Introduction to Large Language Models

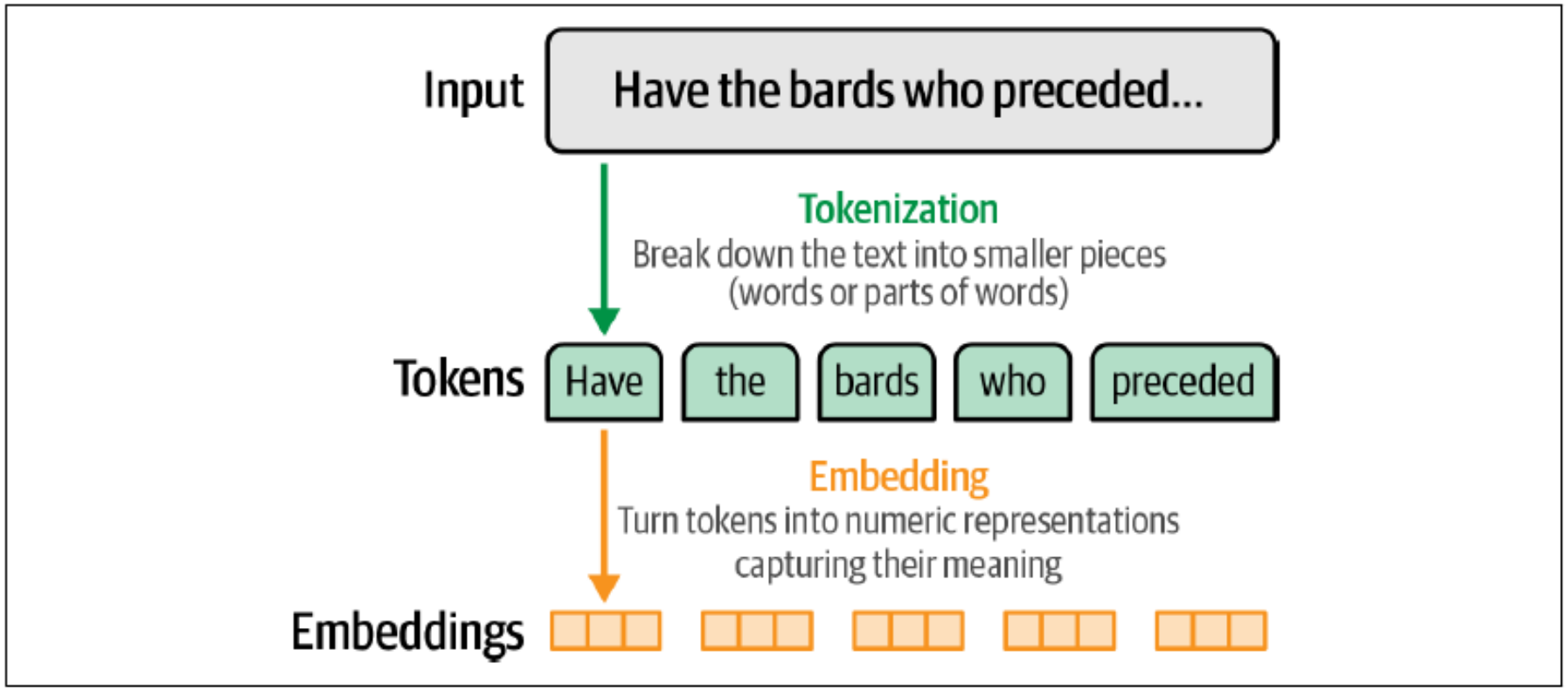
Spring 2026

Word Embeddings

(Some slides adapted from Ralph Grishman at NYU,
Yejin Choi at UWashington, N. Tomura at UDepaul, Jurafsky and
Martin, CS224N, CS224d at Stanford and other resourses on the web)

Tokens and Embeddings

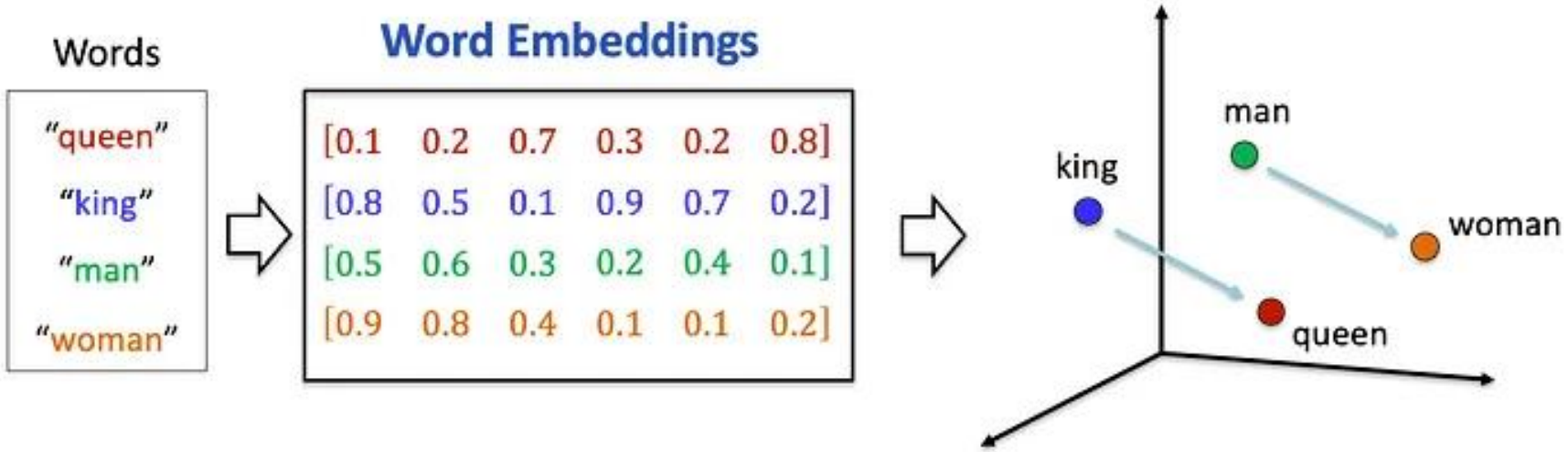
- Tokens and embeddings are two of the central concepts of using large language models (LLMs).



Source: Hands-On Large Language Models by Jay Alammar and Maarten Grootendorst

Word Representation

- Word representation (or word embedding) methods convert words into numerical vectors so that machine learning models can process language.
- These representations aim to capture statistical, syntactic and semantic relationships.



Word Representation Methods

🟦 1. Symbolic / Count-Based Representations (Classical Methods)

- These rely purely on **frequency counts** and **statistics**—no learning, no neural networks.
- These are the traditional methods used before deep learning became dominant.

Methods:

- **One-Hot Encoding**
- **Bag of Words (BoW)**
- **TF (Term Frequency)**
- **TF-IDF**
- **Co-occurrence matrices**
- **PMI / PPMI (Pointwise Mutual Information)**
- **LSA (Latent Semantic Analysis, via SVD)**

Characteristics:

- Sparse, high-dimensional vectors
- No semantics
- No context awareness
- They capture **no semantic meaning**. The model doesn't know that "car" and "automobile" are related; they are just two different indices.

Word Representation Methods

🟦 2. Static Distributed / Prediction-Based Methods - Word Embeddings (Neural, but Context-Free)

These methods learn **dense, low-dimensional** vectors, but each word has **one fixed representation**.

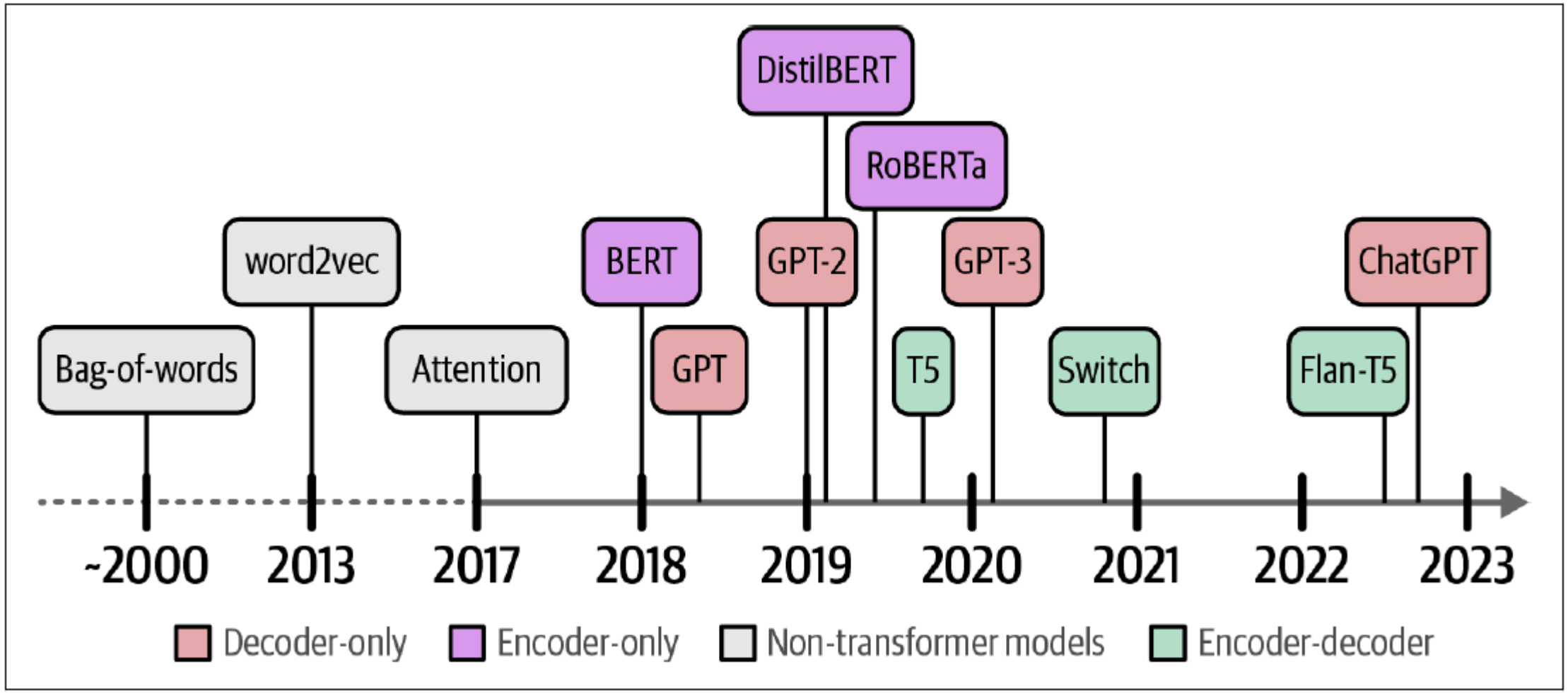
Methods:

- **Word2Vec** (CBOW, Skip-gram)
- **GloVe** (Global Vectors)
- **FastText** (subword-enhanced)

Characteristics:

- Capture semantic similarity
- Dense vectors (50–300 dims)
- Still **static**:
  - “bank” has one vector in all contexts (They cannot handle polysemy (multiple meanings). The word "bank" gets the same vector in "river bank" and "bank account.«)

History of AI in NLP



Source: Hands-On Large Language Models by Jay Alammar and Maarten Grootendorst

Word Representation Methods

3. Contextual Word Embeddings (Deep Learning, Transformer-Based)

These generate a **different vector for the same word depending on context**.

Example:
“bank” (river) ≠ “bank” (finance)

Major Models:

- **ELMo** (bi-LSTM based)
- **BERT family** (BERT, RoBERTa, DistilBERT, etc.)
- **GPT family** (GPT-2, GPT-3, GPT-4, GPT-5 based models)
- **XLNet**
- **T5**

Characteristics:

- Dynamic, context-sensitive
- High performance in modern NLP
- Based on **Transformers**
- Provides the deepest level of language understanding, capturing sarcasm, homonyms, and syntactic nuances.

Word Representation Methods

4. Sentence and Document-Level Embeddings

These go beyond words and represent entire chunks of text.

Methods:

- **Doc2Vec**
- **Sent2Vec**
- **Universal Sentence Encoder (USE)**
- **Sentence-BERT (SBERT)**
- **InferSent**

Characteristics:

- Useful for semantic similarity, retrieval, clustering
- Represent longer semantics beyond individual words

Word Representation Methods Summary

Category	What It Uses	Examples	Context-Aware?	Typical Use
Count-Based	Frequency & statistics	BoW, TF-IDF, PMI	❌ No	IR, text classification
Static Embeddings	Neural, fixed vectors	Word2Vec, GloVe	❌ No	Similarity, clustering
Contextual Embeddings	Deep learning, Transformers	BERT, GPT, ELMo	✔️ Yes	Modern NLP tasks
Sentence/Document Embeddings	Models beyond word level	USE, SBERT, Doc2Vec	✔️ Yes	Retrieval, ranking

Statistical, Count-based Word Representation Methods

- **Statistical, count-based word representation methods** are the earliest approaches in NLP for converting text into numerical vectors.
  They rely purely on **statistics**, **counts**, and **co-occurrence frequencies**—not on neural networks or learned embeddings.
- These methods are **static**, **sparse**, and **context-independent**.

Key Techniques:

- **One-Hot Encoding:** A vector with the length of the entire vocabulary, where a single `1` marks the word and the rest are `0`s.
- **Bag of Words (BoW):** Counts how many times a word appears in a document, disregarding grammar and word order.
- **TF-IDF (Term Frequency-Inverse Document Frequency):** A statistical measure that evaluates how important a word is to a document in a collection. It penalizes words like "the" that appear everywhere and rewards rare, specific words.

One-Hot Encoding

- Each word is represented as a sparse vector full of zeros except for one “1”.
- Vocabulary size = vector size.
- Example (vocabulary = [cat, dog, apple]):

  cat -> [1, 0, 0]

  dog -> [0, 1, 0]

  apple -> [0, 0, 1]

Pros: Simple
Cons:

- No notion of similarity (cat and dog are orthogonal)
- Huge vectors for large vocabularies
- No context information

Bag of Words (BoW)

- A representation where each document is converted into a vector of **raw word counts**.
- **How it works:**
  - Build a vocabulary of all unique words
  - For each document → count how many times each word appears
- Example:
- Imagine we have a tiny library containing only three documents (sentences):
  - **Document 1:** "I love dogs."
  - **Document 2:** "I love cats."
  - **Document 3:** "I love dogs and cats.«
- Document/Term Matrix

Document	I	love	dogs	cats	and
Doc 1	1	1	1	0	0
Doc 2	1	1	0	1	0
Doc 3	1	1	1	1	1

Term-document matrix

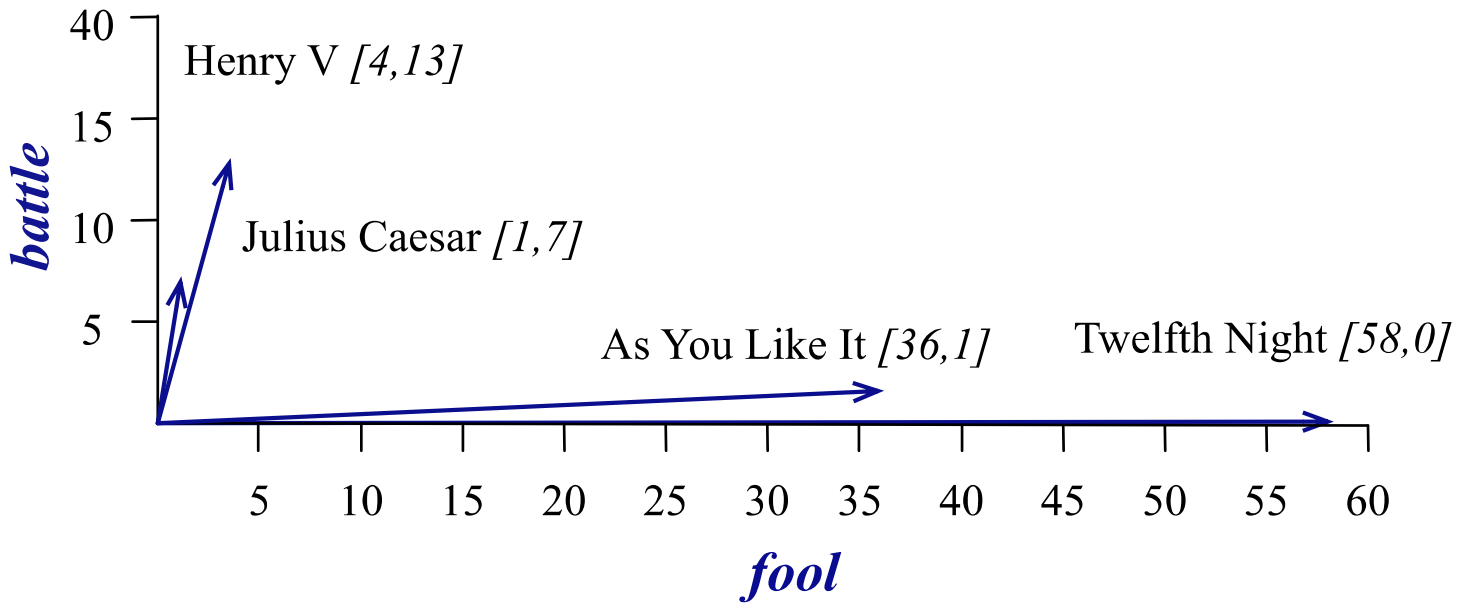
- In a term-document matrix, each row represents a word in the vocabulary and each column represents a document from some collection of documents.

- Each document is represented by a vector of words. The vector for a document is a point in |V| 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

- The term-document matrix for four words in four Shakespeare plays.
- The redboxes show that each document is represented as a column vector of length four.

Visualizing document vectors



- A spatial visualization of the document vectors for the four Shakespeare play documents, showing just two of the dimensions, corresponding to the words *battle* and *fool*.
- The comedies have high values for the *fool* dimension and low values for the *battle* dimension.

Vectors are the basis of information retrieval

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

Term-document matrices are used to find similar documents for the task of ***information retrieval***.

–Two documents that are similar tend to have similar words.

Vectors are similar for the two comedies (*As You Like It* [1,114,36,20] and *Twelfth Night* [0,80,58,15])

Comedies have more *fools* and *wit* and fewer *battles*.

Idea for word meaning: Words can be vectors too!!!

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

The **word vector** is a row vector rather than a column vector.

–The four dimensions of the vector for **fool**, **wit**, **battle**, **good**.

–Each entry in the vector thus represents the counts of the word's occurrence in the document corresponding to that dimension.

- *battle* is "the kind of word that occurs in Julius Caesar and Henry V"
- *fool* is "the kind of word that occurs in comedies, especially Twelfth Night"

Word-Word Matrix

- Rather than the term-document matrix, we use the **term-term matrix**, more commonly called the **word-word matrix** (or the ***term-context matrix***) in which the columns are labeled by words rather than documents.
- **–The matrix is |V|x|V| matrix and each cell records number of times the row (target) wordand column (context) word co-occur in some context in some training corpus.**
  - The context could be the document, in which case the cell represents the number of times the two words appear in the same document.
  - It is most common to use smaller contexts: generally a window around the word, for example of 4 words to left and 4 words to right, in which case the cell represents number of times column word occurs in such a ±4 word window around row word.
- Each row in word-word matrix is **co-occurrence vector (context vector) of that row (target) word**.
- Two words are **similar in meaning** if **their context vectors are similar**.

More common: word-word matrix (or "term-context matrix")

- Two **words** are similar in meaning if their context vectors are similar

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	...

- Note in previous table (Fig. 6.6) that the two words cherry and strawberry are more similar to each other (both pie and sugar tend to occur in their window) than they are to other words like digital;
- Conversely, digital and information are more similar to each other than, say, to strawberry. Fig. 6.7 shows a spatial visualization.

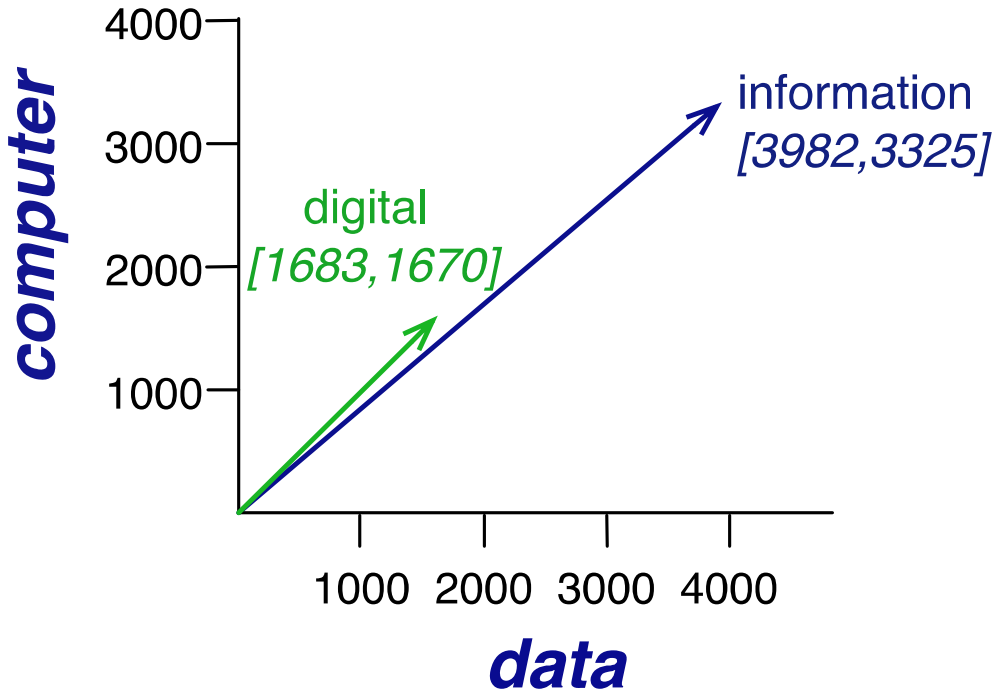


Figure 6.7 A spatial visualization of word vectors for digital and information, showing just two of the dimensions, corresponding to the words data and computer.

	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	...

Cosine for computing word similarity

. To measure similarity between two words, we need a metric that compares two vectors. By far the most common similarity metric is the cosine of the angle between the vectors

Computing word similarity: Dot product and cosine

- To define similarity between two target words **v** and **w**, we need a measure for taking two such vectors and giving a measure of vector similarity.
- The **most common similarity metric** is the **cosine of the angle between the vectors**.
- The **cosine** is based on **dot product** operator, also called **inner product**:
- The dot product between two vectors is a scalar:

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

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

Problem with raw dot-product

- The **raw dot-product** *has a problem as a similarity metric*: it favors long vectors.
- Dot product is higher if a vector is longer (has higher values in many dimension)
- Vector length: $$|\mathbf{v}| = \sqrt{\sum_{i=1}^{N} v_i^2}$$
- Frequent words (of, the, you) have long vectors (since they occur many times with other words).
- The raw dot product will be higher for frequent words.
- But this is a problem; we'd like a similarity metric that tells us how similar two words are regardless of their frequency.

Alternative: cosine for computing word similarity

We modify the dot product to normalize for the vector length by dividing the dot product by the lengths of each of the two vectors. This normalized dot product turns out to be the same as the cosine of the angle between the two vectors (Geometrically, it is the product of the Euclidean magnitudes of the two vectors and the cosine of the angle between them).

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

Based on the definition of the dot product between two vectors a and b

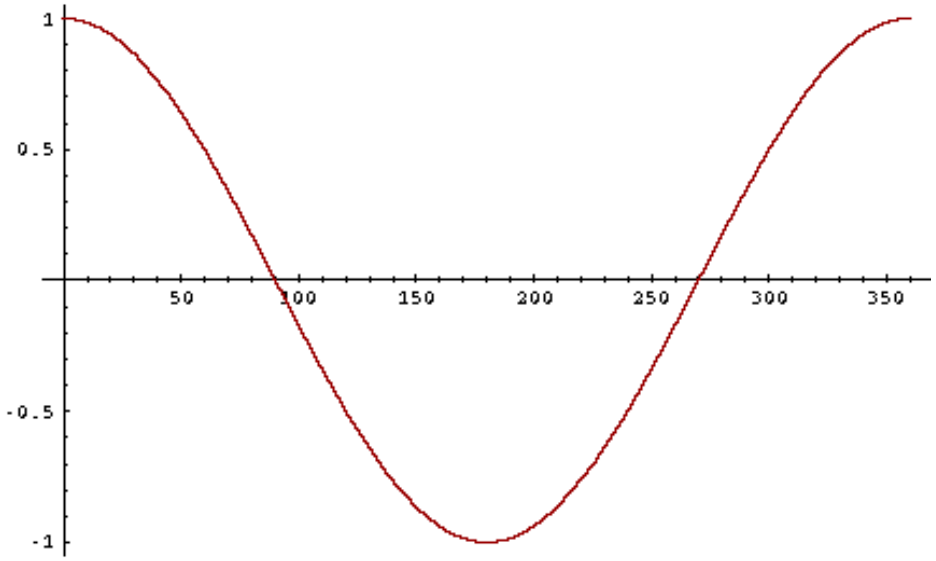
$$\mathbf{a}\cdot\mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$$

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

Cosine as a similarity metric

Cosine Values:

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



- The cosine value ranges from **1 for vectors pointing in the same direction**, through **0 for vectors that are orthogonal**, *to -1 for vectors pointing in opposite directions*.
- –But raw frequency values are non-negative, so cosine for these vectors ranges from 0 to 1.

Cosine examples

$$\cos(\vec{v},\vec{w}) = \frac{\vec{v}\,\square\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\vec{v}}{|\vec{v}|}\,\square\frac{\vec{w}}{|\vec{w}|} = \frac{\square_{i=1}^{N} v_i w_i}{\sqrt{\square_{i=1}^{N} v_i^2}\sqrt{\square_{i=1}^{N} w_i^2}}$$

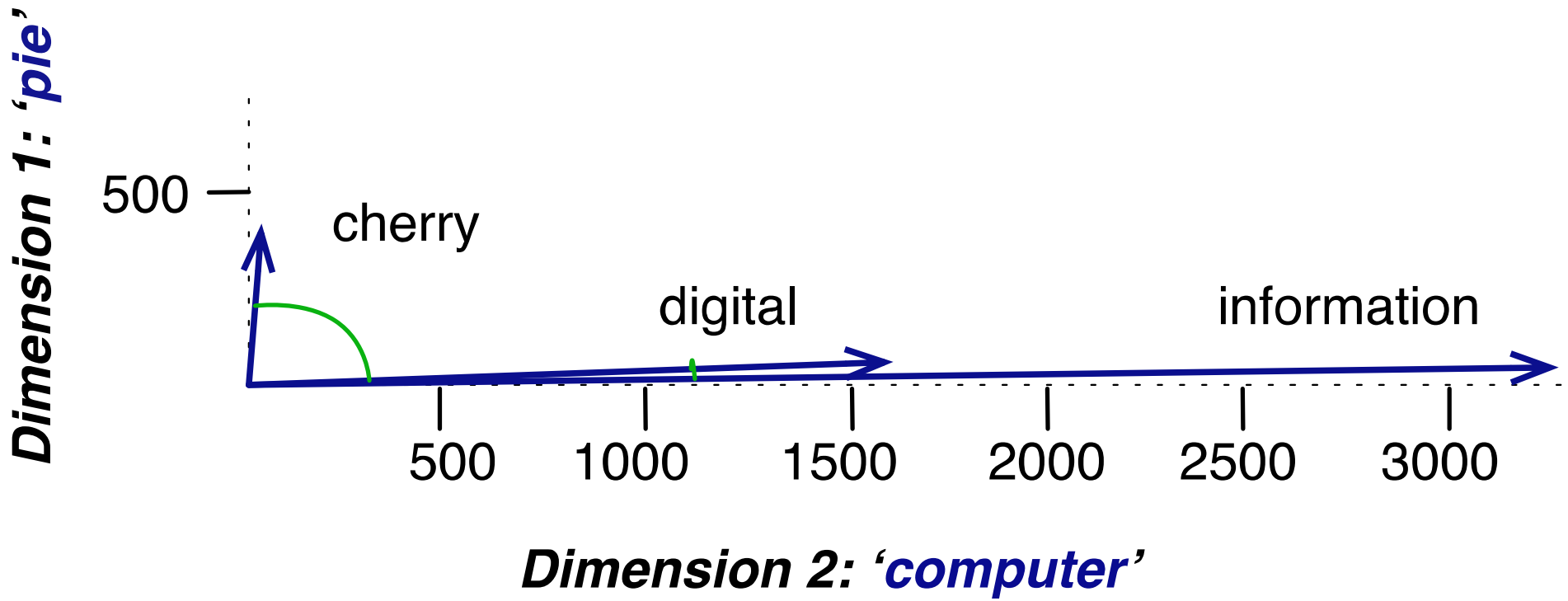
	pie	data	computer
cherry	442	8	2
digital	5	1683	1670
information	5	3982	3325

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

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

The model decides that *information* is way closer to *digital* than it is to *cherry*, a result that seems sensible.

Visualizing cosines



Note that the angle between *digital* and *information* is smaller than the angle between *cherry* and *information*. When two vectors are more similar, the cosine is larger but the angle is smaller; the cosine has its maximum (1) when the angle between two vectors is smallest (0◦); the cosine of all other angles is less than 1.

TF-IDF Weighting

- A common way to reweight counts in term-document matrices
- Information Retrieval workhorse!
- A common baseline model
- Sparse vectors
- The meaning of a word is defined by a simple function of the counts of nearby words.

But raw frequency is a bad representation

- 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?
- Although frequency is important, but simple frequency isn't the best measure of association between words
- **We need a function that resolves this frequency paradox!**
  - **One solution is TF-IDF (term frequency _ inverse document frequency)**
  - TF-IDF value (the '-' here is a hyphen, not a minus sign) is the product of two terms, each term capturing one of these two intuitions.

Common solutions for word weighting

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

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

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

- **PMI:** (Pointwise mutual information)
  - $\mathbf{PMI}(w_1, w_2) = log \frac{p(w_1,w_2)}{p(w_1)p(w_2)}$

See if words like "good" appear more often with "great" than we would expect by chance

Term frequency (tf)

- The first is the term frequency: the frequency of the word *t* in the document *d*. We can just use the raw count as the term frequency:

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

- More commonly we squash the raw frequency a bit, by using the log10 of the frequency instead. The intuition is that a word appearing 100 times in a document doesn't make that word 100 times more likely to be relevant to the meaning of the document. Because we can't take the log of 0, we normally add 1 to the count:

$$\text{tf}_{t,d} = \log_{10}(\text{count}(t,d)+1)$$

Document frequency (df)

- The document frequency df_t *of a term is* the number of documents *t* occurs in.
- The number of documents in a collection (corpus) that contain the term t at least once. It does not count how many times the term appears—only how many documents it appears in.
- Document frequency is not the same as the collection frequency of a term, which is the total number of times the word appears in the whole collection in any document.
- Ex:
  - Consider in the collection of Shakespeare's 37 plays the two words *Romeo* and *action*
  - "*Romeo*" is very distinctive for one Shakespeare play, but not "*action*":

	Collection Frequency	Document Frequency
Romeo	113	1
action	113	31

Inverse document frequency (idf)

- **idf: inverse document frequency: give a higher weight to words that occur only in a few documents.**
- Emphasize discriminative words like Romeo via the inverse document frequency(idf) term weight

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

N is the total number of documents in the collection

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

TF-IDF Value:

- **Tf-idf weighting of the value $w_{t,d}$ for term t in document d combines term frequency with idf:**

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

- Very frequently, you might need to break up the corpus into documents yourself for the purposes of computing idf. Documents can be
  - Could be a play, a Wikipedia article or newspaper article
  - But for the purposes of tf-idf, documents can be **anything**; we often treat each paragraph as a document!

Final tf-idf weighted value for a word

- 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: combines term frequency $\text{tf}_{t,d}$ with idf:

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

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0	0.22	0.28
good	0	0	0	0
fool	0.019	0.021	0.0036	0.0083
wit	0.049	0.044	0.018	0.022

Document Similarity

- The tf-idf model of meaning is often used for document functions like deciding if two documents are similar.
- We represent a document by taking the vectors of all the words in the document, and computing the centroid of all those vectors.
- The centroid is the multidimensional version of the mean; the centroid of a set of vectors is a single vector
- To compare two documents:
  - Take the centroid of vectors of all the words in the document
  - Given k word vectors $w_1,\ldots,w_k$, **centroid document vector d** is:

$$d = \frac{w_1 + w_2 + \ldots + w_k}{k}$$

  - Given two documents, we can then compute their centroid document vectors d_1 and d_2, and estimate the similarity between the two documents by $\cos(d_1,d_2)$.

Pointwise Mutual Information (PMI)

- An alternative weighting function to tf-idf is called **PPMI (positive pointwise mutual information)**.
- PPMI draws on the intuition that best way to weigh the association between two words is to ask how much more the two words co-occur in our corpus than we would have a priori expected them to appear by chance.

Pointwise Mutual Information

- **Pointwise mutual information**: is a measure of how often two events x and y occur, compared with what we would expect if they were independent:

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

- **The pointwise mutual information between a target word w_1 and a context word w_2** is then defined as:

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

- ➔ Weigh the association between two words is to ask how much more the two words words w_1 and w_2 co-occur more than if they were independent?

Positive Pointwise Mutual Information

- PMI ranges from $-\infty$ to $+\infty$
- But the negative values are problematic
  - Things are co-occurring **less than** we expect by chance
  - Unreliable without enormous corpora
    - Imagine w1 and w2 whose probability is each 10^{-6}
    - Hard to be sure p(w1,w2) is significantly different than 10^{-12}
  - Plus it's not clear people are good at "unrelatedness"
- So we just replace negative PMI values by 0
- Positive PMI (**PPMI**) between word1 and word2:

$$\text{PPMI}(word_1, word_2) = \max\left(\log_2 \frac{P(word_1, word_2)}{P(word_1)P(word_2)}, 0\right)$$

Computing PPMI on a term-context matrix

- We have a co-occurrence Matrix F with W rows (words) and C columns (contexts)
- f_{ij} is # of times w_i occurs in context c_j

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

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

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

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

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

- p(w=information,c=data) = 3982/111716 = .3399
- p(w=information) = 7703/11716 = .6575
- p(c=data) = 5673/11716 = .4842

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

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

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

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

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

  - Resulting PPMI matrix (negatives replaced by 0)

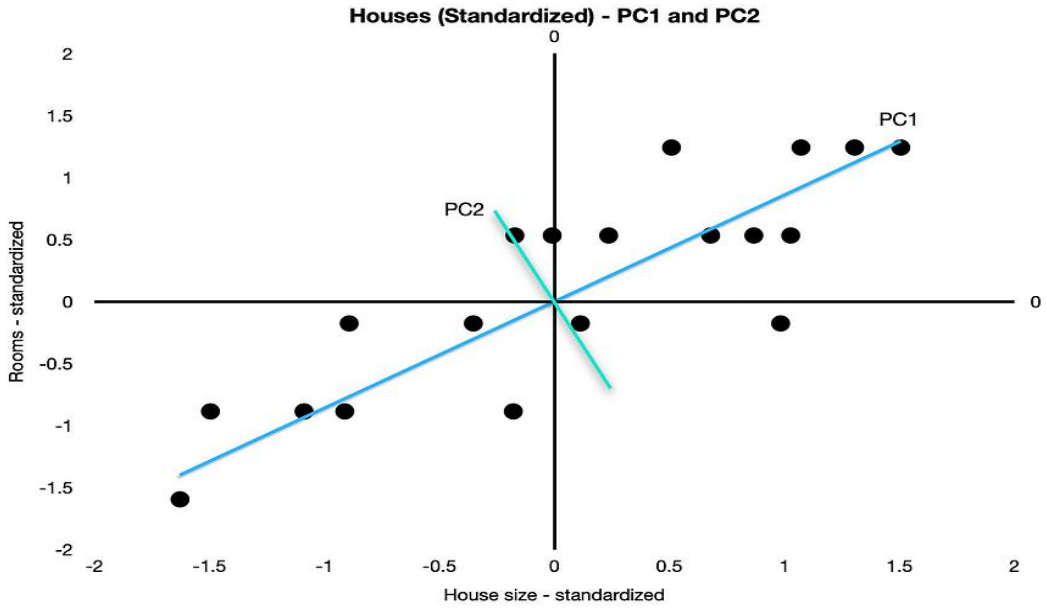
	computer	**data**	**result**	**pie**	**sugar**
cherry	0	0	0	4.38	3.30
strawberry	0	0	0	4.10	5.51
digital	0.18	0.01	0	0	0
information	0.02	0.09	0.28	0	0

Frequency-based Dense Vectors with Singular value decomposition(SVD)

- Intuition
  - Approximate an N-dimensional dataset using fewer dimensions
  - By first rotating the axes into a new space
  - The highest order dimension captures the most variance in the original dataset
  - And the next dimension captures the next most variance, etc
  - Many such (related) methods:
    - PCA - principle components analysis
    - Factor analysis
    - SVD

Dimension Reduction with PCA

- PCA helps us find new axes (principal components) of our dimensions that can better capture the variance of the data.



If we were to list each principal component, PC1 would be the dimension that captures the highest proportion of the data variance, with PC2 being the dimension that captures the highest proportion of the **remaining** variance that PC1 could not capture. Similarly, PC3 would be the dimension capturing the highest proportion of the remaining variance that PC1 and PC2 could not capture, etc.

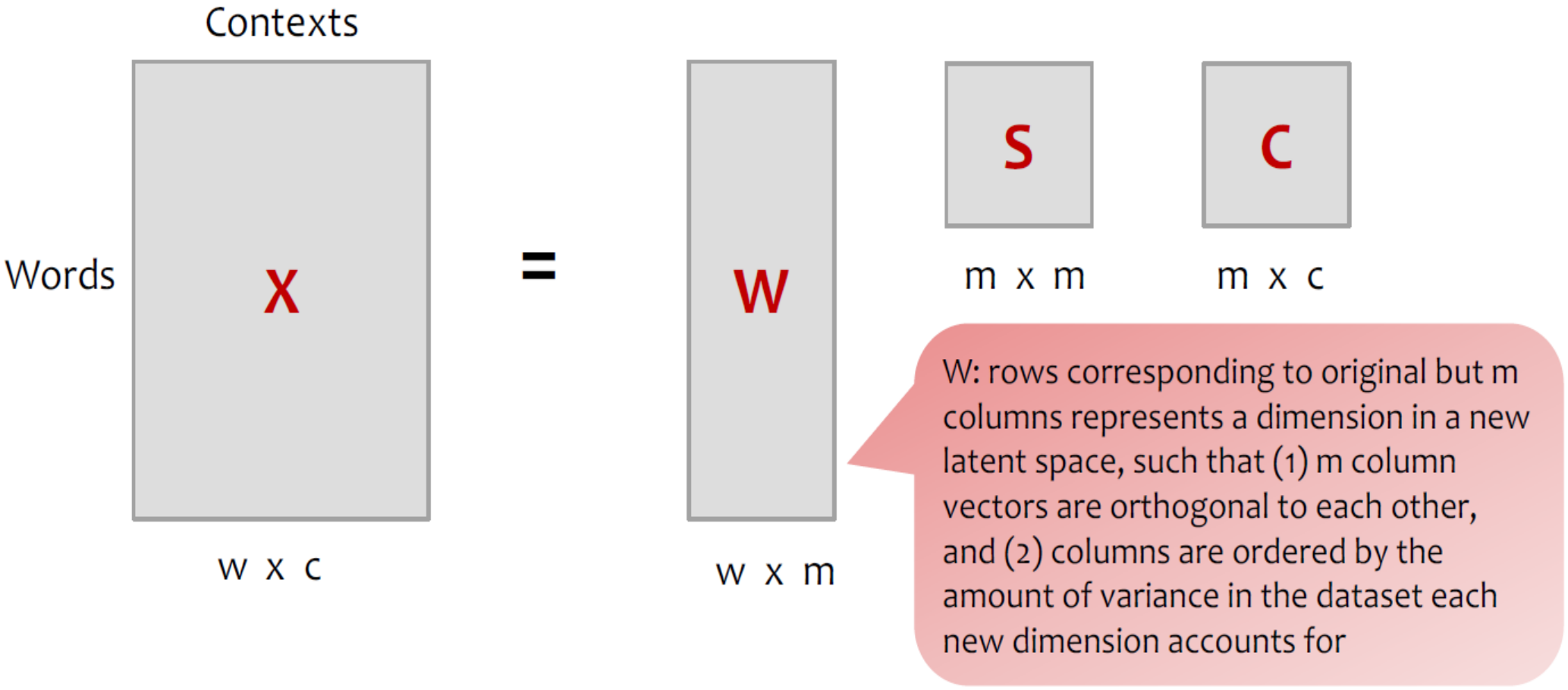
Singular Value Decomposition (SVD)

- Singular value decomposition(SVD) and principal component analysis(PCA) are two eigenvalue methods used to reduce a high-dimensional dataset into fewer dimensions while retaining important information.
- SVD is one of the most popular methods for dimensionality reduction and found its use in NLP originally via latent semantic analysis (LSA). SVD factories the word-context co-occurrence matrix into the product of three matrices:

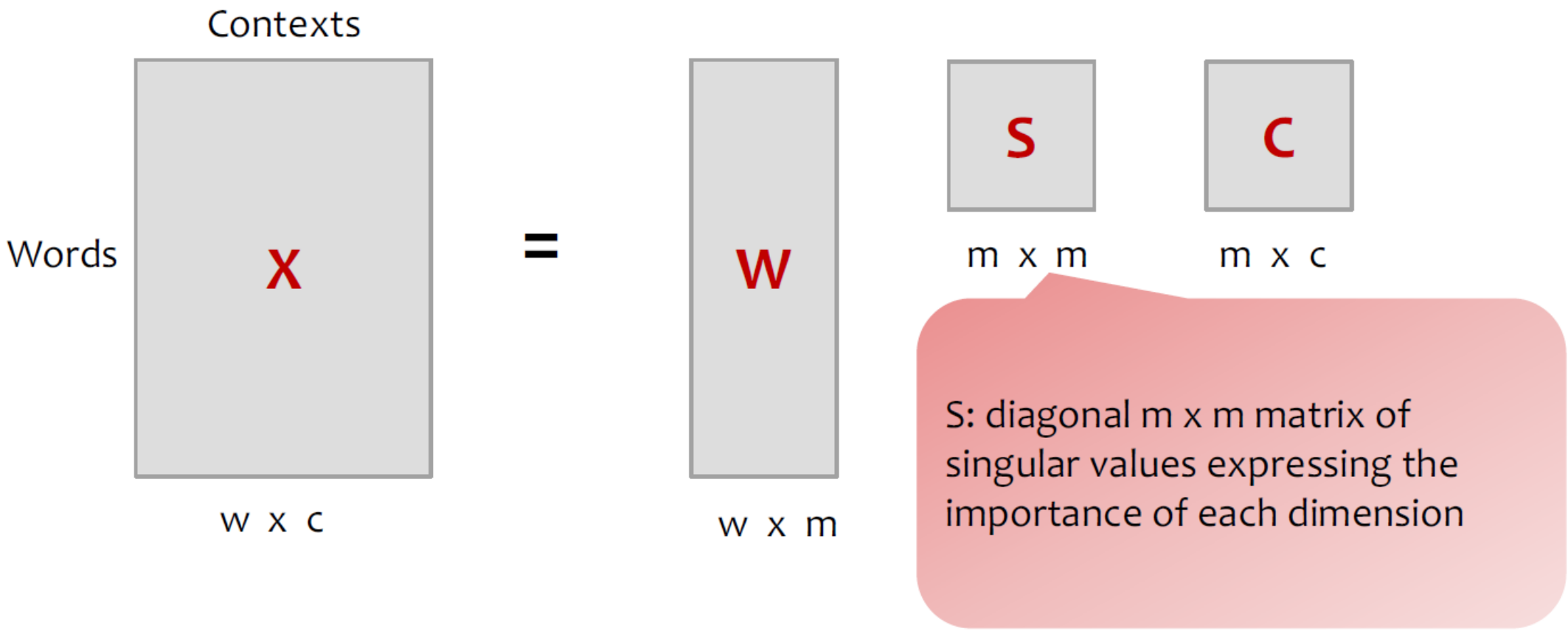
$$W \cdot S \times C^T$$

- where S and C are orthonormal matrices (i.e. square matrices whose rows and columns are orthogonal unit vectors) and S is a diagonal matrix of eigenvalues in decreasing order.

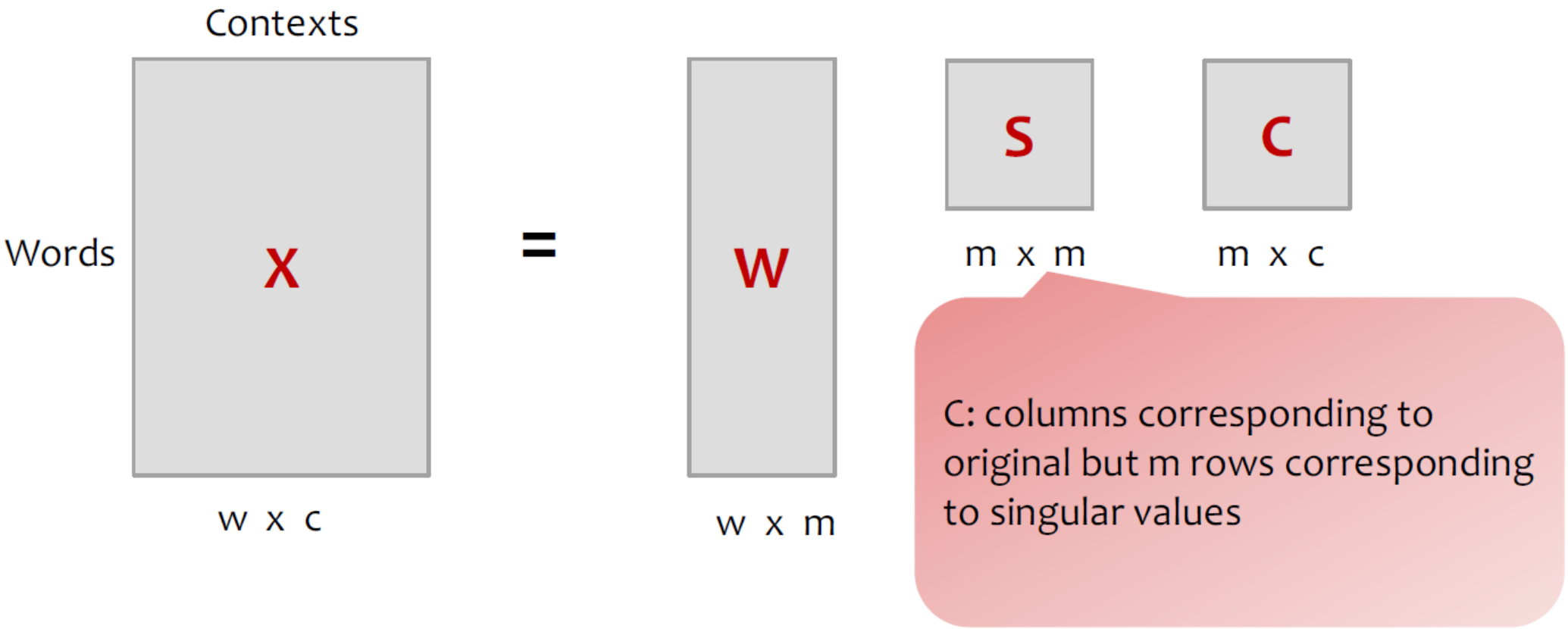
Singular Value Decomposition (SVD)



Singular Value Decomposition (SVD)



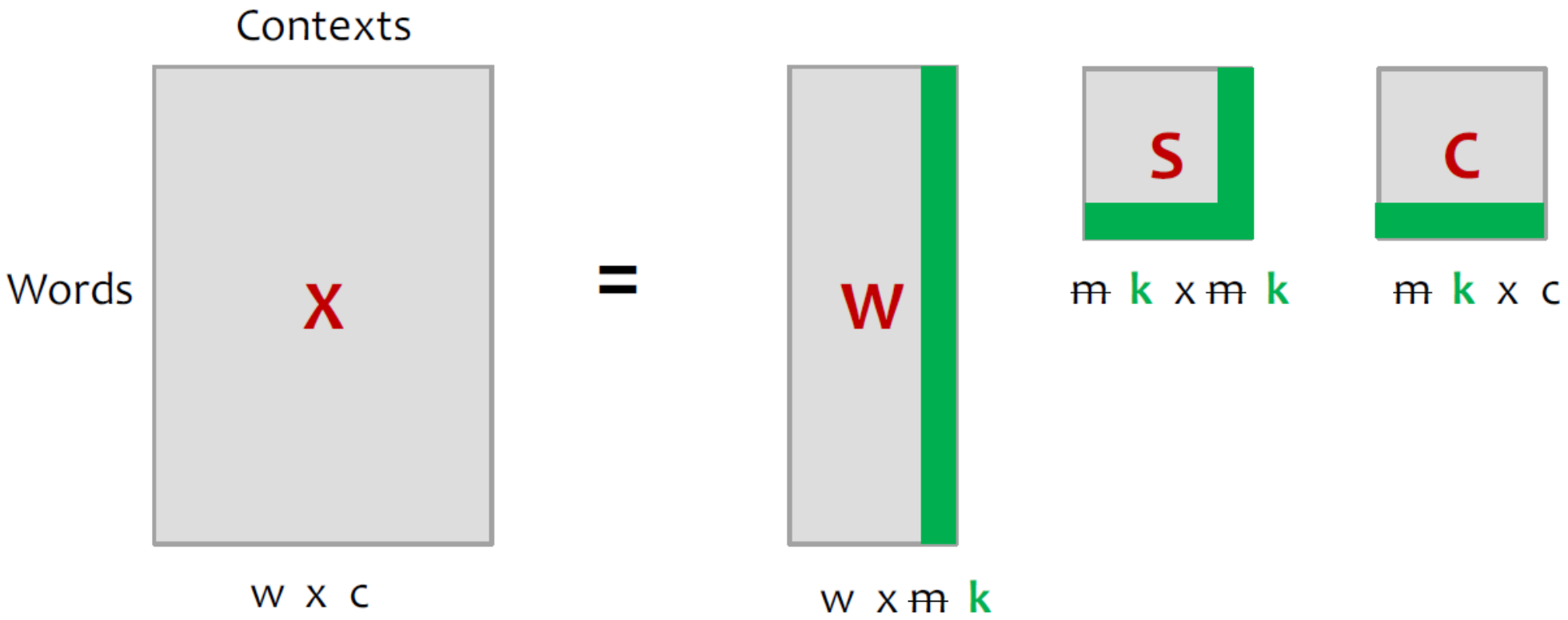
Singular Value Decomposition (SVD)



SVD Applied to Term-Document Matrix: Latent Semantic Analysis

- If instead of keeping all m dimensions, we just keep the top k singular values. Let's say 300.
- The result is a least-square approximation to the original X
- But instead of multiplying, we'll just make use of W

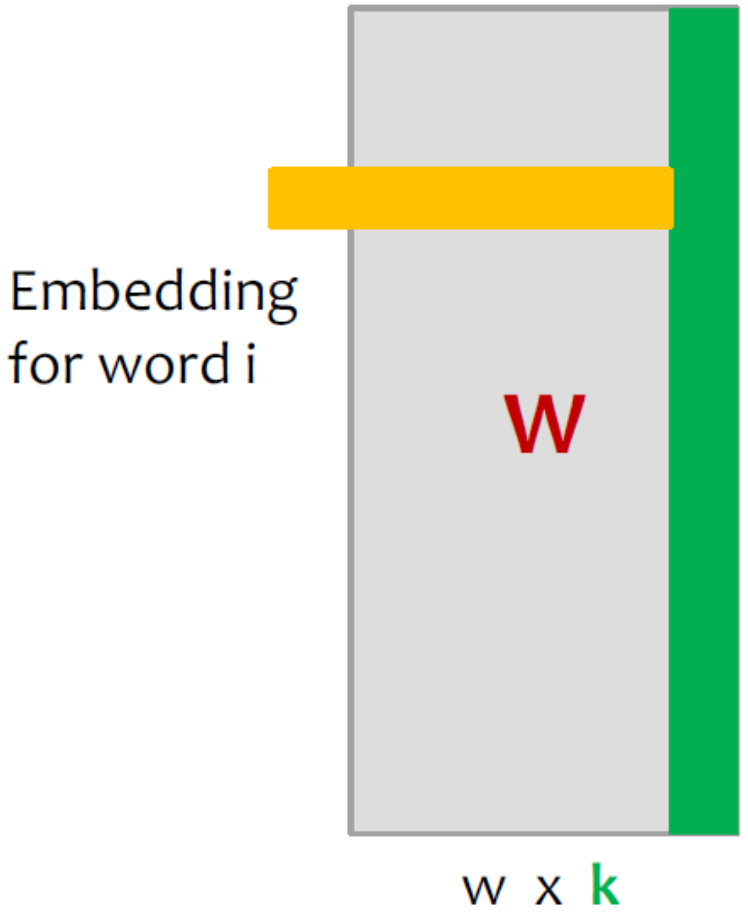
Truncated SVD



Truncated SVD Produces Embeddings

- Each row of W is a k-dimensional representation of each word w
- K might range from 50 to 100

- Generally we keep the top k dimensions, but some experiments suggest that getting rid of the top 1 dimension or even the top 50 dimensions is helpful



Dense SVD Embeddings versus Sparse Vectors

- Dense SVD embeddings sometimes work better than sparse PPMI matrices at tasks like word similarity
  - Denoising: low-order dimensions may represent unimportant information
  - Truncation may help the models generalize better to unseen data
  - Having a smaller number of dimensions may make it easier for classifiers to properly weight the dimensions for the task
  - Dense models may do better at capturing higher order cooccurrence

Problems with SVD method

1. It requires huge memory to store the co-occurrence matrix.
2. The dimensions of the matrix change very often (new words are added very frequently and corpus changes in size).
3. The matrix is extremely sparse since most words do not cooccur.
4. The matrix is very high dimensional in general (106 *106)
5. Quadratic cost to train (i.e. to perform SVD)
6. Requires the incorporation of some hacks on X to account for the drastic imbalance in word frequency

Statistical, Count-based Methods Summary

Method	Based On	Vector Type	Strengths	Weaknesses
BoW	Word counts	Sparse	Simple, fast	No semantics
TF	Normalized counts	Sparse	Document-length normalization	Still lexical only
TF-IDF	Counts + rarity	Sparse	Highlights important words	Context ignored
Co-occurrence Matrix	Word-word counts	Very sparse	Foundation for PMI/GloVe	Huge matrices
PMI	Co-occurrence probabilities	Dense or sparse	Captures association	No context handling
SVD (LSA)	Matrix factorization	Dense	Early semantic structure	Loses nuance

These methods formed the foundation for later **neural** and **contextual** embeddings like Word2Vec → GloVe → BERT → GPT.

Static Distributed / Prediction-Based Word Representations

- These are **dense vector representations** of words learned by training a neural network to **predict context**.
- They are called **static** because **each word has one fixed vector**, regardless of context.
- They are also called **distributed embeddings** because each linguistic feature is **distributed across many dimensions** of the vector.
- They are also known as **prediction-based embeddings**, in contrast to count-based embeddings like TF-IDF or co-occurrence matrices.

Main Methods:

- **Word2Vec** (CBOW, Skip-gram) https://code.google.com/archive/p/word2vec/
- **GloVe** (Global Vectors) http://nlp.stanford.edu/projects/glove/
- **FastText** (subword-enhanced) http://www.fasttext.cc/

Characteristics:

- Capture semantic similarity
- Dense vectors (50–300 dims)

Word2Vec (CBOW & Skip-Gram)

Developed by Mikolov et al. (2013), Word2Vec introduced **neural prediction-based embeddings**.

CBOW (Continuous Bag of Words)

- Task: **Predict the target word** from surrounding context
- Example: Predict “cat” from [“the”, “sat”, “on”, “the”, “mat”]

Skip-Gram

- Task: **Predict context words** from a target word
- Example: Given “cat”, predict nearby words
- Works better for rare words

Characteristics

- Produces **dense vectors** (e.g. 100–300 dims)
- Captures semantic relations:
  - king – man + woman ≈ queen

GloVe (Global Vectors)

While it uses global co-occurrence statistics, **its training objective is prediction-based** (factorizing a weighted log co-occurrence prediction error).

It learns embeddings by predicting the **ratio** of word–word co-occurrence probabilities.

Characteristics:

- Combines global matrix factorization with local prediction
- Produces static embeddings similar to Word2Vec
- Often more stable on large corpora

FastText (Subword-Enhanced Word Embeddings)

Developed by Facebook AI.

Key idea:

- A word is represented as a **bag of character n-grams**
- Example: “playing” → {“pla”, “lay”, “ayi”, “yin”, “ing”, ...}
- Embedding = sum of n-gram embeddings

Advantages

- Handles rare words
- Handles out-of-vocabulary words
- Excellent for morphologically rich languages (Turkish, Finnish, Arabic)

But still **static**:

- “bank” has the same vector in all contexts

Word2vec embeddings

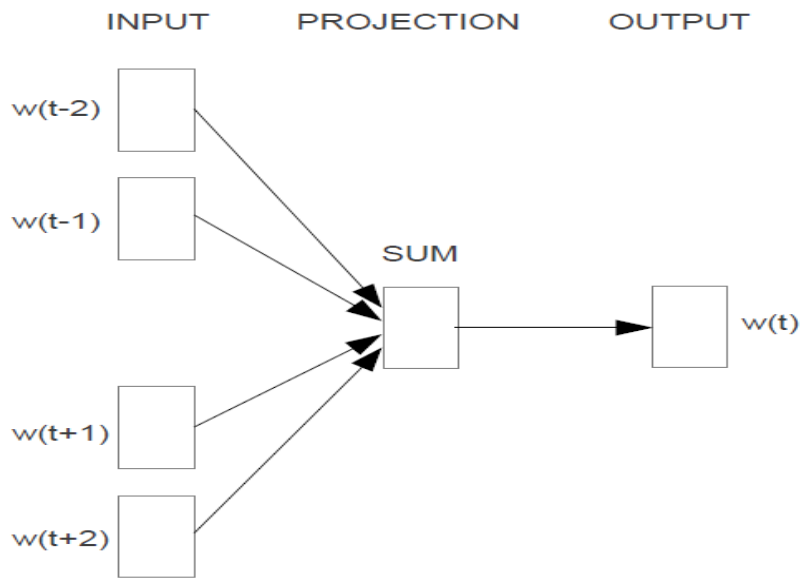
- Word2vec
  - **Dense** vectors
  - Representation is created by training a classifier to **predict** whether a word is likely to appear nearby and far away words.
  - Unlike count-based methods (BoW, TF-IDF), these embeddings are learned by **optimizing prediction tasks.**
  - Given dataset, training teaches the model to place similar words near each other in vector space.

Word2vec

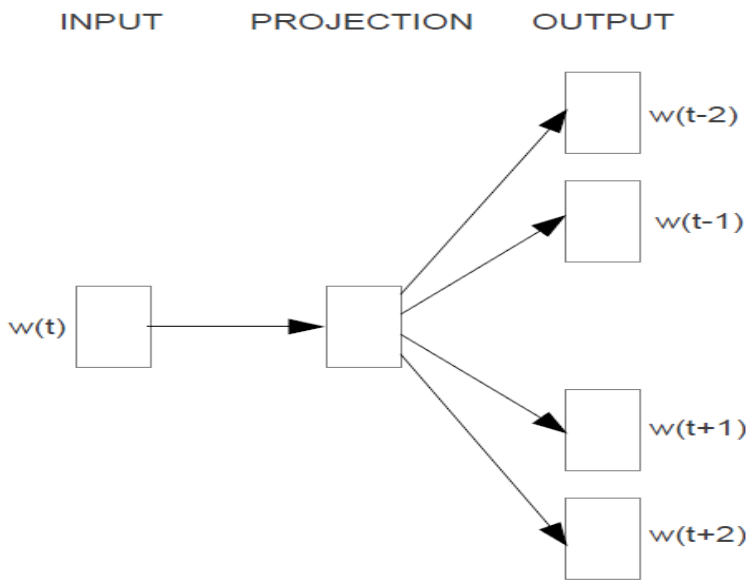
- The intuition: Instead of **counting** how often each word *w* occurs near "*apricot*"
  - Train a classifier on a binary **prediction** task:
    - Is *w* likely to show up near "*apricot*"?
- We don't actually care about this task
    - But we'll take the learned classifier weights as the word embeddings
- 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
    - The idea comes from **neural language modeling**
      - Bengio et al. (2003); Collobert et al. (2011)

Word2vec Models

There are two types of architectures(models/algorithms/approach) to learn the underlying word representations for each word by using neural networks.



CBOW



Skip-gram

Graphical representation of the CBOW model and Skip-gram model.

- In the CBOW model, the distributed representations of context (or surrounding words) are combined to predict the word in the middle.
- In the Skip-gram model, the distributed representation of the input word is used to predict the context.

Skip-Gram 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

Skip-Gram Training Data

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

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

c1 c2 [target] c3 c4

Skip-Gram Classifier

- (assuming a +/- 2 word window)

…lemon, a [tablespoon of apricot jam, a] pinch…
c1 c2 [target] c3 c4

- Our goal is to train a classifier such that,
  - Given a **tuple (t,c)** of **a target word t paired with a context word c**
    - (apricot, jam)
    - (apricot, aardvark)
- It will return the **probability that c is a real context word** (**true** for jam, **false** for aardvark):

P(+ | t,c) : Probability that word c is a real context word for t
P(− | t,c) = 1−P(+|t,c) : Probability that word c is **not** a real context word for t

Similarity is computed from dot product

The **intuition of the skipgram model** is to base this probability on similarity:

- ***A word is likely to occur near the target if its embedding is similar to the target embedding***.
- ***Two vectors are similar if they have a high dot product***.
  - **Similarity(t,c) $\propto$ t · c**
  - The dot product **t.c** is not a probability, it's just a number ranging from 0 to ∞.

Turning dot products into probabilities

- Sim(w,c) ≈ w · *c is just a number*
- To turn this number into a probability ,We'll use the sigmoid from logistic regression.
- The probability that word c is a real context word for target word t is:

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

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

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

How Skip-Gram Classifier computes *P*(+|*w*, *c*)

- This is for one context word, but we have lots of context words.

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

- We need to take account of the multiple context words in the window.
- Skip-gram makes the strong but very useful simplifying assumption that all context words are independent, allowing us to just multiply their probabilities:

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

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

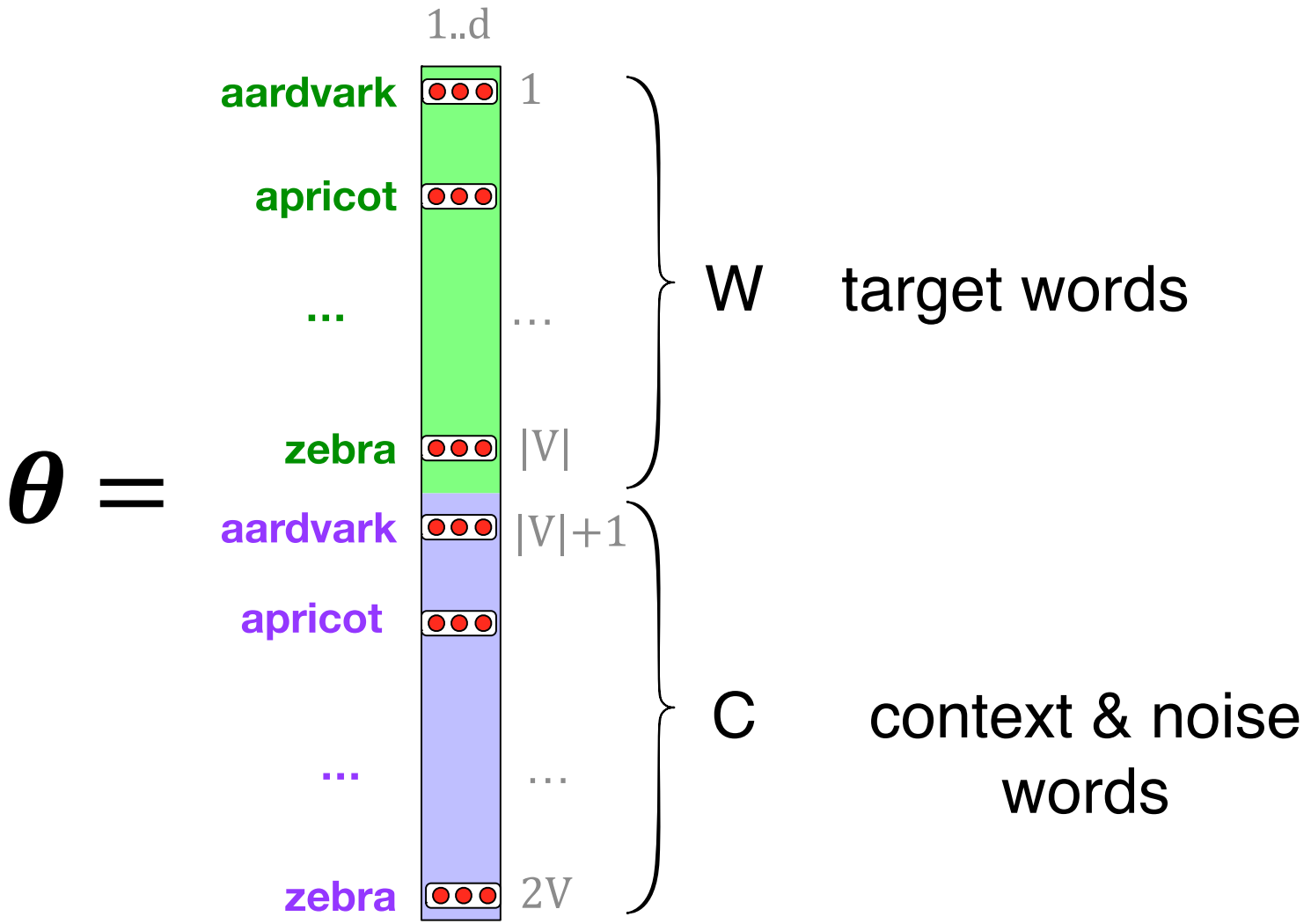
In summary, skip-gram trains a probabilistic classifier that, given a test target word w and its context window of L words $c_{1:L}$, assigns a probability based on how similar this context window is to the target word.

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



- The algorithm stores two embeddings for each word, the target embedding (sometimes called the input embedding) and the context embedding (sometimes called the output embedding).
- The parameter Ø that the algorithm learns is thus a matrix of 2|V| vectors, each of dimension d, formed by concatenating two matrices, the target embeddings W and the context+noise embeddings C.

Learning the embeddings

- The learning algorithm for skip-gram embeddings takes as input a corpus of text, and a chosen vocabulary size N.
- It begins by assigning a random embedding vector for each of the N vocabulary words, and then proceeds to iteratively shift the embedding of each word w to be more like the embeddings of words that occur nearby in texts, and less like the embeddings of words that don't occur nearby

Skip-Gram Training data

Training sentence:

…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

Training data:

- pairs centering on **apricot**
- Asssume a +/-2 word window is used.

Skip-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 by using noise words
- Noise word is any random word that isn't target word t (apricot)

Skip-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

Here's the setting where $k = 2$, so we'll have 2 negative examples in the negative training set − for each positive example w,c_{pos}. The noise words are chosen according to their weighted unigram frequency.

Example 2: Training Data

Training Example	Context Word	Target Word
#1	(i, natural)	like
#2	(like, language)	natural
#3	(natural, processing)	language
#4	(language)	processing

Source: https://thinkinfi.com/word2vec-skip-gram-explained/

One-hot Encoding

- **One-hot encoding**: We need to convert text into one-hot encoding as algorithm can only understand numeric values.

- For example encoded value of the word "i", which appears first in the vocabulary, will be as the vector [1, 0, 0, 0, 0].
- The word "like", which appears second in the vocabulary, will be encoded as the vector [0, 1, 0, 0, 0]

	i	like	natural	language	processing
i	1	0	0	0	0
like	0	1	0	0	0
natural	0	0	1	0	0
language	0	0	0	1	0
processing	0	0	0	0	1

Context-target words in one hot encoded form

You can see table below is our final training data, where encoded context word is Y variable for our model and encoded target word is X variable for our model as skipgram predicts surrounding word of a given word.

Training Example	Encoded Context Word	Encoded Target Word
#1	([1,0,0,0,0], [0,0,1,0,0])	[0,1,0,0,0]
#2	([0,1,0,0,0], [0,0,0,1,0])	[0,0,1,0,0]
#3	([0,0,1,0,0], [0,0,0,0,1])	[0,0,0,1,0]
#4	([0,0,0,1,0])	[0,0,0,0,1]

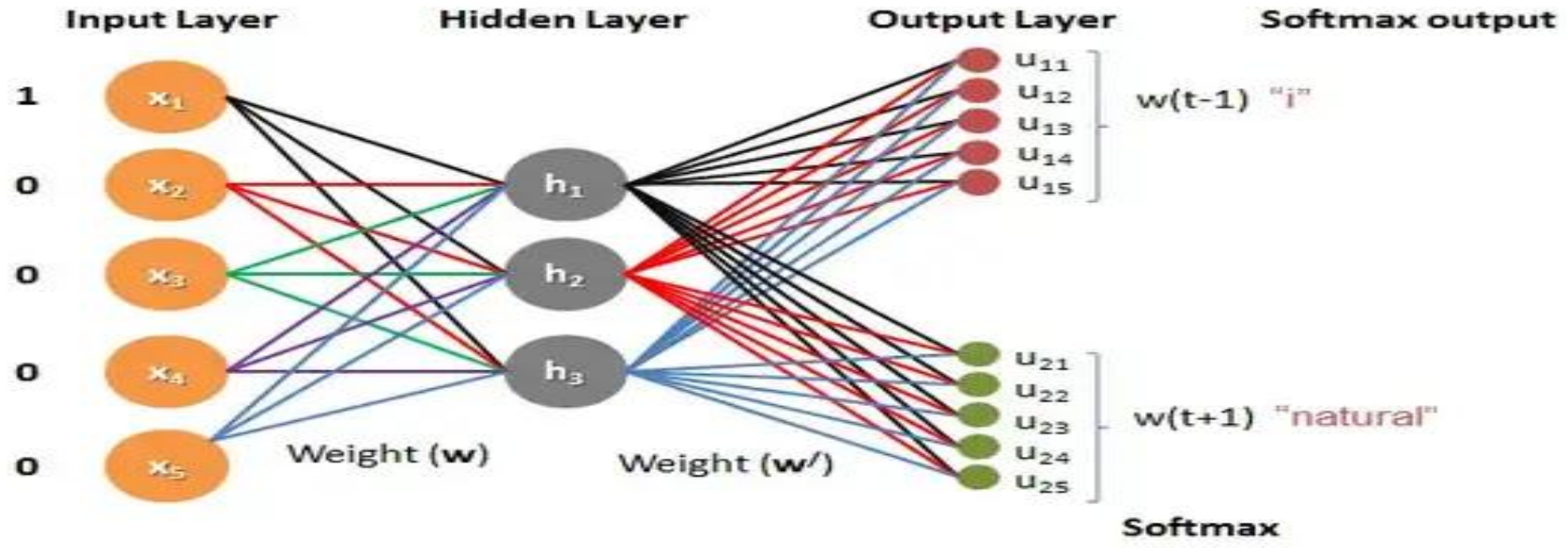
Training Example	Context Word	Target Word
#1	(i, natural)	like
#2	(like, language)	natural
#3	(natural, processing)	language
#4	(language)	processing

Skipgram Model

Skipgram model predicts surrounding word of a given word.

Training Example	Context Word	Target Word
#1	(i, natural)	like
#2	(like, language)	natural
#3	(natural, processing)	language
#4	(language)	processing

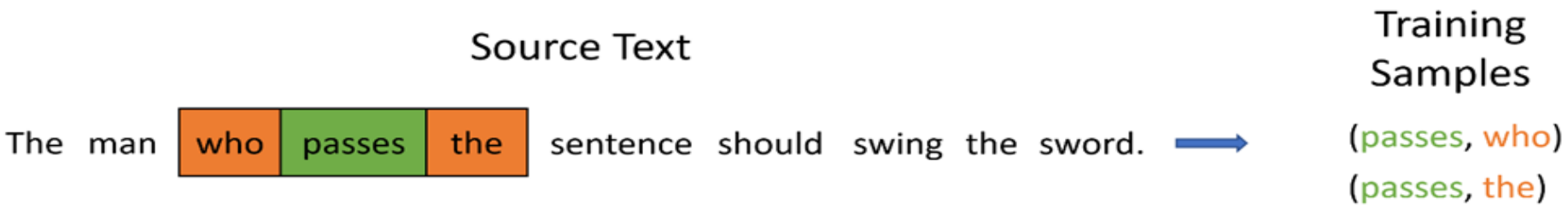
The input word is used to predict the context.



First training data point: The context words are "**i**" and "**natural**" and the target word is "**like**".

Example 3

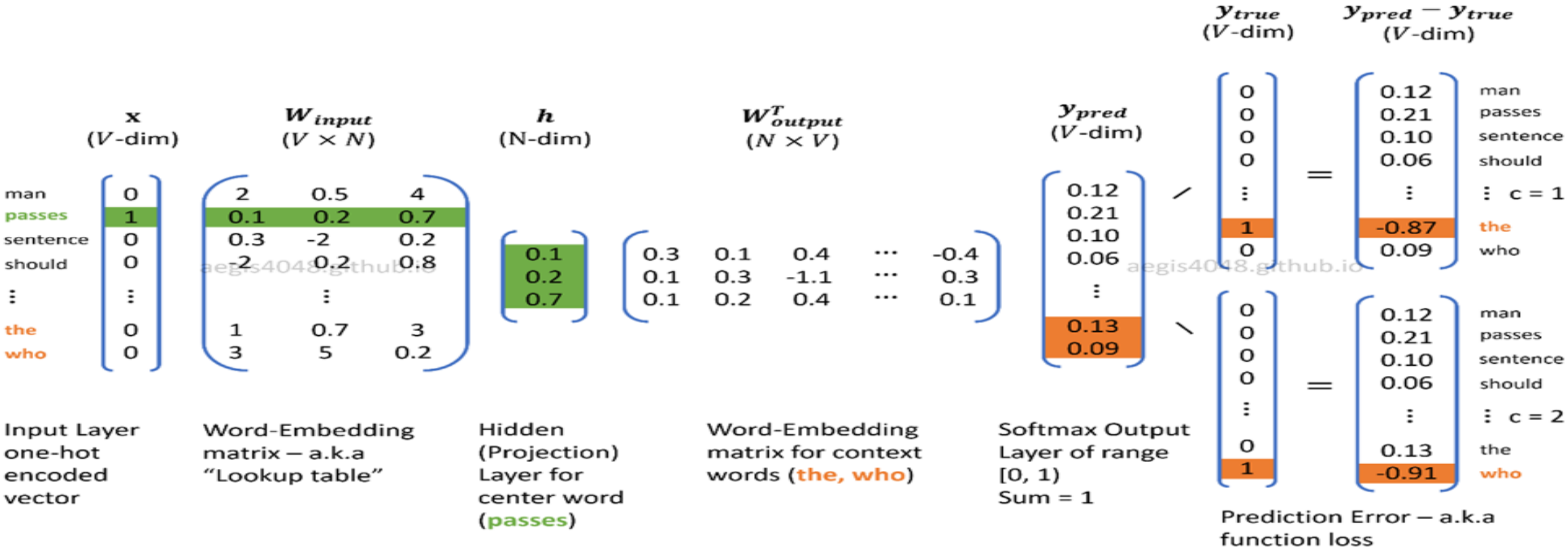
- We will use window=1 and assume that 'passes' is the current center word, making 'who' and 'the' context words. window is a hyper-parameter that can be empirically tuned. It typically has a range of [1,10].



- For illustration purpose, a three-dimensional neural net will be constructed.
  - In gensim, this can be implemented by setting size=3. This makes N=3.
  - Note that size is also a hyper-parameter that can be empirically tuned.
  - In real life, a typical Word2Vec model has 200-600 neurons.

Example 3

- The input weight matrix (Winput) will have a size of 8×3, and output weight matrix (WToutput) will have a size of 3×8. Recall that the corpus, *"The man who passes the sentence should swing the sword"*, has 8 unique vocabularies (V=8).



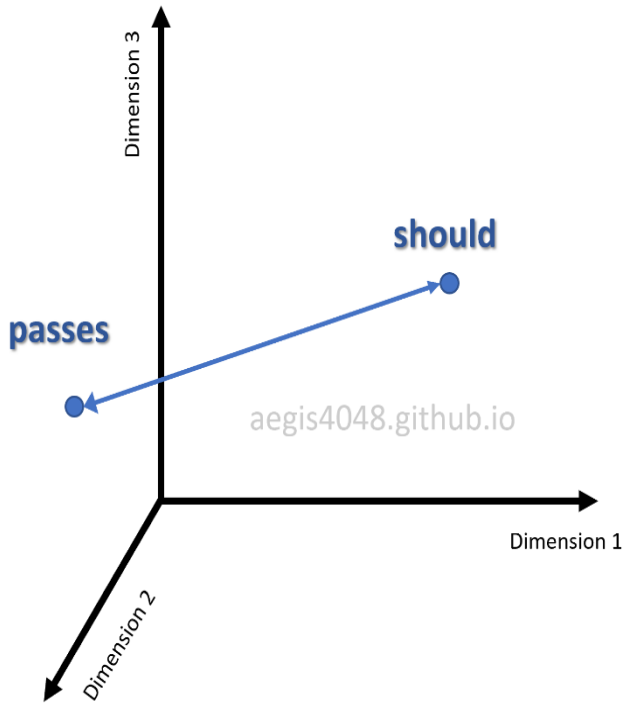
https://aegis4048.github.io/demystifying_neural_network_in_skip_gram_language_modeling

Example 3

$$W_{input} \quad (V \times N)$$

man	2	0.5	4
passes	0.1	0.2	0.7
sentence	0.3	-2	0.2
should	-2	0.2	0.8
⋮	⋮	⋮	
the	1	0.7	3
who	3	5	0.2

passes	0.1	0.2	0.7
should	-2	0.2	0.8



- The words of our interest are "passes" and "should". "passes" has a word vector of [0.1 0.2 0.7] and "should" has [−2 0.2 0.8]. Since we set the size of the weight matrix to be size=3 above, the matrix is three-dimensional, and can be visualized in a 3D vector space:

- Optimizing the embedding (weight) matrices θ results in representing words in a high quality vector space, and the model will be able to capture meaningful relationships among words.
- Related words are placed closer to each other on a vector space. Mathematically, this means that the vector distance between related words are smaller than the vector distance between unrelated words.

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} ...c_{negk}

- In fact skipgram with negative sampling (SGNS) uses more negative examples than positive examples (with the ratio between them set by a parameter k).

- 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.

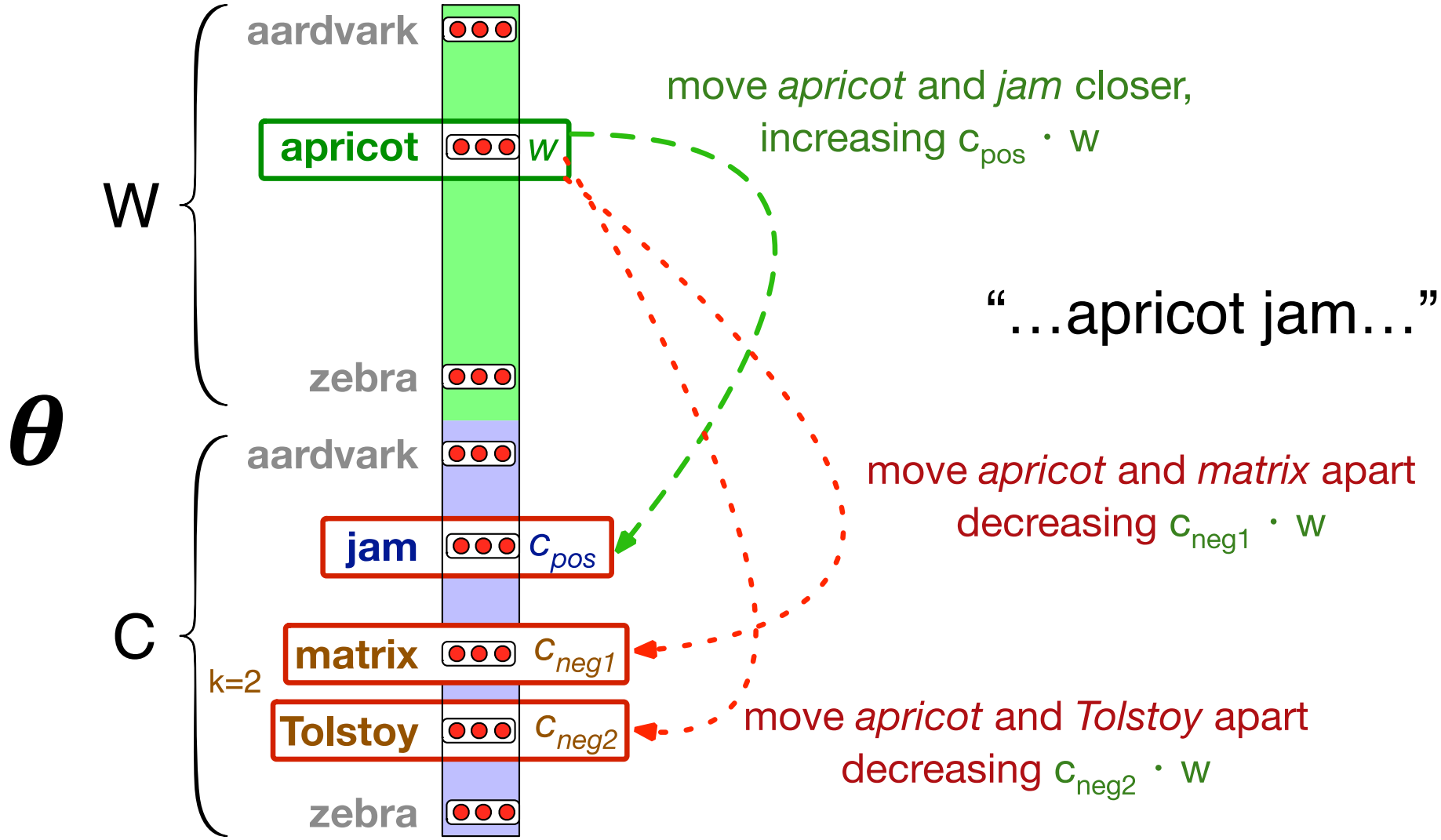
$$
\begin{aligned}
L_{CE} &= -\log\left[P(+|w,c_{pos})\prod_{i=1}^{k}P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos}) + \sum_{i=1}^{k}\log P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos}) + \sum_{i=1}^{k}\log\left(1-P(+|w,c_{neg_i})\right)\right] \\
&= -\left[\log \sigma(c_{pos}\cdot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

Learning the classifier

- How to learn?
  - Use Stochastic gradient descent!
- Let's represent words as vectors of some length (say 300), randomly initialized.
- So, we start with 300 random parameters.
- Over the entire training set, we'll adjust the word weights to
  - make the positive pairs more likely
  - and the negative pairs less likely
- Skip-gram model actually learns two separate embeddings for each word w:
  - target embedding t and
  - context embedding c.
- These embeddings are stored in two matrices,
  - target matrix W and
  - context matrix C (or W').

Intuition of one step of gradient descent

The skip-gram model tries to shift embeddings so the target embeddings (here for *apricot*) are closer to (have a higher dot product with) context embeddings for nearby words (here *jam*) and further from (lower dot product with) context embeddings for noise words that don't occur nearby (here *Tolstoy* and *matrix*).



Reminder: gradient descent

- Just as in logistic regression, the learning algorithm starts with randomly initialized W and C matrices, and then walks through the training corpus using gradient descent to update W and C so as to maximize the objective criteria.
- 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 $\frac{d}{dw}L(f(x;w),y)$ weighted by a **learning rate** η
  - Higher learning rate means move *w* faster

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

The derivatives of the loss function

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

To get the gradient, we need to take the derivative of Eq. given above with respect to the different embeddings. It turns out the derivatives are the following

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

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

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

Update equation in SGD

- Start with randomly initialized C and W matrices, then incrementally do updates
- The update equations going from time step t to t +1 in stochastic gradient descent are thus:

$$
\begin{aligned}
c_{pos}^{t+1} &= c_{pos}^{t} - \eta[\sigma(c_{pos}^{t} \cdot w^{t}) - 1]w^{t} \\
c_{neg}^{t+1} &= c_{neg}^{t} - \eta[\sigma(c_{neg}^{t} \cdot w^{t})]w^{t} \\
w^{t+1} &= w^{t} - \eta\left[[\sigma(c_{pos} \cdot w^{t}) - 1]c_{pos} + \sum_{i=1}^{k}[\sigma(c_{\mathrm{neg}_i} \cdot w^{t})]c_{neg_i}\right]
\end{aligned}
$$

Two sets of embeddings

- SGNS learns two sets of embeddings
  - Target embeddings matrix W
  - Context embedding matrix C (also named W')

- Once embeddings are learned, we'll have two embeddings for each word w_i: t_i and c_i.

  - We can choose to throw away the C matrix and just keep W, in which case each word i will be represented by the vector t_i.
  - Alternatively, we can add the two embeddings together, using the summed embedding t_i+c_i as the new d-dimensional embedding. It's common to just add them together.

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.

Evaluating Vector Models

- Compare to human scores on word similarity-type tasks:
  - WordSim-353 (Finkelstein et al., 2002)
  - SimLex-999 (Hill et al., 2015)
  - Stanford Contextual Word Similarity (SCWS) dataset (Huang et al., 2012)
  - TOEFL dataset

Semantics & Embeddings

Properties of Embeddings

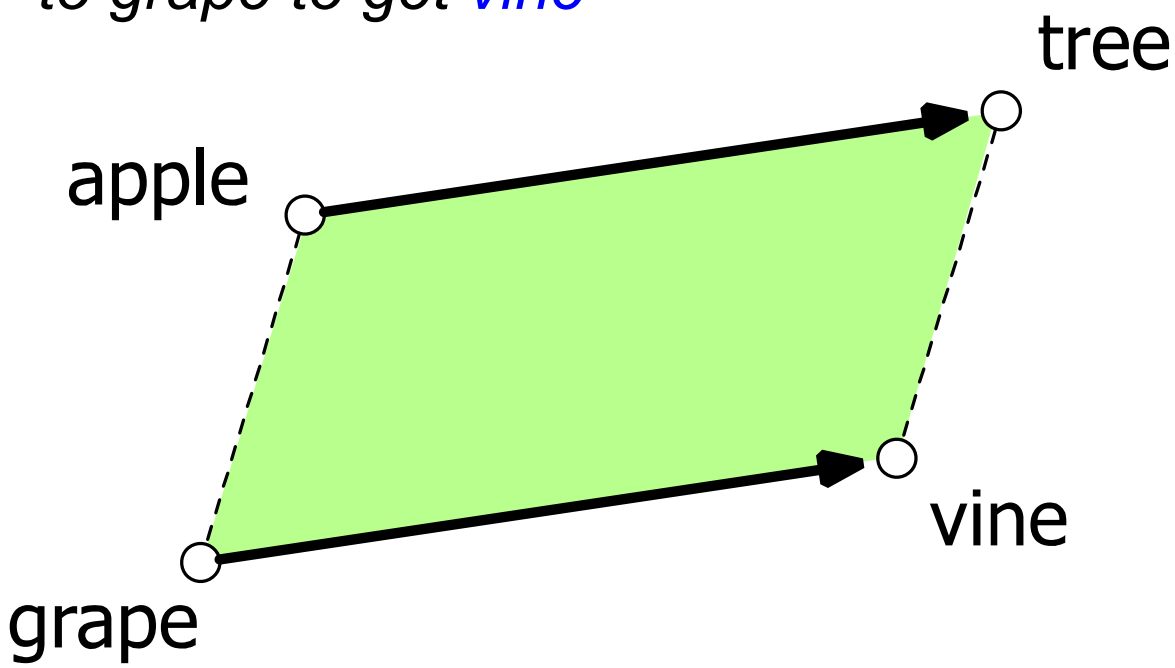
The kinds of neighbors depend on window size

- **Small windows** (C= +/- 2) : nearest words are syntactically similar words in same taxonomy
  - –*Hogwarts* (from the *Harry Potter* series) nearest neighbors are other fictional schools
    - *Sunnydale, Evernight, Blandings*

- **Large windows** (C= +/- 5) : nearest words are related words in same semantic field
  - –*Hogwarts* nearest neighbors are Harry Potter world:
    - *Dumbledore, half-blood, Malfoy*

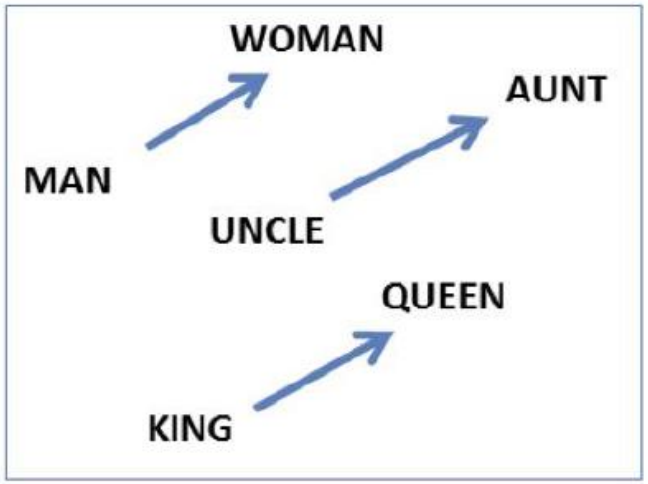
Analogical relations

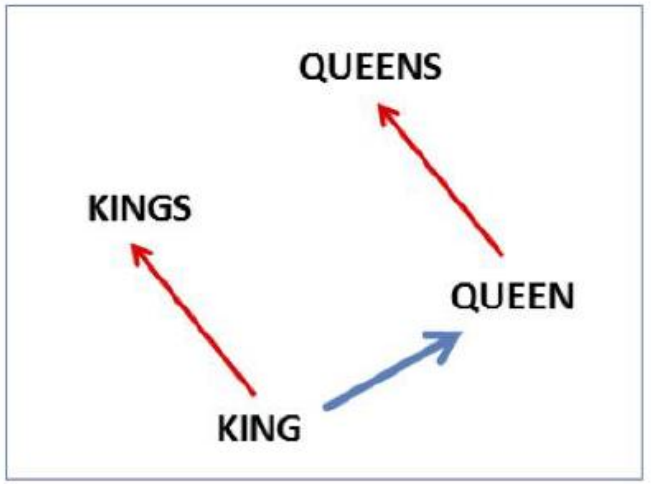
- Another semantic property of embeddings is their ability to capture relational meanings. In an important early vector space model of cognition, Rumelhart and Abrahamson (1973) proposed the parallelogram model for solving simple analogy problems of the form a is to b as a* is to what?
- To solve: *"apple is to tree as grape is to _____"*
- *Add* $\overrightarrow{tree}$ – $\overrightarrow{apple}$ *to* $\overrightarrow{grape}$ *to get* $\overrightarrow{vine}$



Analogical relations

- vector(‘*king*’) - vector(‘*man*’) + vector(‘*woman*’) ≈ vector(‘queen’)
- vector(‘*Paris*’) - vector(‘*France*’) + vector(‘*Italy*’) ≈ vector(‘Rome’)





Analogical relations via parallelogram

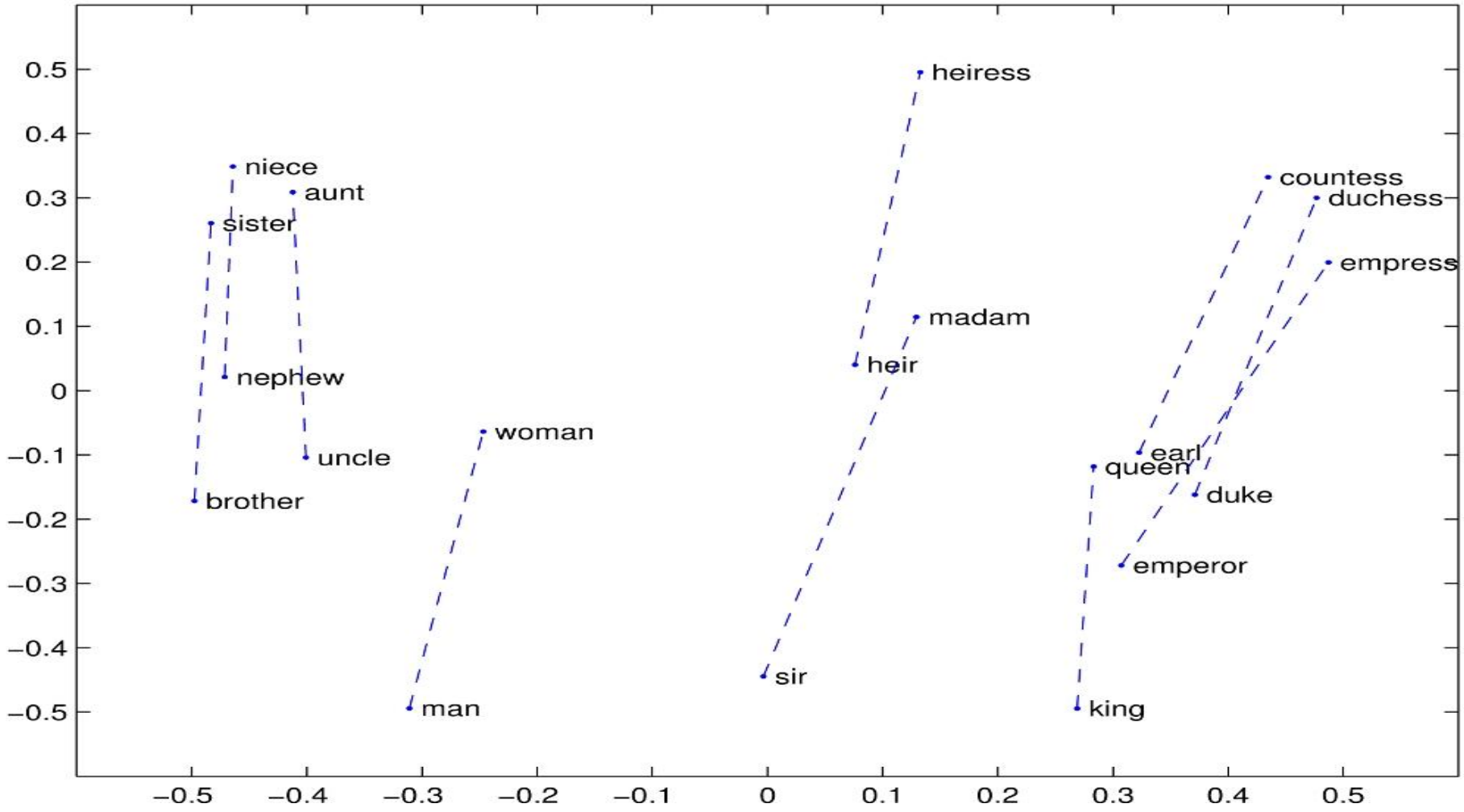
- The parallelogram method can solve analogies with both sparse and dense embeddings (Turney and Littman 2005, Mikolov et al. 2013b)

- $\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ 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:

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

Structure in GloVE Embedding space



Relational properties of the GloVe vector space, shown by projecting vectors onto two dimensions. (king - man + woman is close to queen.)

Caveats with the parallelogram method

- It only seems to work for frequent words, small distances and certain relations (relating countries to capitals, or parts of speech), but not others. (Linzen 2016, Gladkova et al. 2016, Ethayarajh et al. 2019a)

- Understanding analogy is an open area of research (Peterson et al. 2020)

Embeddings as a window onto historical semantics

- Embeddings can also be a useful tool for studying how meaning changes over time, by computing multiple embedding spaces, each from texts written in a particular time period.
- Train embeddings on different decades of historical text to see meanings shift

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

A t-SNE visualization of the semantic change of 3 words in English using word2vec vectors.

William L. Hamilton, Jure Leskovec, and Dan Jurafsky. 2016. Diachronic Word Embeddings Reveal Statistical Laws of Semantic Change. Proceedings of ACL.

Embeddings reflect cultural bias!

Bolukbasi, Tolga, Kai-Wei Chang, James Y. Zou, Venkatesh Saligrama, and Adam T. Kalai. "Man is to computer programmer as woman is to homemaker? debiasing word embeddings." In *NeurIPS*, pp. 4349-4357. 2016.

- Ask "Paris : France :: Tokyo : x"
  - x = Japan
- Ask "father : doctor :: mother : x"
  - x = nurse
- Ask "man : computer programmer :: woman : x"
  - x = homemaker

Algorithms that use embeddings as part of e.g., hiring searches for programmers, might lead to bias in hiring

Historical embedding as a tool to study cultural biases

Garg, N., Schiebinger, L., Jurafsky, D., and Zou, J. (2018). Word embeddings quantify 100 years of gender and ethnic stereotypes. Proceedings of the National Academy of Sciences 115(16), E3635–E3644.

- Compute a **gender or ethnic bias** for each adjective: e.g., how much closer the adjective is to "woman" synonyms than "man" synonyms, or names of particular ethnicities
  - Embeddings for **competence** adjective (*smart, wise, brilliant, resourceful, thoughtful, logical)* are biased toward men, a bias slowly decreasing 1960-1990
  - Embeddings for **dehumanizing** adjectives (barbaric, monstrous, bizarre) were biased toward Asians in the 1930s, bias decreasing over the 20th century.
- These match the results of old surveys done in the 1930s

Summary

- In vector semantics, a word is modeled as a vector—a point in high-dimensional space, also called an **embedding**. In this chapter we focus on **static embeddings**, where each word is mapped to a fixed embedding.
- Vector semantic models fall into two classes: **sparse** and **dense**. In sparse models each dimension corresponds to a word in the vocabulary V and cells are functions of **co-occurrence counts**. The **term-document** matrix has a row for each word (**term**) in the vocabulary and a column for each document. The **word-context** or **term-term** matrix has a row for each (target) word in the vocabulary and a column for each context term in the vocabulary. Two sparse weightings are common: the **tf-idf** weighting which weights each cell by its **term frequency** and **inverse document frequency**, and **PPMI** (point-wise positive mutual information), which is most common for word-context matrices.

Summary

- Dense vector models have dimensionality 50–1000. **Word2vec** algorithms like **skip-gram** are a popular way to compute dense embeddings. Skip-gram trains a logistic regression classifier to compute the probability that two words are 'likely to occur nearby in text'. This probability is computed from the dot product between the embeddings for the two words.
- Skip-gram uses stochastic gradient descent to train the classifier, by learning embeddings that have a high dot product with embeddings of words that occur nearby and a low dot product with noise words.
- Other important embedding algorithms include **GloVe**, a method based on ratios of word co-occurrence probabilities.
- Whether using sparse or dense vectors, word and document similarities are computed by some function of the **dot product** between vectors. The cosine of two vectors—a normalized dot product—is the most popular such metric.