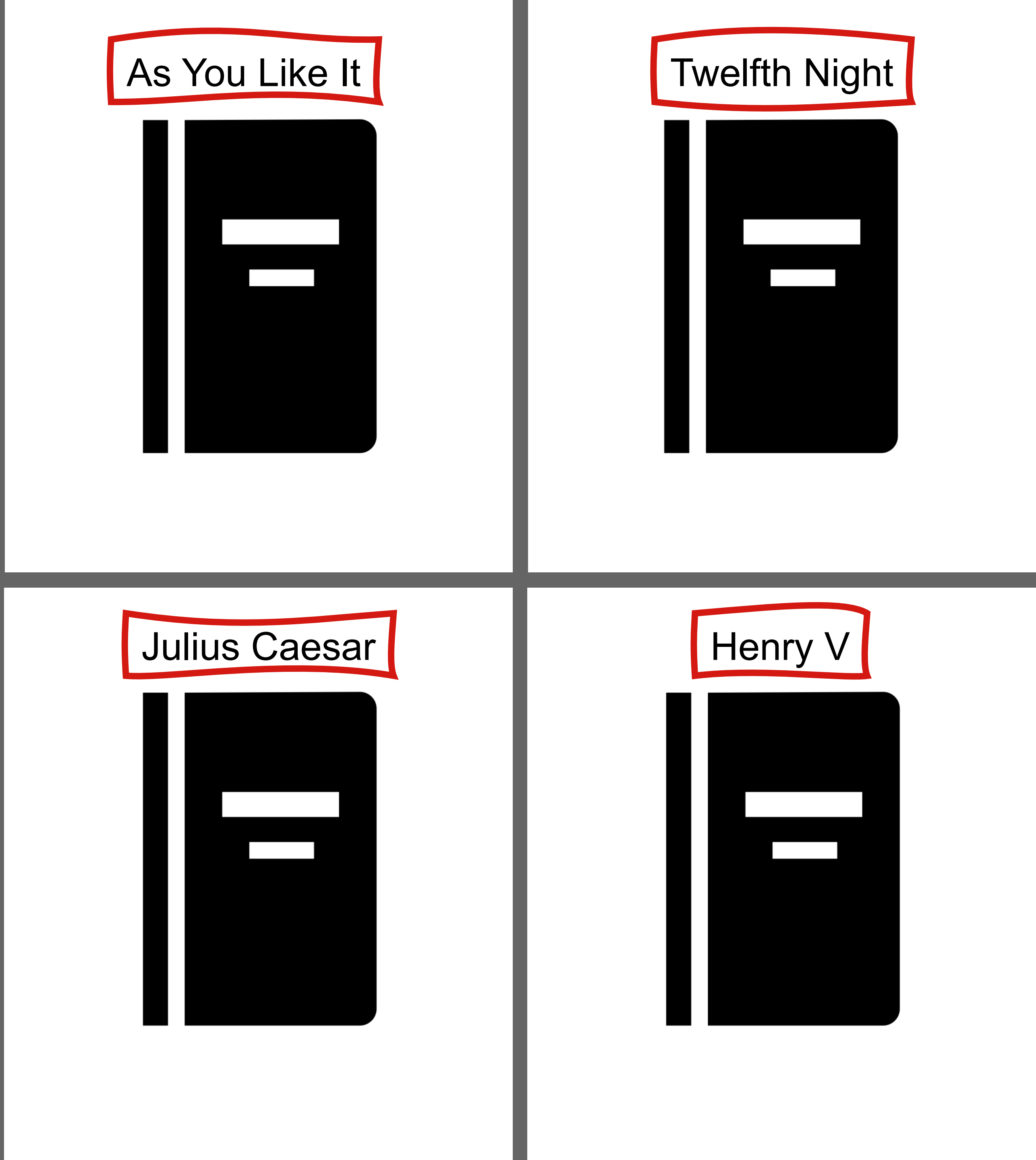
- GloVe
- ELMo
- BERT
- Etc.…

TF*IDF

- Term Frequency * Inverse Document Frequency
- Meaning of a word is defined by the counts of nearby words
- To do this, a specific type of **co-occurrence matrix** is needed
    - **Term-document matrix**

Term-Document Matrix

- Rows: Words in a vocabulary
- Columns: Documents in a selection

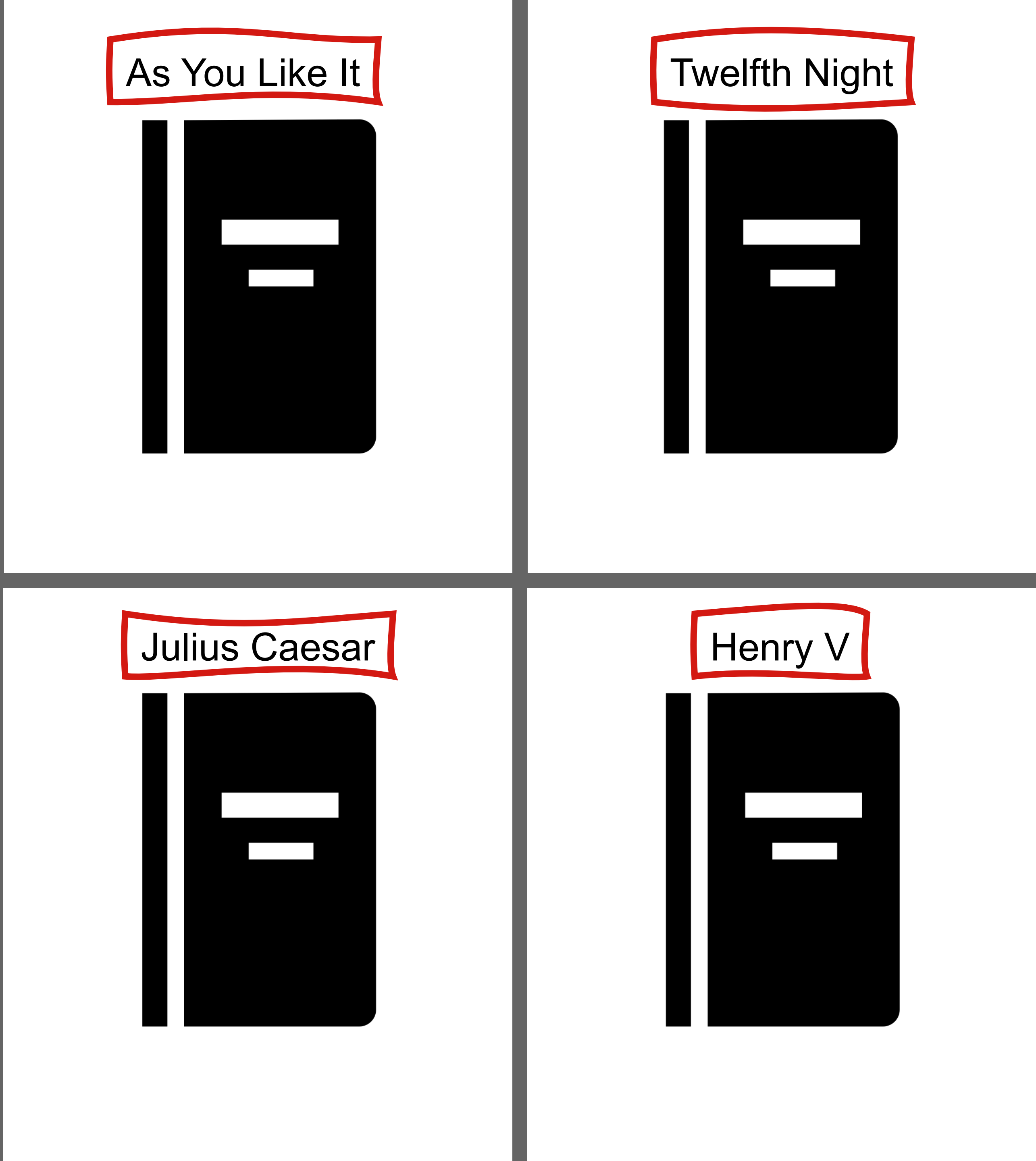


Term-Document Matrix

- Rows: Words in a vocabulary
- Columns: Documents in a selection

	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

“wit” appears 3 times in Henry V

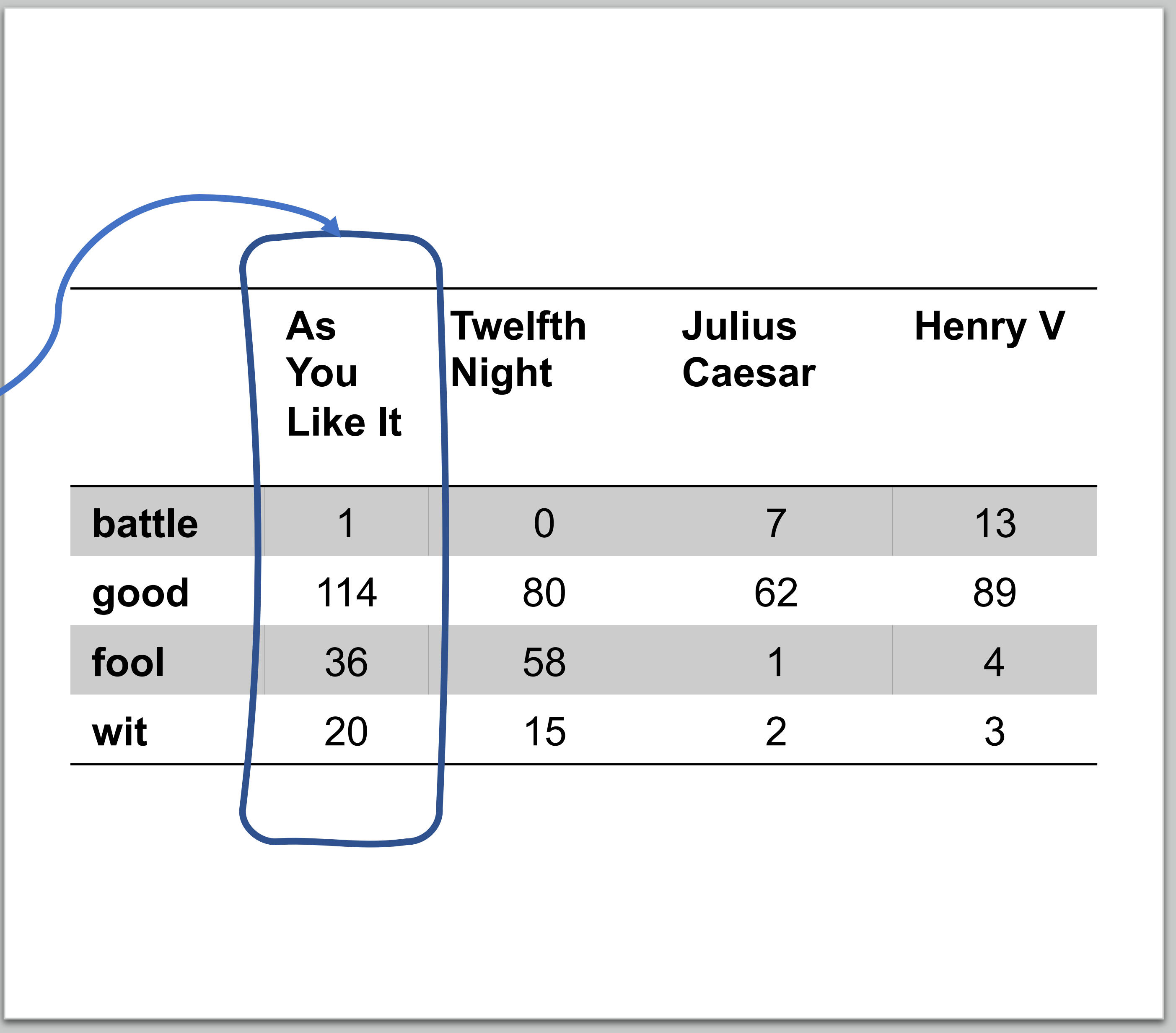


Ties to Information Retrieval

- Term-document matrices were first defined as part of the **vector space model** of **information retrieval**
  - Document = Count vector (column in the previous slide)
- Useful for searching through collections of documents (books, webpages, etc.) and returning those most relevant to a given search query

Linear Algebra Refresher

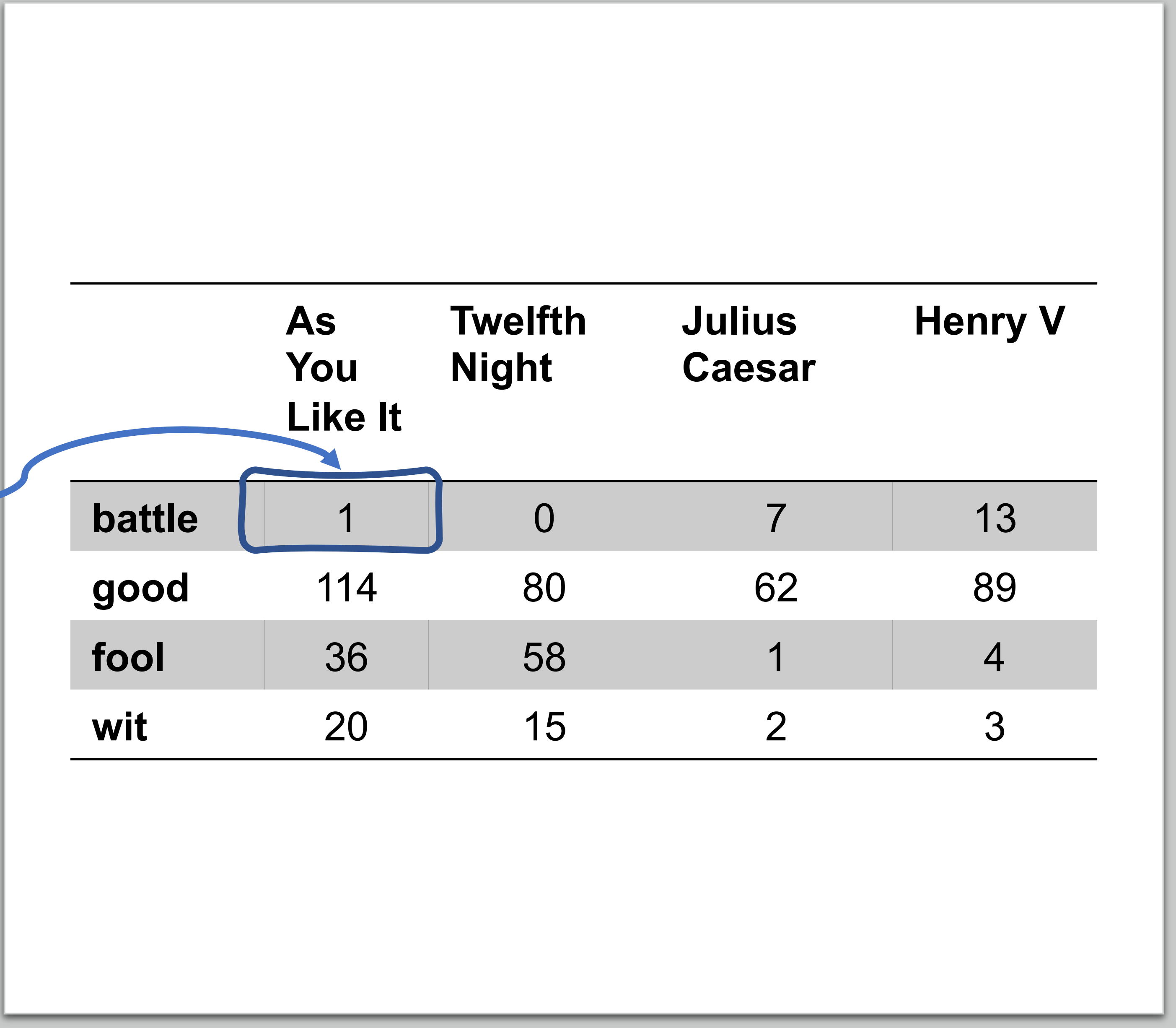
- **Vector:** A list or array of numbers
  - As You Like It = [1, 114, 36, 20]
  - Julius Caesar = [7, 62, 1, 2]
- **Vector Space:** Collection of vectors
- **Dimension:** Vector length
  - For TF*IDF, document vectors have a **dimensionality** of |V|, the vocabulary size



	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

Linear Algebra Refresher

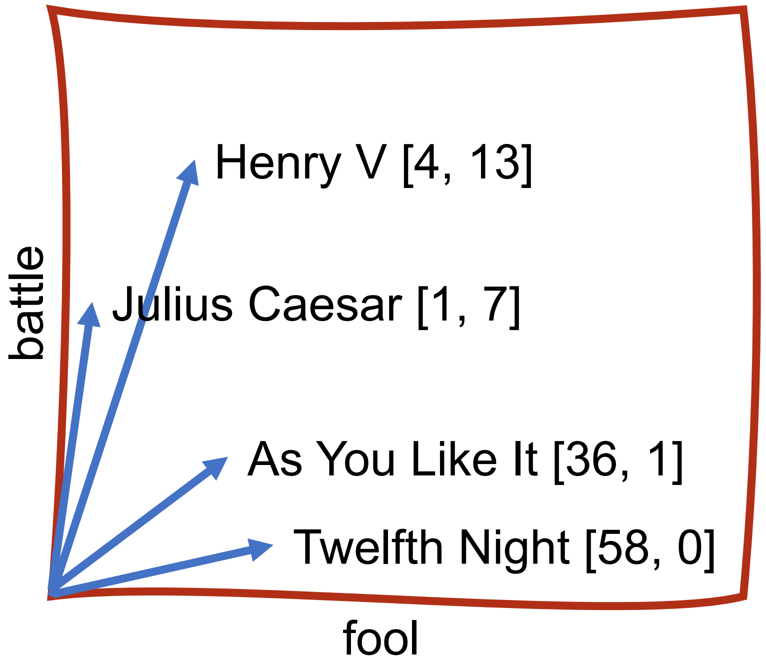
- Order is not arbitrary!
- Each position in a vector indicates a meaningful dimension in which a given unit (in this case, a document) can vary
  - AsYouLikeIt[0] = 1

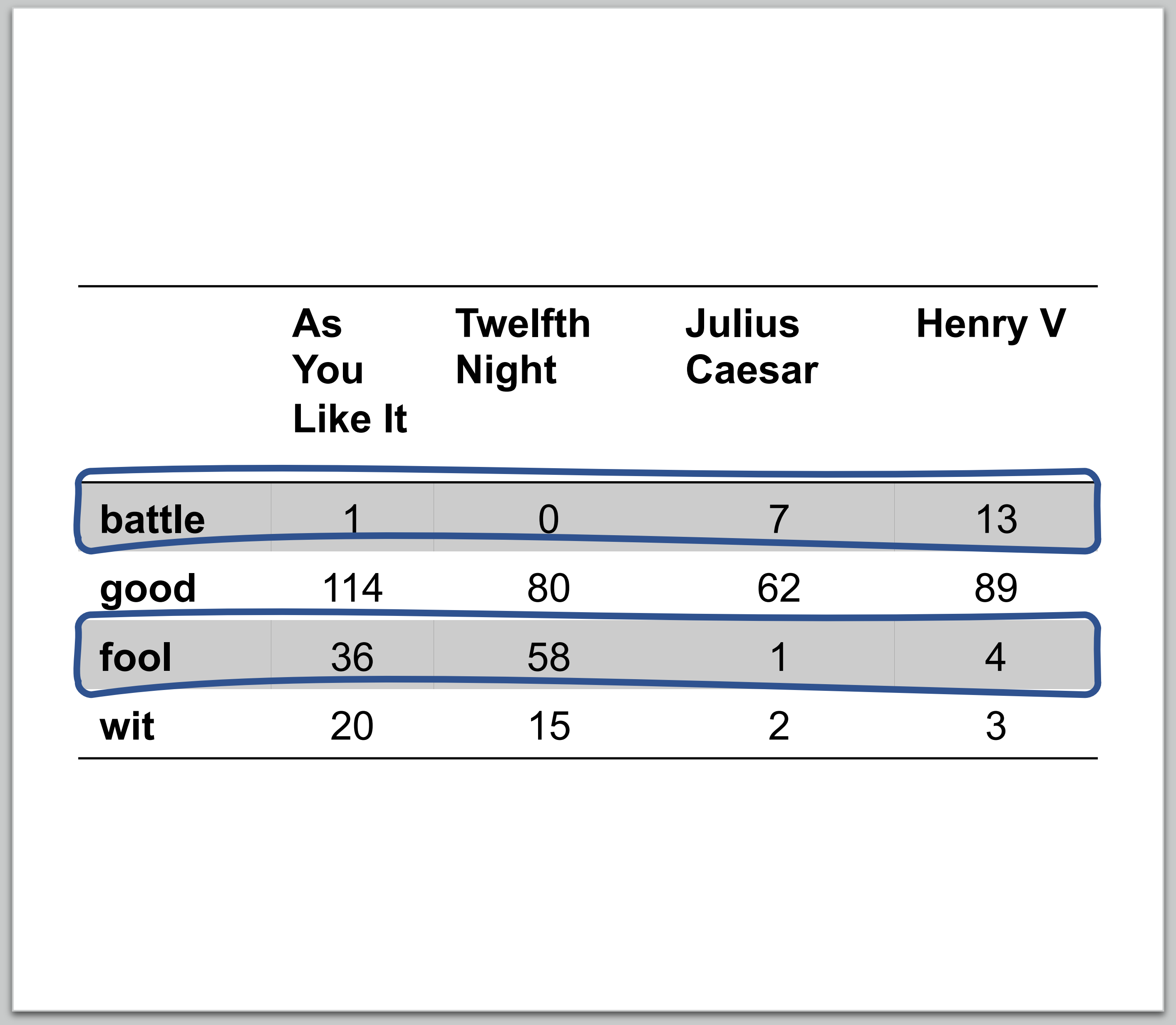


	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	$\boxed{1}$	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Linear Algebra Refresher

- Vectors indicate a unit's point in n-dimensional space

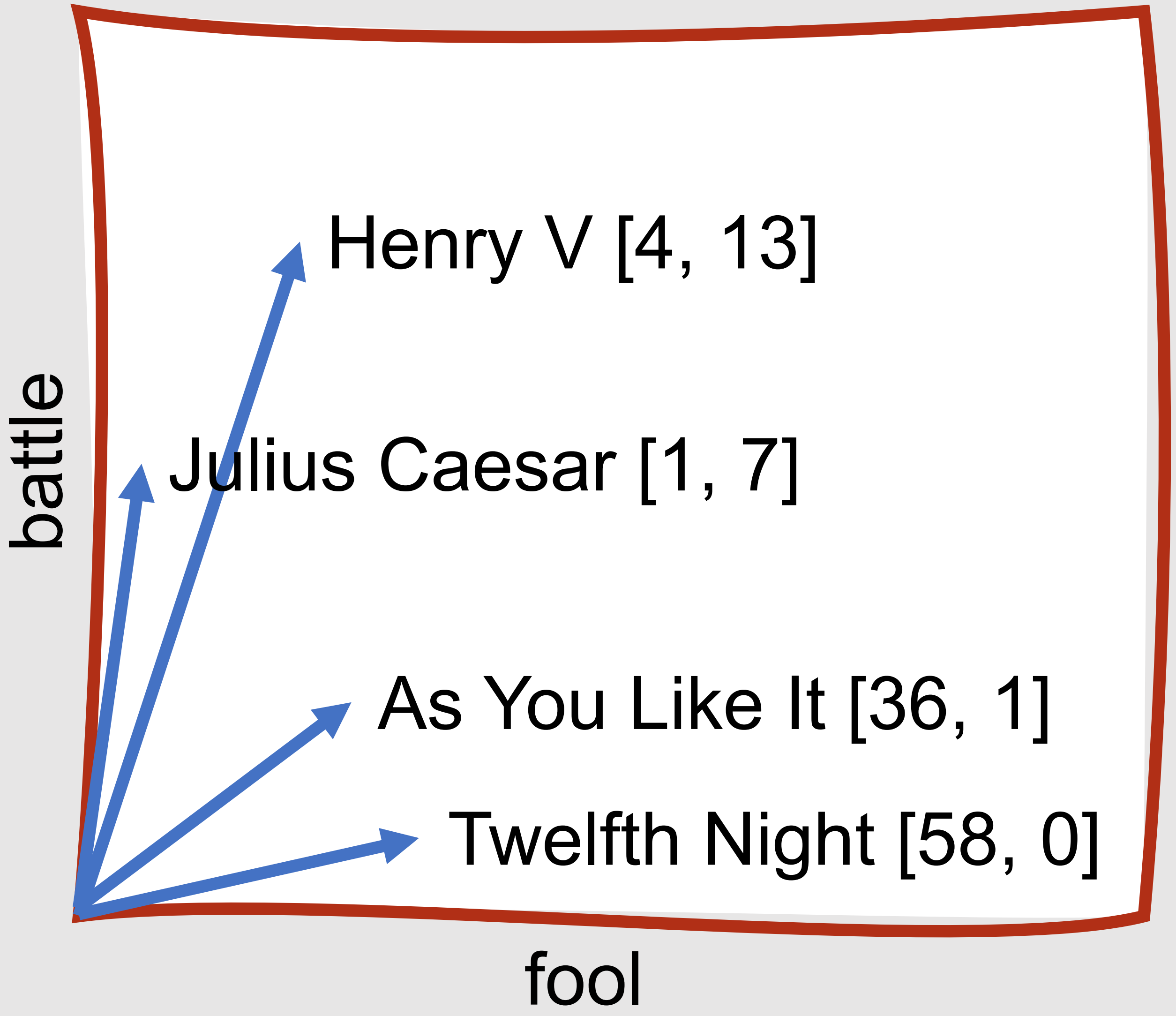




	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

What can we infer from this visualization?

- As You Like It and Twelfth Night are closer (and therefore more similar!) to each other than to Julius Caesar or Henry V

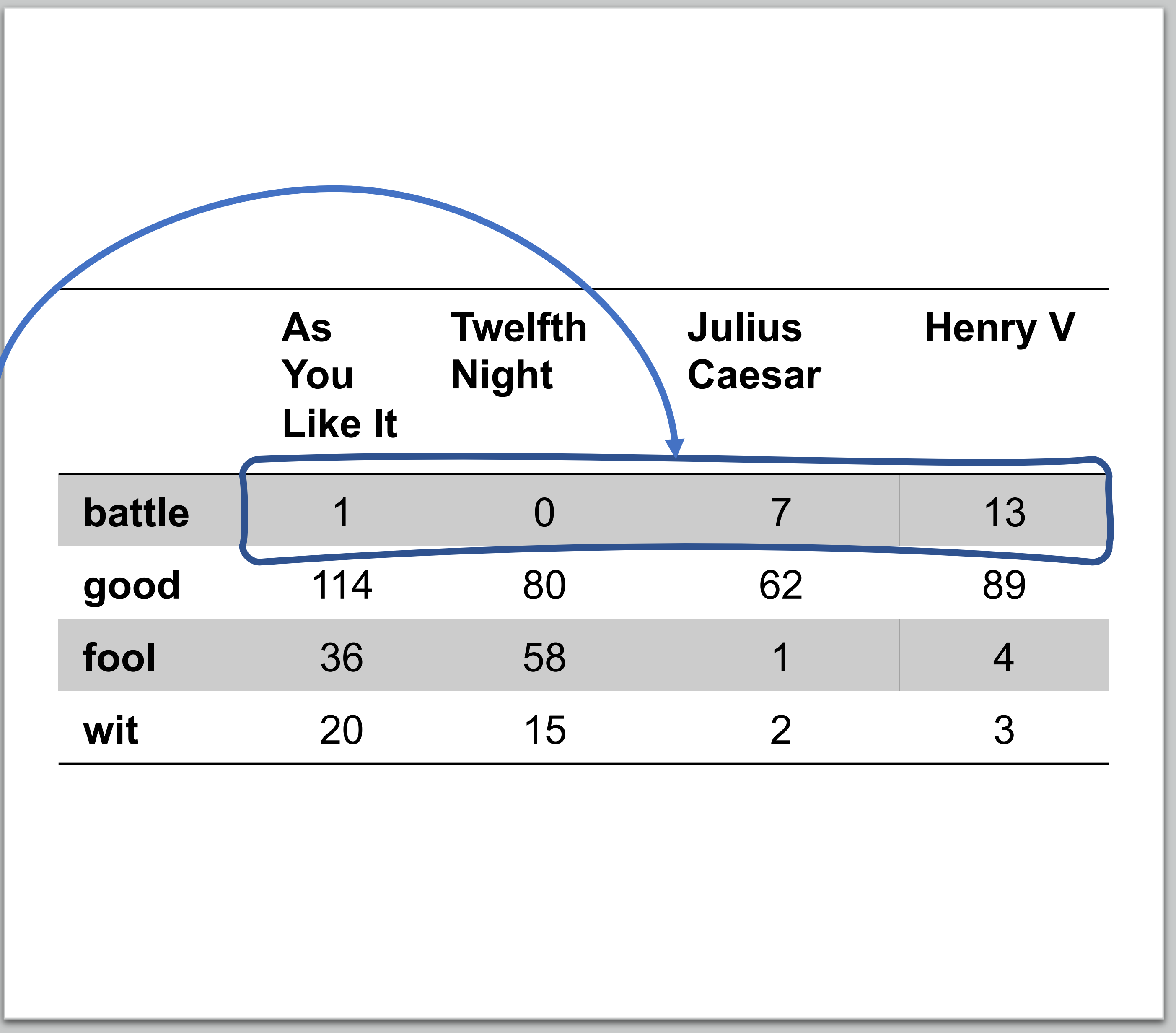


Real-world term-document matrices are much bigger!

- More generally, term-document matrices have |V| rows (one for each word type in the vocabulary) and D columns (one for each document in the collection)
- Vocabulary sizes are generally 10k+
- Number of documents can be enormous (e.g., all pages indexed by a web crawler)

Words as Vectors

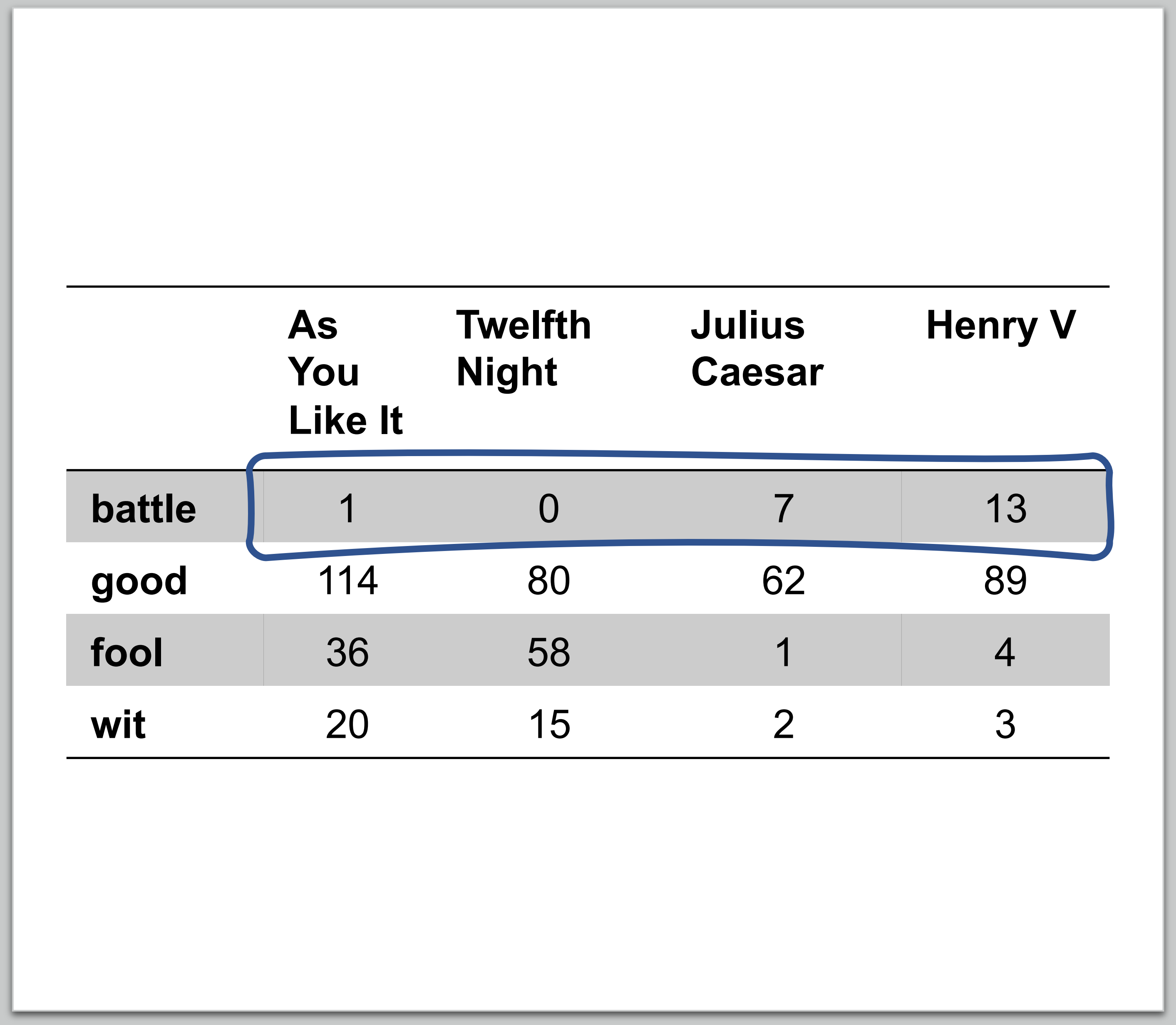
- In the previous example, we represented documents using vectors
- However, word embeddings represent **words** as vectors
  - battle = [1, 0, 7, 13]



	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

Words as Vectors

- In this scenario, similar words have similar vectors because they tend to occur in similar documents
- Thus, we're representing the meaning of a word by the documents in which it tends to occur



	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

Different Types of Context

- Documents aren’t the most common type of context used to represent meaning in word vectors
- More common: **word context**
  - Referred to as a term-term matrix, word-word matrix, or term-context matrix
- In a **word-word matrix**, the columns are also labeled by words
  - Thus, dimensionality is |V| x |V|
  - Each cell records the number of times the row (target) word and the column (context) word co-occur in some context in a training corpus

How can you decide if two words occur in the same context?

- Common **context windows**:
  - Entire document
    - Cell value = number of times the words co-occur in the same document
  - Predetermined number of words surrounding the target
    - Cell value = number of times the words co-occur in this set of words

Example Context Window (Size = 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

- We can take each occurrence of a word (e.g., strawberry) and count the context words around it to get a word-word co-occurrence matrix

Example Context Window (Size = 4)

- A simplified subset of a word-word co-occurrence matrix for the example words computed from Wikipedia could appear as shown

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	…

vector for “digital”

How can we measure the similarity between word vectors?

- **Cosine similarity**
  - Based on the **dot product** (also called **inner product**) from linear algebra
    - $\text{dot product}(\mathbf{v}, \mathbf{w}) = \mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \cdots + v_N w_N$
- Similar vectors (those with large values in the same dimensions) will have high values; dissimilar vectors (those with zeros in different dimensions) will have low values

Why don't we just use the dot product?

- More frequent words tend to co-occur with more words and have higher co-occurrence values with each of them
- Thus, the **raw dot product will be higher for frequent words**
- This is problematic!
  - We want our similarity metric to tell us how similar two words are regardless of frequency
- The simplest way to fix this problem is to **normalize for the vector length** (divide the dot product by the lengths of the two vectors)

Normalized Dot Product = Cosine of the angle between two vectors

- The cosine similarity metrics between two vectors v and w can thus be computed as:
  - $\text{cosine}(\mathbf{v}, \mathbf{w}) = \frac{\mathbf{v} \cdot \mathbf{w}}{|\mathbf{v}||\mathbf{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2} \sqrt{\sum_{i=1}^{N} w_i^2}}$
- This value ranges between 0 (dissimilar) and 1 (similar)

Example: Computing Cosine Similarity

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

cos(cherry, information) = ?

Example: Computing Cosine Similarity

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

$$\text{cos(cherry, information)} = \frac{[442{,}8{,}2]\cdot[5{,}3982{,}3325]}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}}$$

Example: Computing Cosine Similarity

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

$$\boxed{\text{cos(cherry, information)} = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}}}$$

Example: Computing Cosine Similarity

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

$$\boxed{\text{cos(cherry, information)} = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = 0.017}$$

Example: Computing Cosine Similarity

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

cos(cherry, information) = $\frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = 0.017$

cos(digital, information) = $\frac{5*5+1683*3982+1670*3325}{\sqrt{5^2+1683^2+1670^2}\sqrt{5^2+3982^2+3325^2}} = 0.996$

Example: Computing Cosine Similarity

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

$$\text{cos(cherry, information)} = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = 0.017$$

$$\text{cos(digital, information)} = \frac{5*5+1683*3982+1670*3325}{\sqrt{5^2+1683^2+1670^2}\sqrt{5^2+3982^2+3325^2}} = 0.996$$

Result: *information* is way closer to *digital* than it is to *cherry*!

In-Class Exercise

- Compute the cosine similarity to determine whether **pumpkin** or **squash** is closer to **halloween**.
- https://www.google.com/search?q=timer

	pie	jack-o'-lantern	zucchini
pumpkin	3	3	2
squash	2	1	3
halloween	2	3	1

In-Class Exercise

	pie	jack-o'-lantern	zucchini
pumpkin	3	3	2
squash	2	1	3
halloween	2	3	1

$$\text{cos(pumpkin, halloween)} = \frac{3*2+3*3+2*1}{\sqrt{3^2+3^2+2^2}\sqrt{2^2+3^2+1^2}} = \frac{17}{\sqrt{22}\sqrt{14}} = 0.969$$

$$\text{cos(squash, halloween)} = \frac{2*2+1*3+3*1}{\sqrt{2^2+1^2+3^2}\sqrt{2^2+3^2+1^2}} = \frac{10}{\sqrt{14}\sqrt{14}} = 0.714$$

So far, the co-occurrence matrices have contained raw frequency counts of word co-occurrences.

- However, this isn't the best measure of association between words
  - Some words co-occur frequently with many words, so won't be very informative
    - the, it, they
- We want to know about **words that co-occur frequently with one another, but less frequently across all texts**

How do we compute this?

- **TF*IDF**
  - Term Frequency * Inverse Document Frequency
- **Term Frequency:** The frequency of the word *t* in the document *d*
  - $tf_{t,d} = \text{count}(t, d)$
- **Document Frequency:** The number of documents in which the word *t* occurs
  - Different from collection frequency (the number of times the word occurs in the entire collection of documents)

Computing TF*IDF

- **Inverse Document Frequency:** The inverse of document frequency, where *N* is the total number of documents in the collection
  - $idf_t = \frac{N}{df_t}$
- IDF is higher when the term occurs in fewer documents
- What is a document?
  - Depends on your corpus!
    - Website
    - Book
    - Play
    - Article
    - Etc.
- If desired (e.g., if the document collection is large), we can squash both TF and IDF by using the $\log_{10}(tf_{t,d}+1)$ and $\log_{10} idf_t$

Computing TF*IDF

- TF*IDF is then simply the combination of TF and IDF
  - $tfidf_{t,d} = tf_{t,d} \times idf_t$

Example: Computing TF*IDF

- TF*IDF(battle, As You Like It) = ?

	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

Example: Computing TF*IDF

- TF*IDF(battle, As You Like It) = ?
- TF(battle, As You Like It) = 1

	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

Example: Computing TF*IDF

- TF*IDF(battle, As You Like It) = ?
- TF(battle, As You Like It) = 1
- IDF(battle) = N/DF(battle) = 4/3 = 1.33

	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

Example: Computing TF*IDF

- TF*IDF(battle, As You Like It) = ?
- TF(battle, As You Like It) = 1
- IDF(battle) = N/DF(battle) = 4/3 = 1.33
- TF*IDF(battle, As You Like It) = 1 * 1.33 = 1.33

	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

Example: Computing TF*IDF

- TF*IDF(battle, As You Like It) = ?
- TF(battle, As You Like It) = 1
- IDF(battle) = N/DF(battle) = 4/3 = 1.33
- TF*IDF(battle, As You Like It) = 1 * 1.33 = 1.33
- Alternately, TF*IDF(battle, As You Like It) = $\log_{10}(1+1) * \log_{10} 1.33 = 0.037$

	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

Extending TF*IDF Computation

	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

Word	df	idf
battle	21	$\log_{10}\frac{37}{21} = 0.246$
good	37	$\log_{10}\frac{37}{37} = 0.000$
fool	36	$\log_{10}\frac{37}{36} = 0.012$
wit	34	$\log_{10}\frac{37}{34} = 0.037$

- TF*IDF(battle, As You Like It) = ?
- TF(battle, As You Like It) = 1
- IDF(battle) = N/DF(battle) = 0.246
- TF*IDF(battle, As You Like It) = 1 * 0.246 = 0.246
- Alternately, TF*IDF(battle, As You Like It) = $\log_{10}(1+1) * 0.246 =$ 0.074

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074			
good				
fool				
wit				

- TF*IDF(battle, Twelfth Night) = ?

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074			
good				
fool				
wit				

- TF*IDF(battle, Twelfth Night) = ?
- TF(battle, Twelfth Night) = 0

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074			
good				
fool				
wit				

- TF*IDF(battle, Twelfth Night) = ?
- TF(battle, Twelfth Night) = 0
- IDF(battle) = N/DF(battle) = 0.246

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074			
good				
fool				
wit				

- TF*IDF(battle, Twelfth Night) = ?
- TF(battle, Twelfth Night) = 0
- IDF(battle) = N/DF(battle) = 0.246
- TF*IDF(battle, Twelfth Night) = $\log_{10}(0+1) * 0.246 = 0.000$

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0.000		
good				
fool				
wit				

- TF*IDF(battle, Twelfth Night) = ?
- TF(battle, Twelfth Night) = 0
- IDF(battle) = N/DF(battle) = 0.246
- TF*IDF(battle, Twelfth Night) = $\log_{10}(0+1) * 0.246 = 0.000$

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0.000		
good				
fool				
wit				

- TF*IDF(good, As You Like It) = ?

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0.000		
good				
fool				
wit				

- TF*IDF(good, As You Like It) = ?
- TF(good, As You Like It) = 114

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0.000		
good				
fool				
wit				

- TF*IDF(good, As You Like It) = ?
- TF(good, As You Like It) = 114
- IDF(good) = N/DF(good) = 0.000

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0.000		
good				
fool				
wit				

- TF*IDF(good, As You Like It) = ?
- TF(good, As You Like It) = 114
- IDF(good) = N/DF(good) = 0.000
- TF*IDF(good, As You Like It) = $\log_{10}(114 + 1) * 0.000 = 0.000$

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0.000		
good	0.000			
fool				
wit				

- TF*IDF(good, As You Like It) = ?
- TF(good, As You Like It) = 114
- IDF(good) = N/DF(good) = 0.000
- TF*IDF(good, As You Like It) = $\log_{10}(114 + 1) * 0.000 = 0.000$

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0.000	0.220	0.280
good	0.000	0.000	0.000	0.000
fool	0.019	0.021	0.004	0.008
wit	0.049	0.044	0.018	0.022

Example: Converting TF matrix to TF*IDF Matrix

	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

Word	idf
battle	$\log_{10}\frac{37}{21} = 0.246$
good	$\log_{10}\frac{37}{37} = 0.000$
fool	$\log_{10}\frac{37}{36} = 0.012$
wit	$\log_{10}\frac{37}{34} = 0.037$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0.000	0.220	0.280
good	0.000	0.000	0.000	0.000
fool	0.019	0.021	0.004	0.008
wit	0.049	0.044	0.018	0.022

- As shown, using TF*IDF eliminates the importance of the ubiquitous “good”
- It also reduces the impact of the almost-ubiquitous “fool”
- It increases the importance of the rarer word “battle”

Applications of TF*IDF Models

- The TF*IDF model produces a sparse (most cells have values of 0) vector with TF*IDF values in each cell
- Common uses of this model:
  - Computing word similarity
  - Computing document similarity
    - Find the vectors of all words in a document, compute the centroid of those vectors, and then compute the similarity between two document centroids

Summary: Word Embeddings (Part 1)

- **Word embeddings** are vector representations of meaning
- A vector for a word is computed based on the **contexts** in which the word occurs
  - Context = Documents or windows of words
- Word embeddings can be **sparse** or **dense**
  - **Sparse:** Bag of words, TF*IDF
  - **Dense:** Word2Vec, GloVe, ELMo, BERT
- **TF*IDF** vectors represent a word's meaning based on a combination of term frequency and inverse document frequency
- **Cosine similarity** can be used to determine the similarity between two word vectors

Sparse vs. Dense Word Embeddings

Sparse:

- Very high-dimensional vectors
- Lots of empty (zero-valued) cells

Dense:

- Lower-dimensional (50-1000 cells) vectors
- Most cells have non-zero values

Which embedding type is better for NLP tasks?

- **Dense vectors!**
- Why?
  - Easier to include as **features** in machine learning systems
    - Classifiers have to learn ~100 weights instead of ~50,000
  - Fewer **parameters** → lower chance of overfitting
    - May generalize better to new data
  - Better at capturing **synonymy**
    - Words are not distinct dimensions; instead, dimensions correspond to meaning components

What is the best way to generate dense word vectors?

- The answer changes quite frequently:
  - https://gluebenchmark.com/leaderboard/
  - https://rajpurkar.github.io/SQuAD-explorer/
- Current state-of-the-art models are **bidirectional** (trained to represent words using both their left and right context), **contextual** (produce different vectors for different word senses) models built using **transformers** (a type of neural network)

Classic Method

Skip-gram with **negative sampling**

Often referred to as **Word2Vec** (Mikolov et al., 2013)

Available online with code and pretrained embeddings: https://code.google.com/archive/p/word2vec/

Characteristics of Word2Vec

- Non-contextual
  - River **bank** and financial **bank** will both correspond to the same word vector
- Fast
- Efficient to train

Word2Vec Intuition

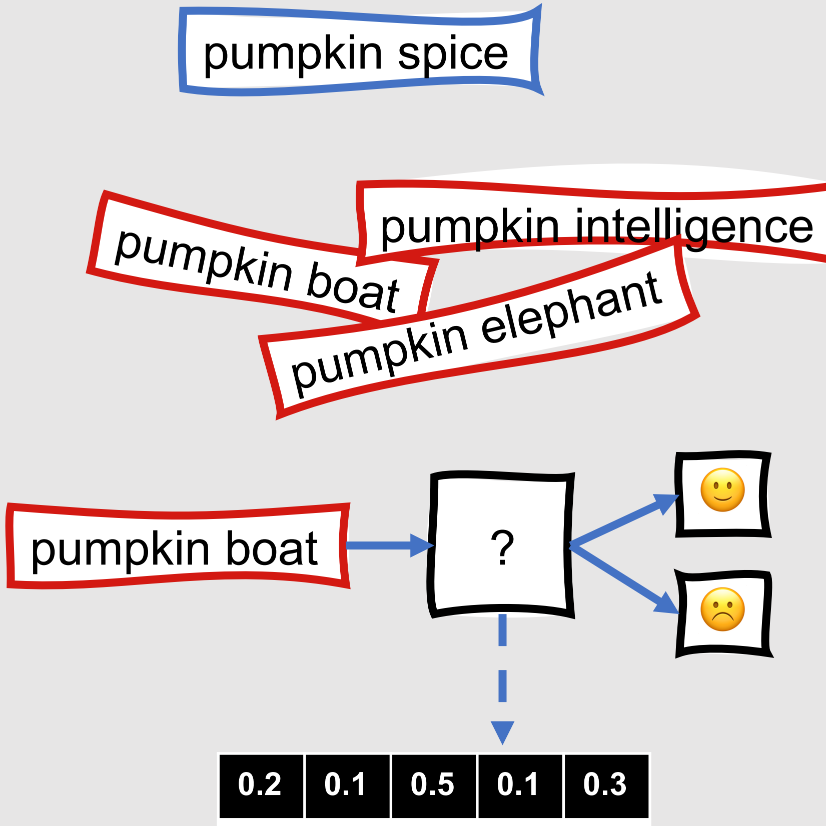
- Instead of counting how often each word occurs near each context word, train a classifier on a **binary prediction task**
  - Is word *w* likely to occur near context word *c*?
- The twist: **We don’t actually care about the classifier!**
- We use the **learned classifier weights** from this prediction task as our word embeddings

Word2Vec can carry out this entire process without manual supervision.

- Running text is **implicitly supervised** training data
  - Given the question: Is word w likely to occur near context word c?
    - If w occurs near c in the training corpus, the gold standard answer is “yes”
- This idea comes from **neural language modeling** (neural networks that predict the next word based on prior words)
- However, Word2Vec is simpler than a neural language model:
  - It makes **binary yes/no predictions** rather than predicting words
  - It trains a **logistic regression classifier** instead of a deep neural network

High-Level Overview: How Word2Vec Works

- Treat the target word *w* and a neighboring context word *c* as positive samples
- Randomly sample other words in the lexicon to get negative samples
- Use logistic regression to train a classifier to distinguish between those two cases
- Use the weights from that classifier as the embeddings for each word



pumpkin spice

pumpkin boat

pumpkin intelligence

pumpkin elephant

pumpkin boat

?

🙂

☹️

0.2	0.1	0.5	0.1	0.3

Classification Task

- Assume the following:
  - **Sentence fragment:** sugar, a tablespoon of pumpkin puree, a pinch
  - **Target word:** pumpkin
  - **Context window:** ± 2 words

sugar,	a	tablespoon	of	pumpkin	puree,	a	pinch
		c1	c2	t	c3	c4	

Classification Task

- Goal: Train a classifier that, given a tuple (*t*, *c*) of a target word *t* paired with a context word *c* (e.g., (pumpkin, puree) or (pumpkin, laminator)), will return the probability that *c* is a real context word
    - P(+|*t*,*c*)

sugar,	a	tablespoon	of	pumpkin	puree,	a	pinch
		c1	c2	t	c3	c4	

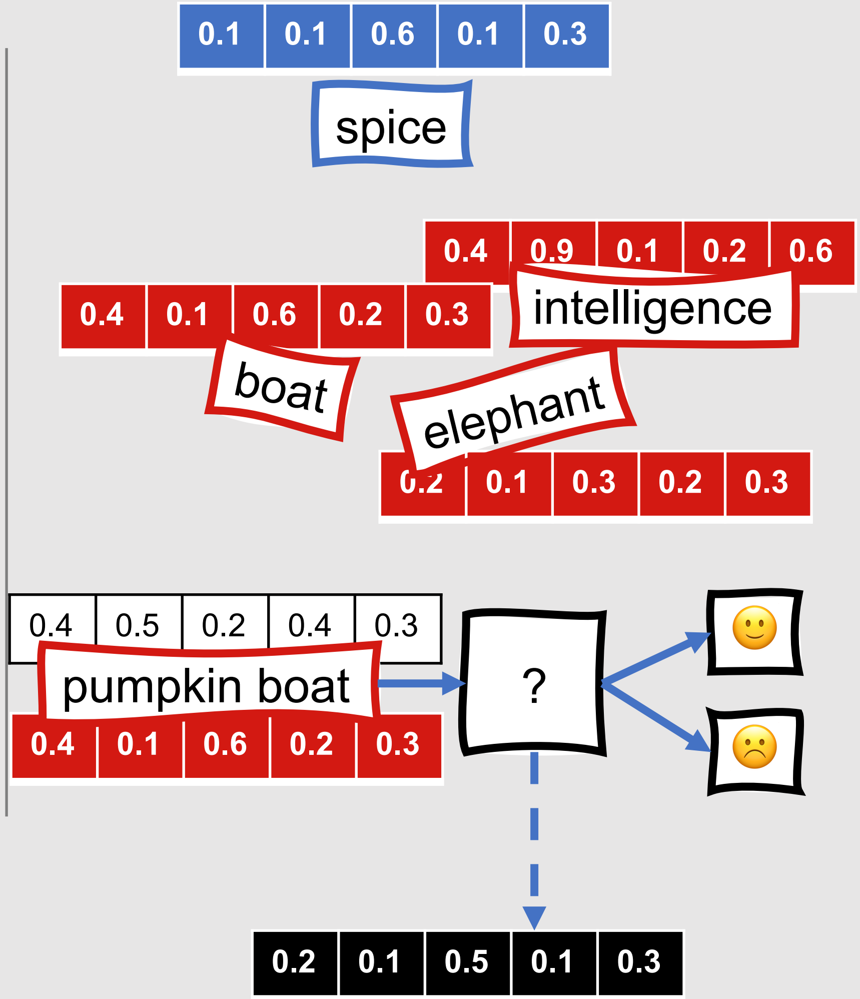
Classification Task

- How to compute P(+|*t*,*c*)?
  - Base it on similarity: A word is likely to occur near the target if its embedding is similar to the target embedding

sugar,	a	tablespoon	of	pumpkin	puree,	a	pinch
		c1	c2	t	c3	c4	

Updated High-Level Overview: How Word2Vec Works

- Represent all words in a vocabulary as a vector
- Treat the target word *w* and a neighboring context word *c* as positive samples
- Randomly sample other words in the lexicon to get negative samples
- Find the similarity for each (t,c) pair and use this to calculate P(c|t)
- Use logistic regression to train a classifier to maximize these probabilities to distinguish between positive and negative cases
- Use the weights from that classifier as the embeddings for each word



Classification Task

- How to compute P(+|*t*,*c*)?
  - Base it on similarity: A word is likely to occur near the target if its embedding is similar to the target embedding
  - As we already saw, vector similarity can be modeled using the dot product
    - Similarity(*t*,*c*) $\propto t \cdot c$
  - However, the dot product does not produce a probability (neither does the cosine similarity!)

sugar,	a	tablespoon	of	pumpkin	puree,	a	pinch
		c1	c2	t	c3	c4	

Classification Task

- How do we turn the dot product into a probability?
  - We can use the **logistic** or **sigmoid** function $\sigma(x)$
  - This function forms the core of logistic regression:
    - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - Since Similarity(*t*,*c*) $\propto t \cdot c$, we can set:
    - P(+|*t*,*c*) = $\frac{1}{1+e^{-t \cdot c}}$

sugar,	a	tablespoon	of	pumpkin	puree,	a	pinch
		c1	c2	t	c3	c4	

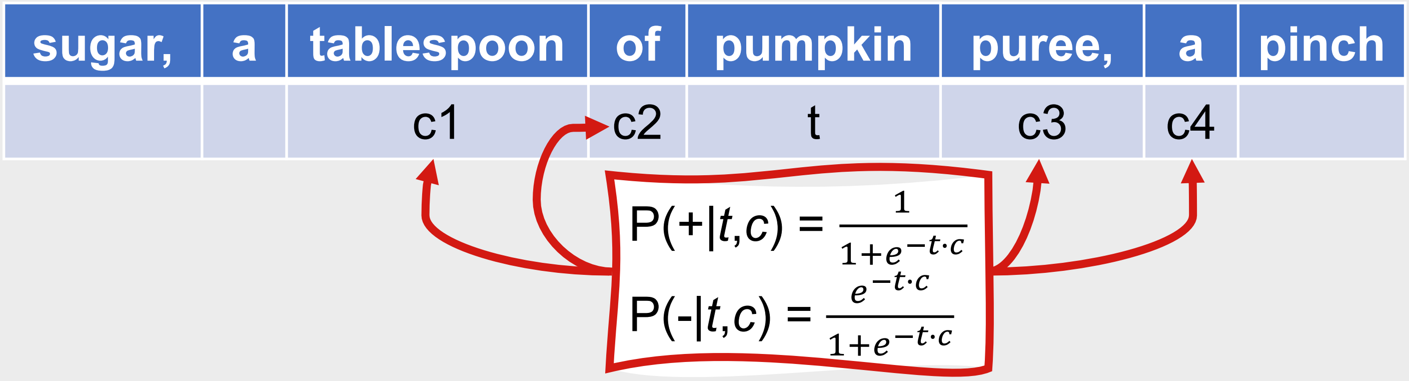
Classification Task

- The sigmoid function doesn’t automatically return a probability---it just returns a number between 0 and 1
  - To make it a probability, we need to make sure that the sum of the values returned for our two possible events (c is or is not a real context word) equals 1.0
    - P(+|t,c) = $\frac{1}{1+e^{-t\cdot c}}$
    - P(-|t,c) = 1 - P(+|t,c) = $\frac{e^{-t\cdot c}}{1+e^{-t\cdot c}}$

sugar,	a	tablespoon	of	pumpkin	puree,	a	pinch
		c1	c2	t	c3	c4	

Classification Task

- This gives us the probability for one word …but we have multiple context words in our window. How do we take all of them into account?
  - Simplifying assumption: All context words are independent
  - So, we can just multiply their probabilities:
    - P(+|t,$c_{1:k}$) = $\prod_{i=1}^{k} \frac{1}{1+e^{-t\cdot c_i}}$, or
    - log P(+|t,$c_{1:k}$) = $\sum_{i=1}^{k} \log \frac{1}{1+e^{-t\cdot c_i}}$



sugar,	a	tablespoon	of	pumpkin	puree,	a	pinch
		c1	c2	t	c3	c4	

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

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

Classification Task

- Thus, given a test target word t and a context window of k words $c_{1:k}$, we can assign a probability based on how similar the context window is to the target word
- The probability that we assign is based on applying the logistic function to the dot product of the embeddings of t with each context word c

sugar,	a	tablespoon	of	pumpkin	puree,	a	pinch
		c1	c2	t	c3	c4	
		P(+|*pumpkin*, *tablespoon*) = .7	P(+|*pumpkin*, *of*) = .5		P(+|*pumpkin*, *puree*) = .9	P(+|*pumpkin*, *a*) = .5	

$$\boxed{P(+|t,c_{1:k}) = .7 * .5 * .9 * .5 = .1575}$$

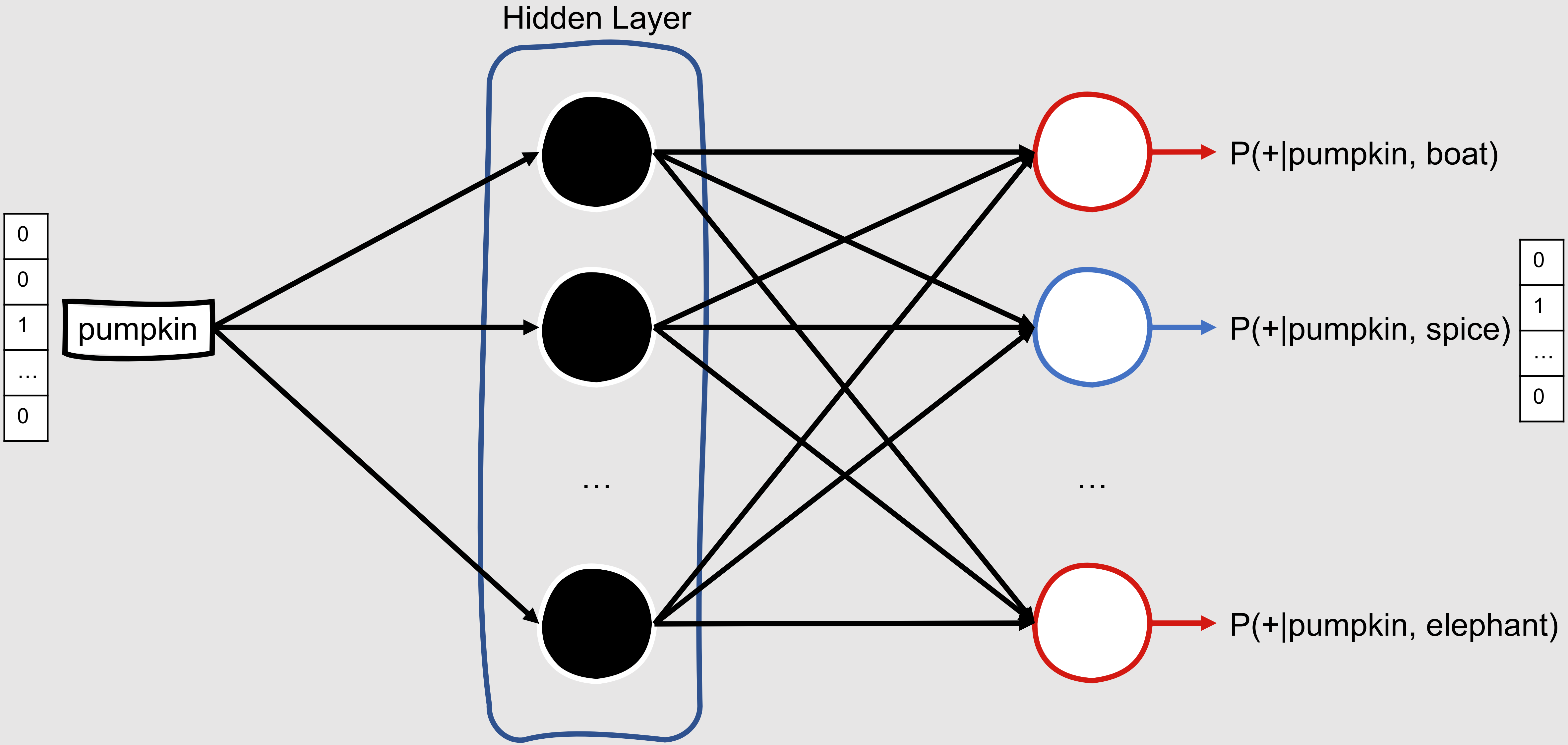
But, wait …where did we get our embeddings for each word?

- These embeddings are **learned over time**
- (Remember, this is the real goal of training the classifier in the first place!)

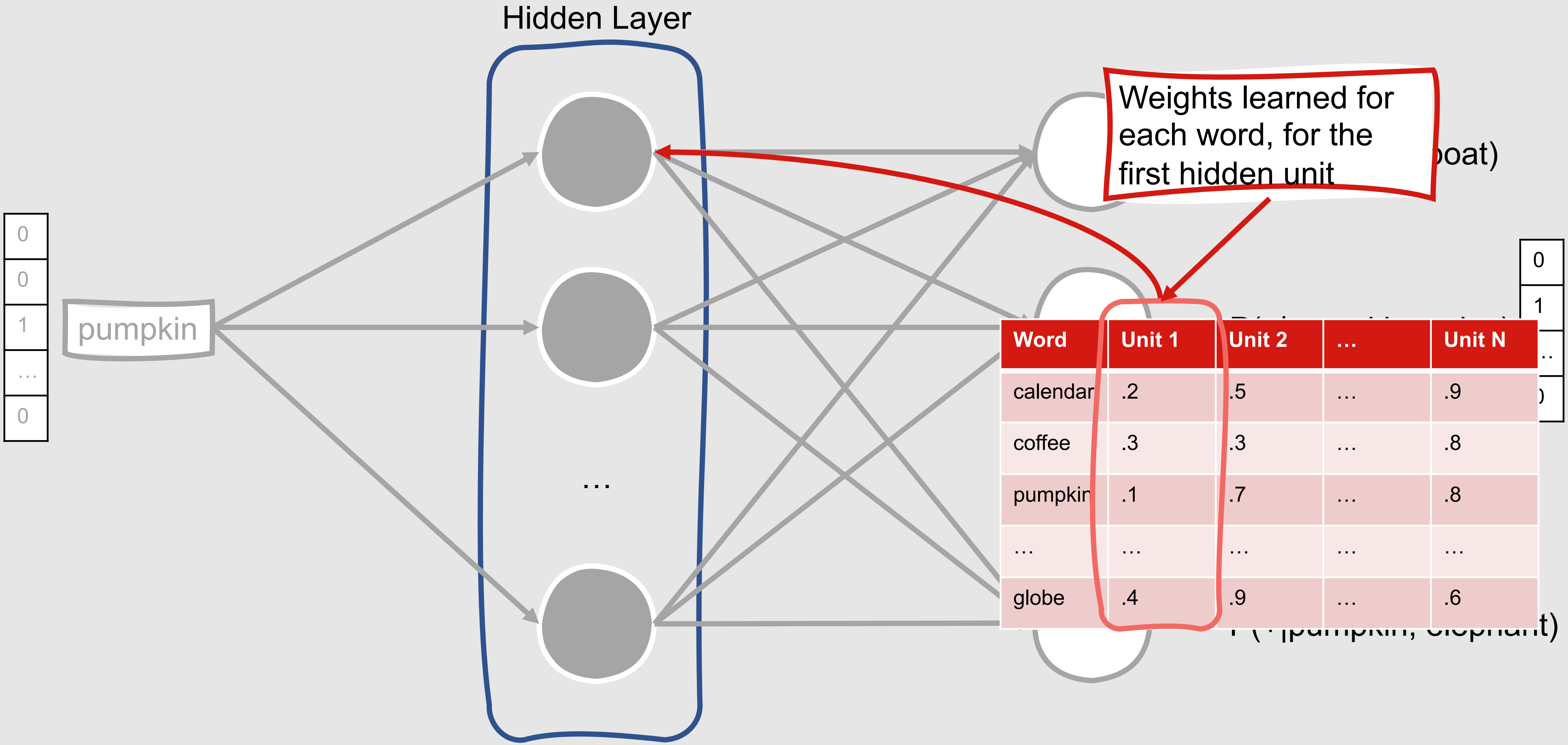
Learning Skip-Gram Embeddings

- Clarification Point:
  - There are two different types of embeddings that are referred to when discussing Word2Vec
    - Input/output of the model
    - Hidden weights
- Input and output words are represented as **one-hot vectors**
  - Bag-of-words approach: Place a "1" in the position corresponding to a given word, and a "0" in every other position
- Learned embeddings are the weights for components of the classifier's **hidden layer**, learned for each input vector

Learning Skip-Gram Embeddings



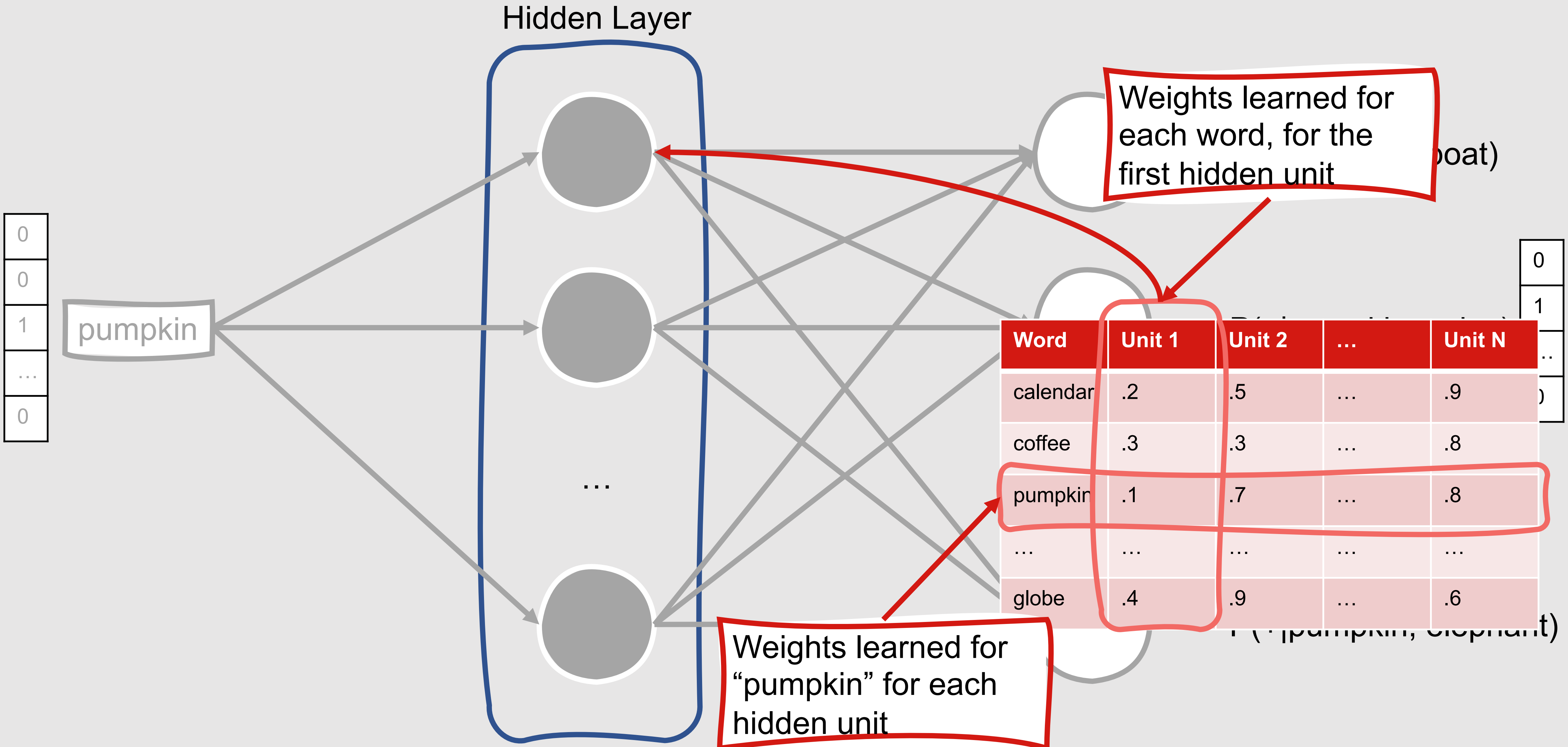
Learning Skip-Gram Embeddings



Weights learned for each word, for the first hidden unit

Word	Unit 1	Unit 2	…	Unit N
calendar	.2	.5	…	.9
coffee	.3	.3	…	.8
pumpkin	.1	.7	…	.8
…	…	…	…	…
globe	.4	.9	…	.6

Learning Skip-Gram Embeddings

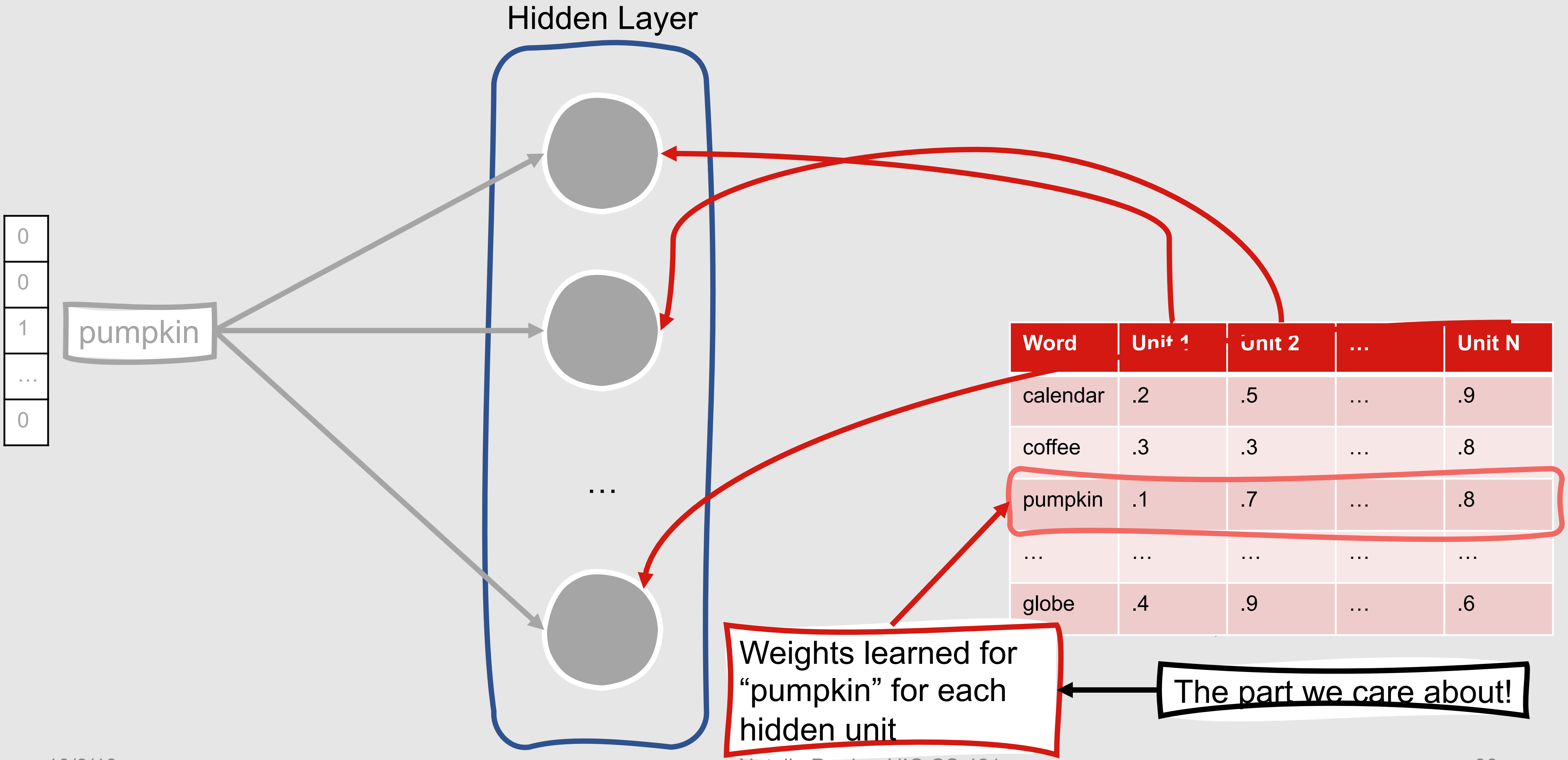


Weights learned for each word, for the first hidden unit

Word	Unit 1	Unit 2	…	Unit N
calendar	.2	.5	…	.9
coffee	.3	.3	…	.8
pumpkin	.1	.7	…	.8
…	…	…	…	…
globe	.4	.9	…	.6

Weights learned for “pumpkin” for each hidden unit

Learning Skip-Gram Embeddings



Word	Unit 1	Unit 2	…	Unit N
calendar	.2	.5	…	.9
coffee	.3	.3	…	.8
pumpkin	.1	.7	…	.8
…	…	…	…	…
globe	.4	.9	…	.6

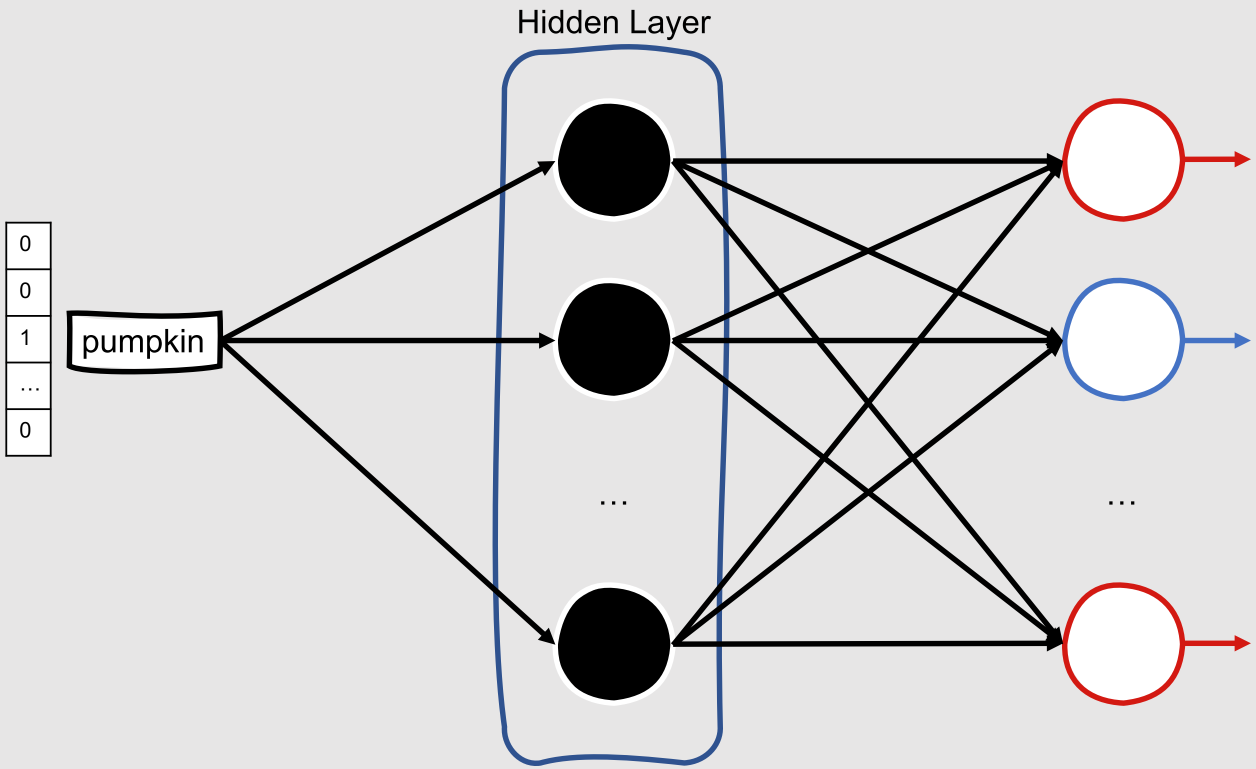
Weights learned for "pumpkin" for each hidden unit

The part we care about!

Learning Skip-Gram Embeddings

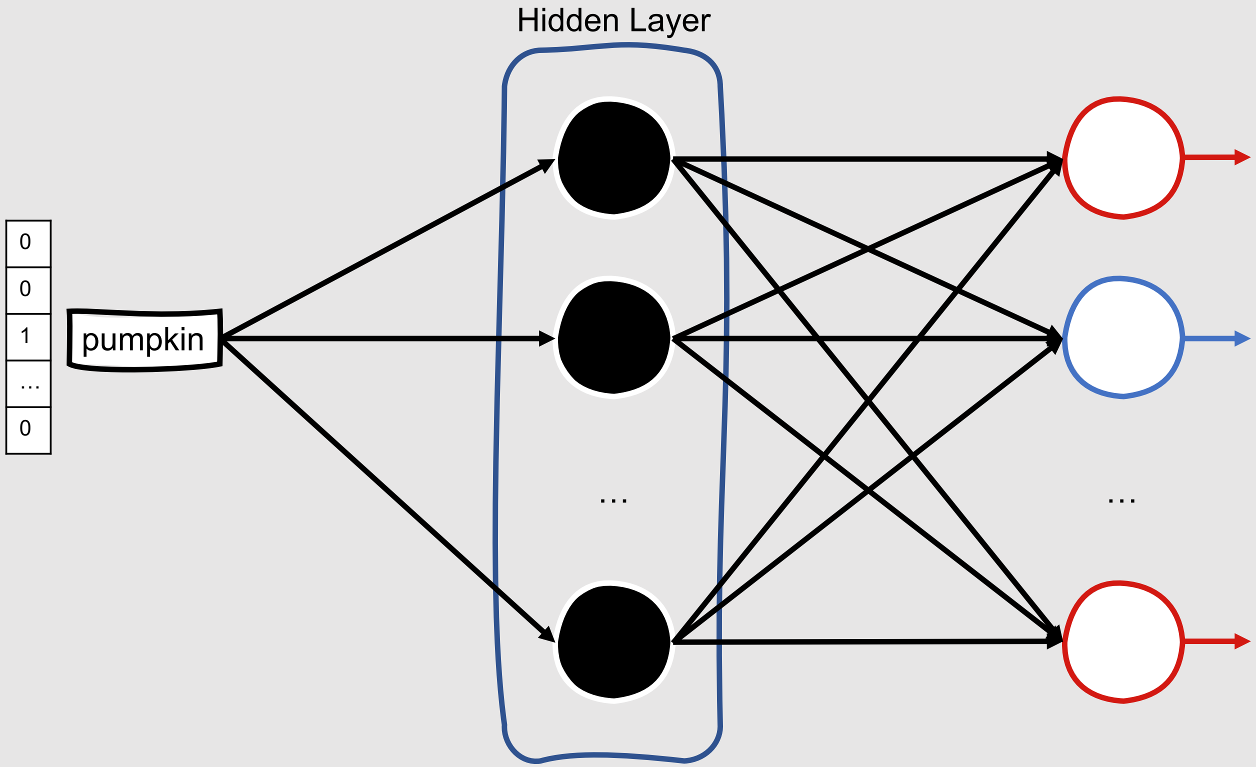
- The weights in the hidden layer of the classifier are **initialized** to some random value for each word
- They are then iteratively updated to be more similar for words that occur in similar contexts in the training set, and less similar for words that do not
  - Specifically, we want to find weights that maximize P(+|*t*,*c*) for words that occur in similar contexts and minimize P(+|*t*,*c*) for words that do not, given the information we have at the time
- Note that throughout this process, we're actually maintaining two sets of hidden weight vectors
  - One for the input (the target words)
  - One for the output (the context words)

Since we initialize our weights randomly, the classifier's first prediction will almost certainly be wrong.



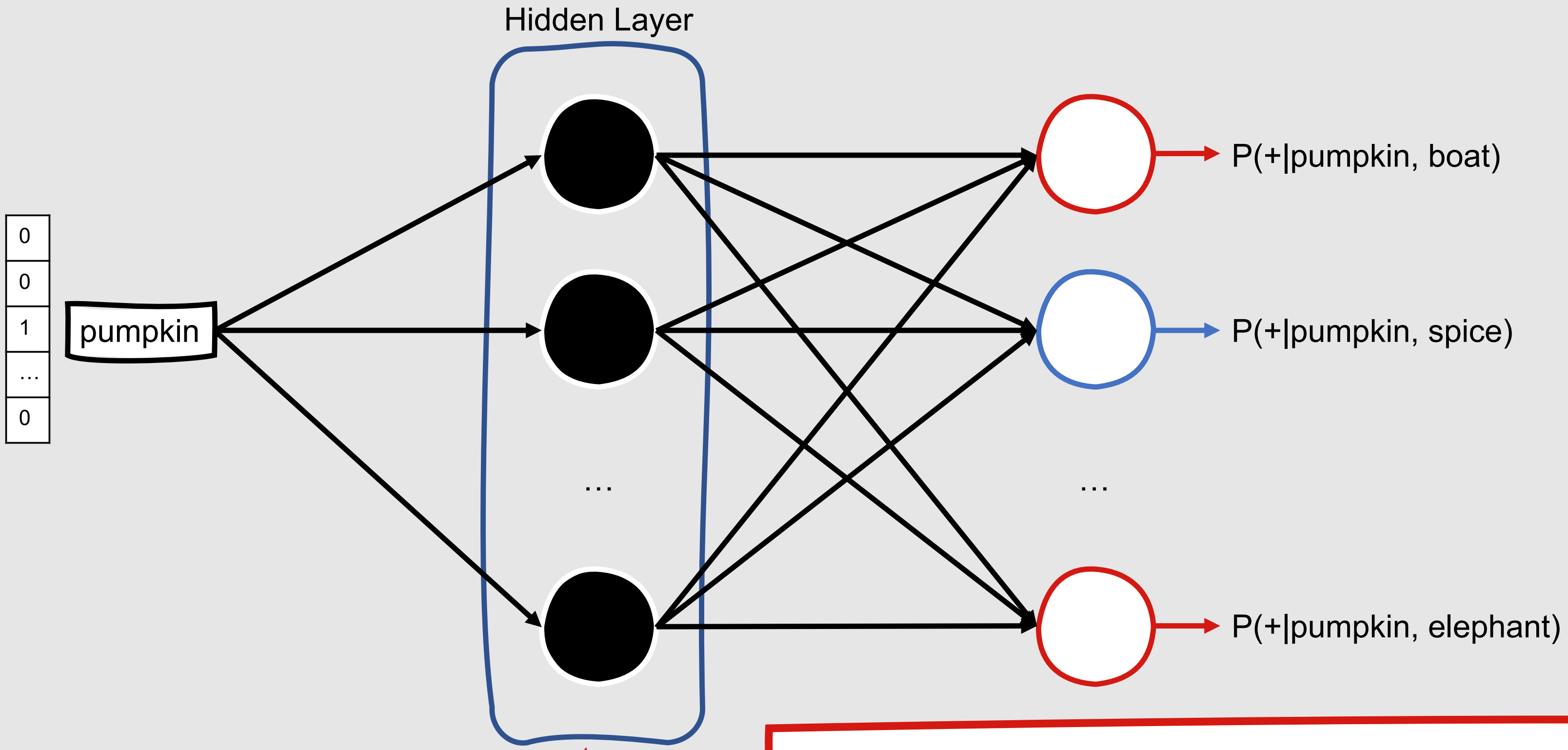
	Predicted	Actual
P(+|pumpkin, boat)	.9	0
P(+|pumpkin, spice)	.2	1
…		
P(+|pumpkin, elephant)	.6	0

However, the error values from our incorrect guesses are what allow us to improve our embeddings over time.



	Predicted	Actual	Error
P(+|pumpkin, boat)	.9	0	-.9
P(+|pumpkin, spice)	.2	1	.8
P(+|pumpkin, elephant)	.6	0	-.6

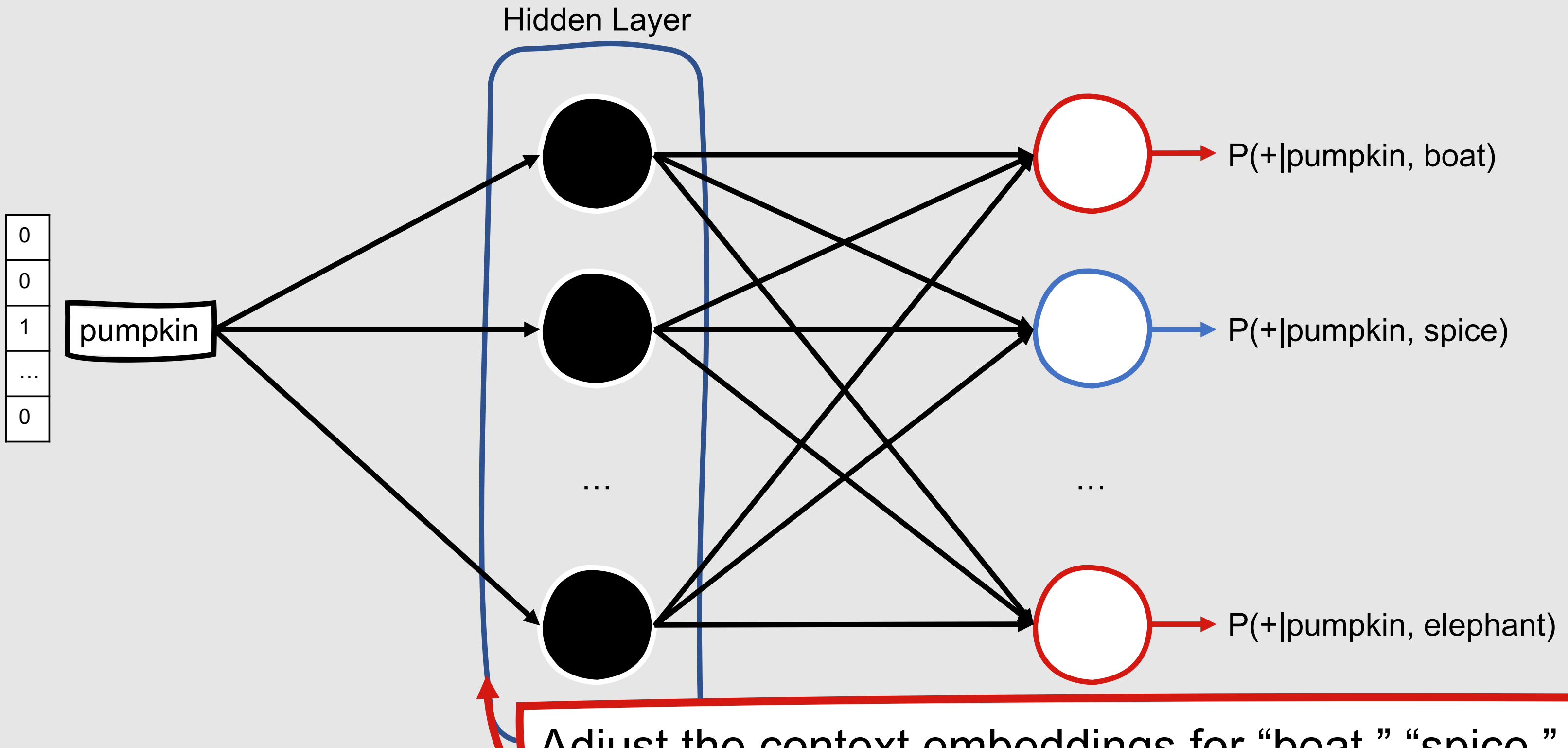
However, the error values from our incorrect guesses are what allow us to improve our embeddings over time.



	Predicted	Actual	Error
P(+|pumpkin, boat)	.9	0	-.9
P(+|pumpkin, spice)	.2	1	.8
P(+|pumpkin, elephant)	.6	0	-.6

Adjust the target embedding for “pumpkin” so if we tried to make these predictions again, we’d have lower error values

However, the error values from our incorrect guesses are what allow us to improve our embeddings over time.



Predicted	Actual	Error
.9	0	-.9
.2	1	.8
.6	0	-.6

Adjust the context embeddings for "boat," "spice," and "elephant" so if we tried to make these predictions again, we'd have lower error values

Training Data

- We are able to assume that all occurrences of words in similar contexts in our training corpus are **positive samples**

sugar,	a	tablespoon	of	pumpkin	puree,	a	pinch
		c1	c2	t	c3	c4	

Positive Examples

t	c
pumpkin	tablespoon
pumpkin	of
pumpkin	puree
pumpkin	a

Training Data

- However, we also need negative samples!
- Word2Vec actually uses more negative samples than positive samples (the exact ratio can vary)
- We need to create our own negative examples

Positive Examples

t	c
pumpkin	tablespoon
pumpkin	of
pumpkin	puree
pumpkin	a

Training Data

- How to create negative examples?
  - Target word + “noise” word that is sampled from the training set
  - Noise words are chosen according to their weighted unigram frequency $p_\alpha(w)$, where α is a weight:
    - $p_\alpha(w) = \frac{\text{count}(w)^\alpha}{\sum_{w'} \text{count}(w')^\alpha}$

Positive Examples

t	c
pumpkin	tablespoon
pumpkin	of
pumpkin	puree
pumpkin	a

Training Data

- How to create negative examples?
  - Often, $\alpha = 0.75$ to give rarer noise words slightly higher probability of being randomly sampled
- Assuming we want twice as many negative samples as positive samples, we can thus randomly select noise words according to weighted unigram frequency

Negative Examples

t	c
pumpkin	calendar
pumpkin	exam
pumpkin	loud
pumpkin	bread
pumpkin	cellphone
pumpkin	enemy
pumpkin	penguin
pumpkin	drive

Positive Examples

t	c
pumpkin	tablespoon
pumpkin	of
pumpkin	puree
pumpkin	a

Learning Skip-Gram Embeddings

- It is with these samples that the algorithm:
  - Maximizes the similarity of the target, context pairs drawn from positive examples
  - Minimizes the similarity of the target, context pairs drawn from negative examples
- This **objective** is formally expressed over the whole training set as:
  - $L(\theta) = \sum_{(t,c)\in+} \log P(+|t,c) + \sum_{(t,c)\in-} \log P(-|t,c)$
- For a specific target, context pair having *k* noise words $\mathrm{n}_1,\ldots,\mathrm{n}_\mathrm{k}$, this objective would be:
  - $L(\theta) = \log P(+|t,c) + \sum_{i=1}^{k} \log P(-|t,n_i)$

    $= \log \sigma(c \cdot t) + \sum_{i=1}^{k} \log \sigma(-n_i \cdot t) = \log \frac{1}{1+e^{-c\cdot t}} + \sum_{i=1}^{k} \log \frac{1}{1+e^{n_i\cdot t}}$
- The above means that we want to maximize the dot product of the word with the actual context words, and minimize the dot products of the word with the noise words

Learning Skip-Gram Embeddings

- How do we train to this objective?
  - **Stochastic gradient descent** (make small adjustments to the learned embeddings after each example is used)
- Remember, we start with randomly initialized embeddings
- We then iterate through our training corpus, using stochastic gradient descent to update our hidden target and context weight vectors after each sample to maximize our **objective function**
  - This means that the vectors function as the **parameters** θ that our **logistic regression** function is **tuning**

Learning Skip-Gram Embeddings

Even though we're maintaining two embeddings for each word during training (the target vector and the context vector), we only need one of them

When we're finished learning the embeddings, we can just discard the context vector

Alternately, we can add them together to create a summed embedding of the same dimensionality, or we can concatenate them into a longer embedding with twice as many dimensions

Context window size can impact performance.

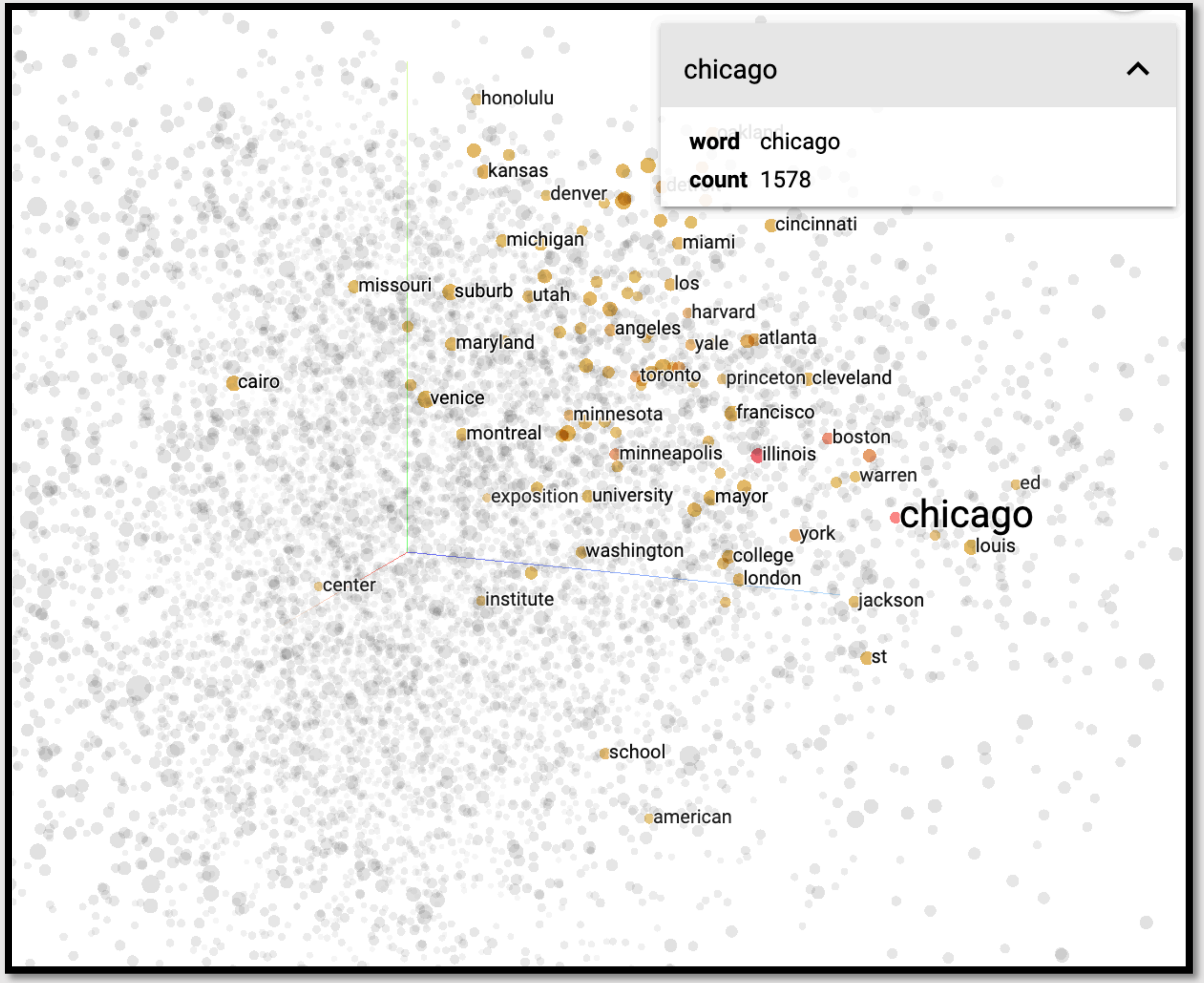
- Because of this, context window size is often tuned on a development set
- Larger window size → more required computations (important to consider when using very large datasets!)

How can we visualize (> 2-dimensional) word embeddings?

- Several approaches based on overall vector similarity:
  - List the **most similar words** to a given word, w, by sorting the vectors for all other words by their cosine similarity with the vector for w
    - chicago: illinois, boston, philadelphia, minneapolis, harvard, york, toronto, california, yale, minnesota, angeles, detroit, atlanta
    - student: students, school, teacher, graduate, professional, college, university, education, undergraduate, friend, professor, faculty, newspaper
  - Use a **hierarchical clustering algorithm** to show which words are similar to others
    - location → city → chicago

How can we visualize (> 2-dimensional) word embeddings?

- Other approaches project embeddings into lower-dimensional (e.g., 2- or 3-dimensional) spaces
  - Principal component analysis (PCA)
  - T-Distributed Stochastic Neighbor Embedding (T-SNE)
  - Uniform Manifold Approximation and Projection (UMAP)
- Play around with visualization here: http://projector.tensorflow.org/



chicago

word chicago

count 1578

Semantic Properties of Embeddings

- Major advantage of dense word embeddings: Ability to capture elements of meaning
- Context window size impacts what type of meaning is captured
  - Shorter context window → more **syntactic representations**
    - Information is from immediately nearby words
    - Most similar words tend to be semantically similar words with the same parts of speech
  - Longer context window → more **topical representations**
    - Information can come from longer-distance dependencies
    - Most similar words tend to be topically related, but not necessarily similar (e.g., waiter and menu, rather than spoon and fork)

Two Types of Similarity

- **First-Order Co-Occurrence:** Two words are typically nearby each other
  - *wrote* is a first-order associate of *book*
- **Second-Order Co-Occurrence:** Two words have similar neighbors
  - *wrote* is a second-order associate of *said*

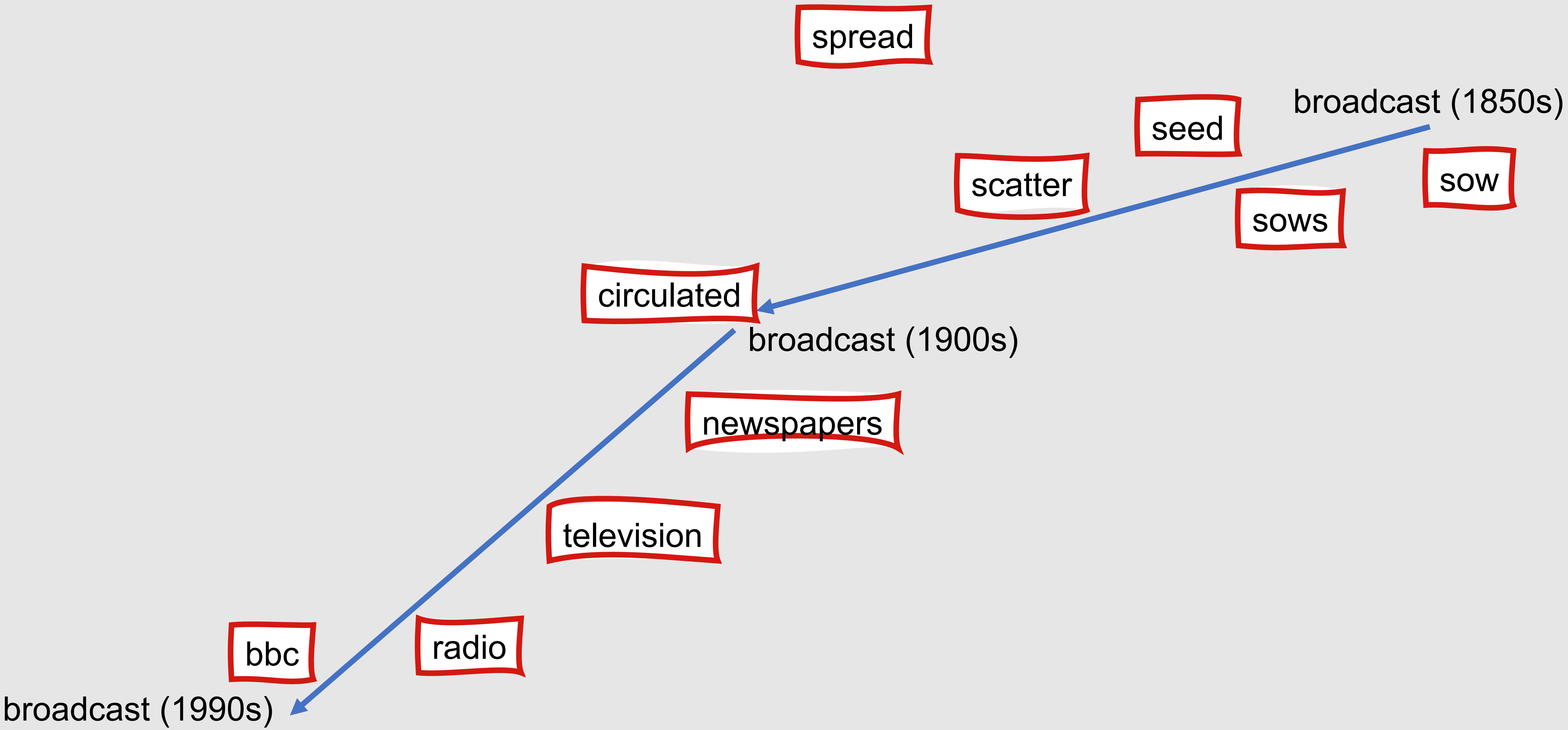
Analogy

- Word embeddings can also capture **relational meanings**
- This is done by computing the offsets between values in the same columns for different vectors
- Famous examples (Mikolov et al., 2013; Levy and Goldberg, 2014):
  - king - man + woman = queen
  - Paris - France + Italy = Rome

Embeddings and Historical Semantics

- Embeddings can also be useful for studying how meaning changes over time
- How?
  - Compute multiple embedding spaces
  - Each space is computed using only texts from a certain historical period
- Useful corpora for this:
  - Google N-grams: https://books.google.com/ngrams
  - Corpus of Historical American English: https://www.english-corpora.org/coha/

Embeddings and Historical Semantics



Bias and Embeddings

The good:

- Word embeddings automatically learn semantic properties and relationships from text

The bad:

- They also end up reproducing the implicit biases and stereotypes that are latent in the text

Bias and Embeddings

- Recall: king - man + woman = queen
- Word embeddings trained on news corpora also produce:
  - man - computer programmer + woman = homemaker
  - doctor - father + mother = nurse
- Issues like these are problematic in real-world applications!
  - E.g., algorithms may automatically assign lower scores to resumes containing common female names when ranking them for technical positions

Bias and Embeddings

- Embeddings also encode **implicit associations**
- Many implicit associations are harmless, and even useful for sentence processing
  - flowers → pleasant
  - roaches → unpleasant
- However, other implicit associations are very harmful
  - Caliskan et al. (2017) identified the following known, harmful implicit associations in GloVe embeddings:
    - African-American names were more closely associated with unpleasantness than European-American names
    - Male names were more closely associated with mathematics than female names
    - Female names were more closely associated with the arts than male names
    - Names common among older adults were more closely associated with unpleasantness than those common among younger adults

Bias and Embeddings

- Thus, learning word representations poses an increasingly important ethical dilemma!
- Recent research has begun examining ways to **debias** word embeddings by:
  - Developing transformations of embedding spaces that remove gender stereotypes but preserves definitional gender
  - Changing training procedures to eliminate these issues before they arise
- Although these methods reduce bias, they do not eliminate it

Bias and Embeddings + Historical Semantics

- An interesting fusion of historical semantics with research on implicit biases in word embeddings: **How have people's biases shifted over time?**
- Historical corpora show that the cosine similarity between women's versus men's names and common occupations correlate with the historical percentages of men and women in those occupations
- These same corpora replicate trends in both ethnic and gender biases over time
  - Adjectives related to competence (e.g., smart, wise, thoughtful, and resourceful) have historically had a higher cosine similarity with male than female names, but in English text this bias has been slowly decreasing since 1960

Evaluating Vector Models

Extrinsic Evaluation

- Add the vectors as features in a downstream NLP task, and see whether and how this changes performance relative to a baseline model
- Most important evaluation metric for word embeddings!
  - Word embeddings are rarely needed in isolation
  - They are almost solely used to boost performance in downstream tasks

Intrinsic Evaluation

- Performance at predicting word similarity

Evaluating Similarity Performance

- Compute the **cosine similarity** between vectors for pairs of words
- Compute the **correlation** between those similarity scores and word similarity ratings for the same pairs of words manually assigned by humans
- Corpora for doing this:
  - WordSim-353
  - SimLex-999
  - TOEFL Dataset
    - *Levied* is closest in meaning to: (a) imposed, (b) believed, (c) requested, (d) correlated

Other Common Evaluation Tasks

Semantic Textual Similarity

- Evaluates the performance of sentence-level similarity algorithms, rather than word-level similarity

Analogy

- Evaluates the performance of algorithms at solving analogies
  - Athens is to Greece as Oslo is to (Norway)
  - Mouse is to mice as dollar is to (dollars)

How do other word embedding methods differ from Word2Vec?

- In the context of this course, we learned about one specific type of dense word vectors
    - Skip-gram with negative sampling
- As noted, there are many others!
- Even in the original Word2Vec package:
    - Continuous Bag of Words (CBOW)
    - Negative sampling isn't necessary (it just makes performance *much* faster)

A (Very Brief) Overview of Other Embedding Methods

- CBOW
- GloVe
- ELMo
- BERT

Continuous Bag of Words (CBOW)

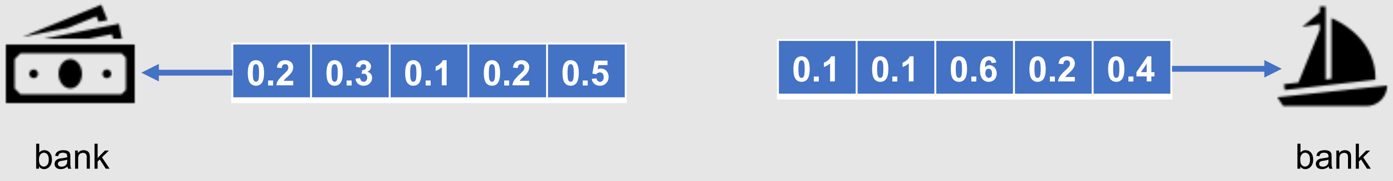
- Very similar to skip-gram model!
- The difference:
  - Instead of learning to predict a context word from a target word vector, you learn to **predict a target word from a set of context word vectors**
- Still **non-contextual**
- In general, skip-gram embeddings are good with:
  - Small datasets
  - Rare words and phrases
- CBOW embeddings are good with:
  - Larger datasets (they're faster to train)
  - Frequent words

Global Vectors for Word Representation (GloVe)

- While Word2Vec is a **predictive model** (it learns to predict whether words belong in a target word's context), GloVe is a **count-based model** (it's basically a fancy co-occurrence matrix)
- In a nutshell, GloVe embeddings are constructed by:
  - Building a huge word x context co-occurrence matrix
  - Performing dimensionality reduction on the matrix to reduce it to a more manageable size
- Still non-contextual

Embeddings from Language Models (ELMo)

- **Contextual** (predicts different vectors for different word senses)
- Does this by concatenating information from multiple layers of a **bidirectional** neural language model
- Accepts character inputs instead of words, which enables the model to predict embeddings for out-of-vocabulary words
- Predicts an embedding for a target word given its context



0.2 0.3 0.1 0.2 0.5

0.1 0.1 0.6 0.2 0.4

Bidirectional Encoder Representations from Transformers (BERT)

- Also contextual
- Learns embeddings for **subwords** (more than a character, but less than a full word)
- This allows the model to also predict embeddings for out-of-vocabulary words
- Uses a bidirectional neural language model to do this, similar to ELMo
    - The specific type of neural language model differs (**Transformer** rather than **LSTM**)

Which embeddings are best?

- It depends on your data!
- In general, Word2Vec and GloVe produce similar embeddings
- Word2Vec → slower to train but less memory intensive
- GloVe → faster to train but more memory intensive
- Word2Vec and Glove both produce context-independent embeddings
- ELMo and BERT produce context-dependent embeddings
- Both can also predict embeddings for new words
- BERT (or variants thereof) is the current state of the art
- ELMo may be better in cases with lots of obscure words that aren't easily chunked into subwords

Summary: Word Embeddings (Part 2)

- **Dense vectors** are generally better for NLP tasks
- **Word2Vec**, **GloVe**, **ELMo**, and **BERT** are all examples of dense word embeddings
- Word2Vec (specifically, the **skip-gram** variant) learns a classifier that predicts whether a word is in the context of a target word
- The weights from the hidden layer of this classifier are the learned **word embeddings**
- These embeddings are learned using a formula that maximizes the similarity between vectors for words that occur in the same context
- Word embeddings capture **semantic properties** (😀) but they also capture **harmful stereotypes** (🙁) …coming up with good debiasing methods is still an open problem
- Word embeddings are best evaluated in **extrinsic tasks**, but can also be evaluated based on their ability to accurately capture **word similarity**
- Different word embeddings are good for different tasks (in general, **BERT is the current state of the art**)