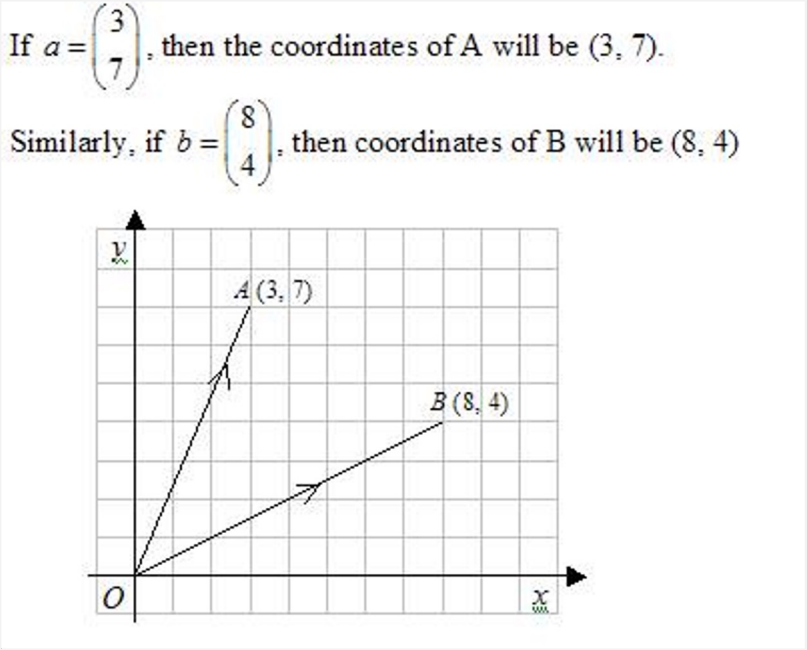
# Intro to NLP
# Vector Spaces and Word Embeddings

# Overview

1. Vectors and Vector Spaces
2. Document-level vectors
3. Co-occurrence vectors
4. Word2Vec - Word Embeddings
5. Translation

# What is a vector?




Source: https://www.onlinemathlearning.com/position-vector.html

There are many points of view

- Mathematical
- Physical
- Computational

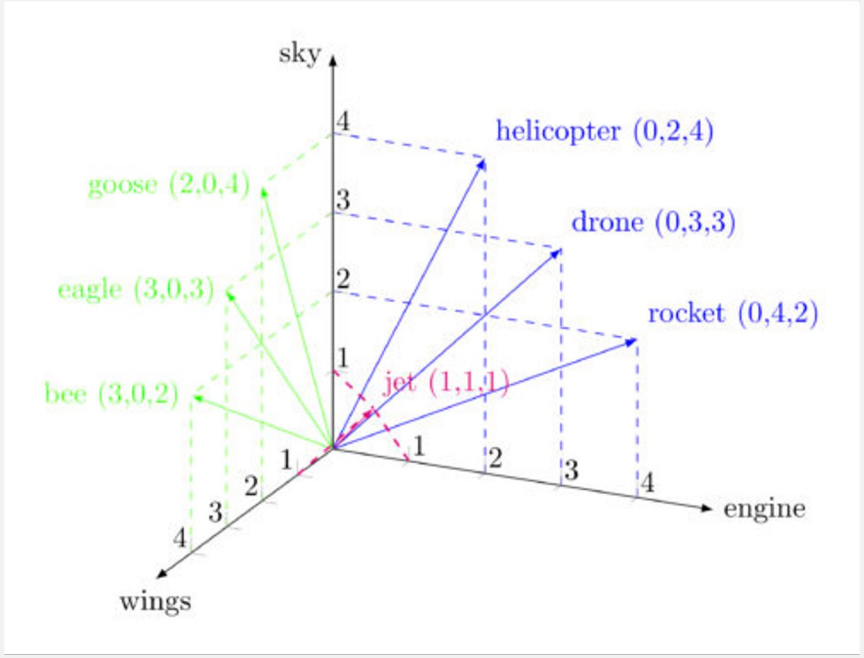
- An ordered series of values
- Multidimensional
  - We'll work with high dimensionality
- Computers deal very well with them
- Each dimension is independent of each other and will mean a different quantity
- The vector itself will represent something

# Vector Spaces

- What a vector represents is tied to it's vector space
  - "The first dimension means [...], the second dimension [...]"
  - These definitions can be as varied as we like

- This can be used as a way of making computers understand text better
  - We could build a vector space where each dimension means something useful

- Spatial proximity -> similar words




Source: https://corpling.hypotheses.org/495

# Vector Spaces

- What can a vector represent?
  - Words, Documents, Corpus, etc

- For example, we could have a vocabulary of *V* known words
  - We could represent a sentence/document
    - Assign a dimension to each word
    - Count the occurrences of each word

| | about | bird | heard | is | the | word | you |
|---|---|---|---|---|---|---|---|
| About the bird, the bird, bird bird bird | 1 | 5 | 0 | 0 | 2 | 0 | 0 |
| You heard about the bird | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| The bird is the word | 0 | 1 | 0 | 1 | 2 | 1 | 0 |

Source: https://vitalflux.com/text-classification-bag-of-words-model-python-sklearn/

# Vector Spaces

- What can a vector represent?
  - Words, Documents, Corpus, etc

- For example, we could have a vocabulary of *V* known words
  - We could represent a sentence/document
    - Assign a dimension to each word
    - Count the occurrences of each word

| | about | bird | heard | is | the | word | you |
|---|---|---|---|---|---|---|---|
| About the bird, the bird, bird bird bird | 1 | 5 | 0 | 0 | 2 | 0 | 0 |
| You heard about the bird | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| The bird is the word | 0 | 1 | 0 | 1 | 2 | 1 | 0 |

**Bag of words!**

Source: https://vitalflux.com/text-classification-bag-of-words-model-python-sklearn/

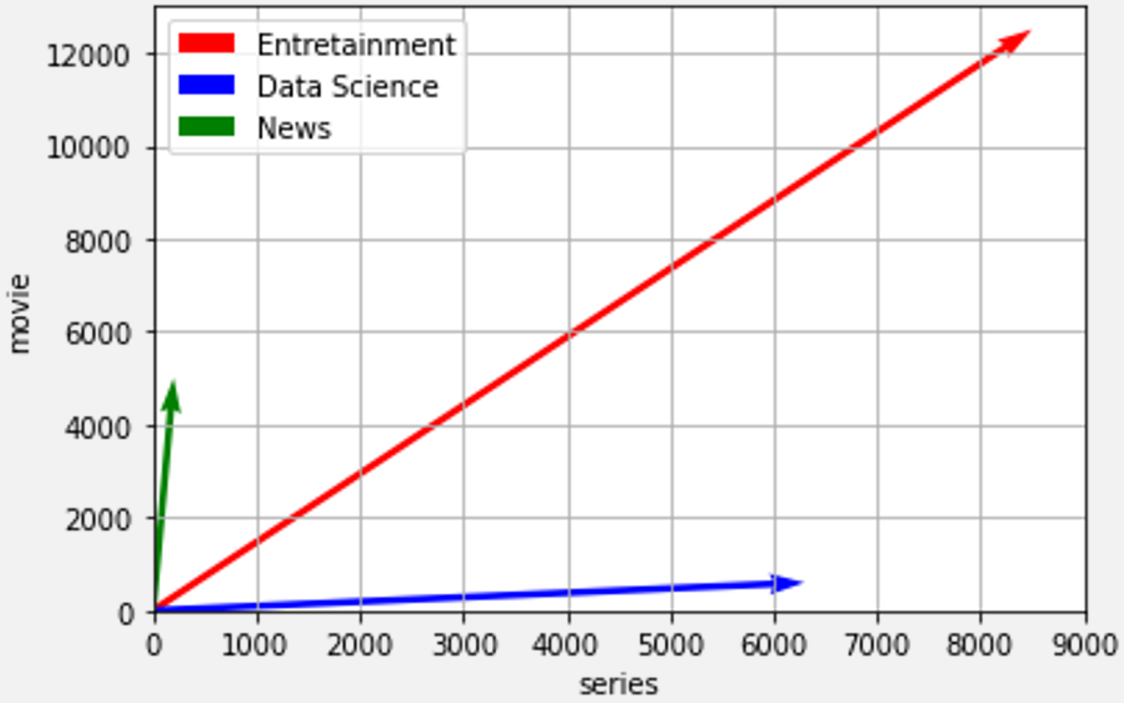
# Vector Spaces: Representing documents

- We can engineer categories of interest to be our dimensions
- Then use the number of times the word occurs in each document category as the values for the dimensions
- Vectors can be created by simply “reading” the corpus and knowing to which category of interest it belongs to

| | Entertainment | Data Science | News |
|---|---|---|---|
| series | 8500 | 6300 | 200 |
| movie | 12500 | 600 | 5000 |

# Vector Spaces: Representing documents

- Use a vector to represent a single document
- Number of times a word occurs in each document category



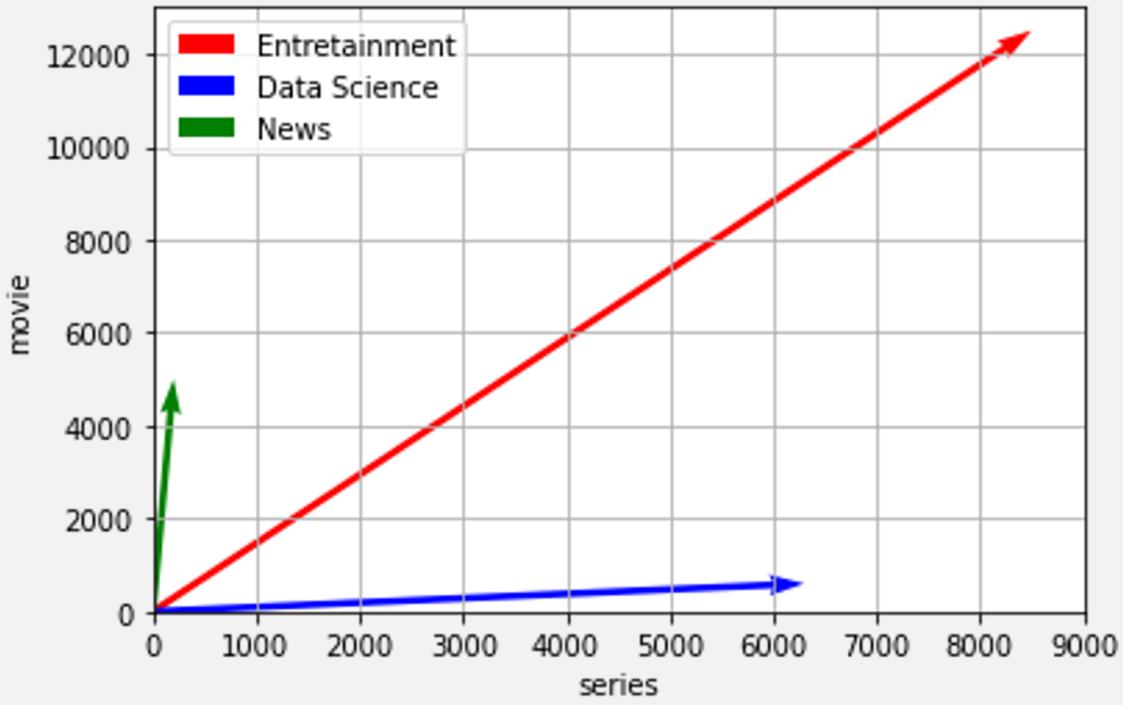

| | Entertainment | Data Science | News |
|---|---|---|---|
| series | 8500 | 6300 | 200 |
| movie | 12500 | 600 | 5000 |

For N dimensions and enough data, we start to cluster these categories in the Vector Space and it can then be used alongside other traditional ML algorithms/models (e.g.: KNN)

# Euclidean Distance

- $d(B,A) = \sqrt{(B_1 - A_1)^2 + (B_2 - A_2)^2}$

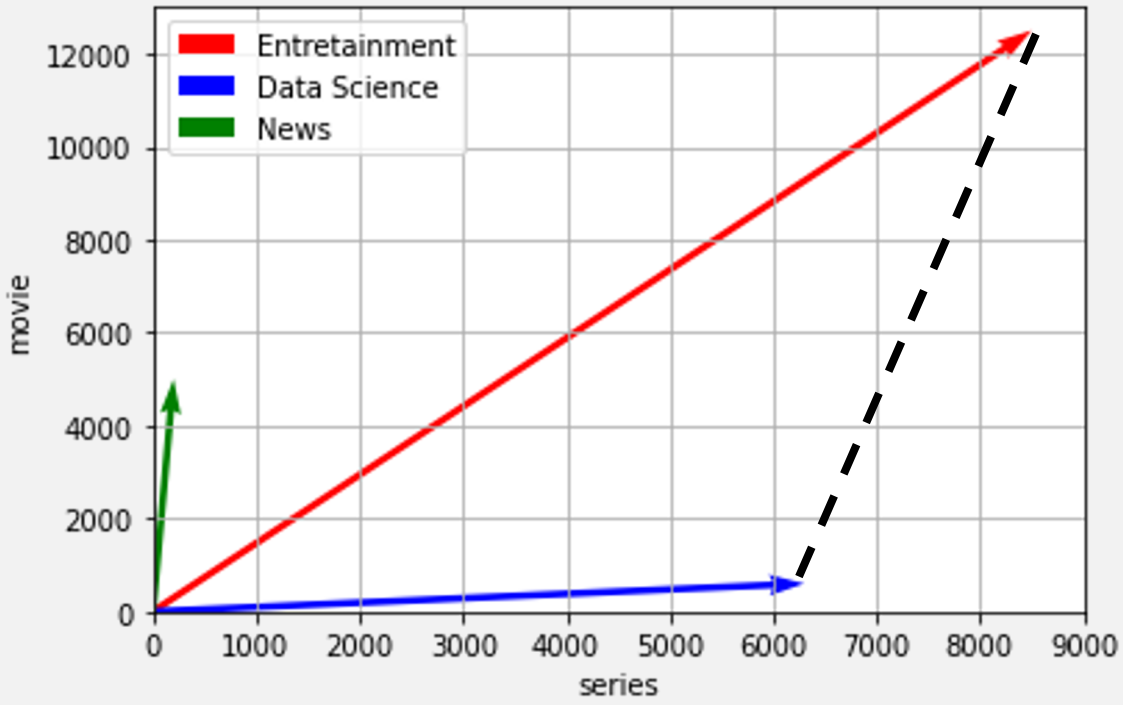
| | Entertainment | Data Science |
|---|---|---|
| series | 8500 | 6300 |
| movie | 12500 | 600 |



# Euclidean Distance

- $d(B,A) = \sqrt{(B_1 - A_1)^2 + (B_2 - A_2)^2}$

| | Entertainment | Data Science |
|---|---|---|
| series | 8500 | 6300 |
| movie | 12500 | 600 |

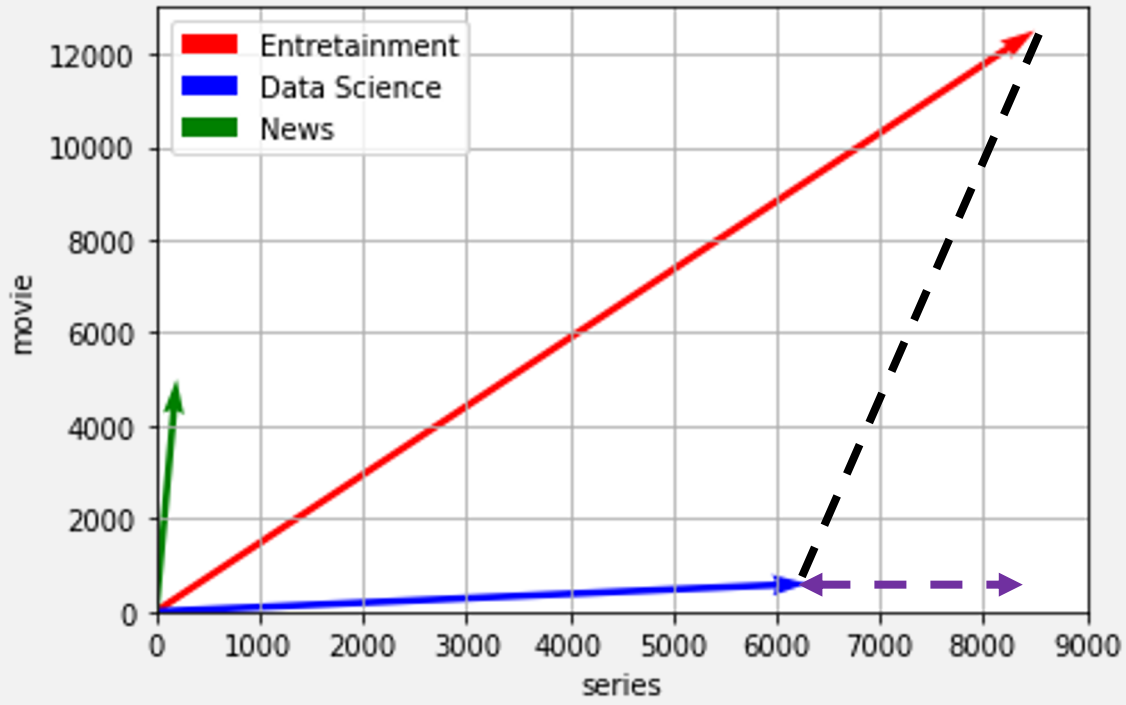


# Euclidean Distance

- $d(B,A) = \sqrt{(B_1 - A_1)^2 + (B_2 - A_2)^2}$

- $B_1 - A_1$

| | Entertainment | Data Science |
|---|---|---|
| series | 8500 | 6300 |
| movie | 12500 | 600 |

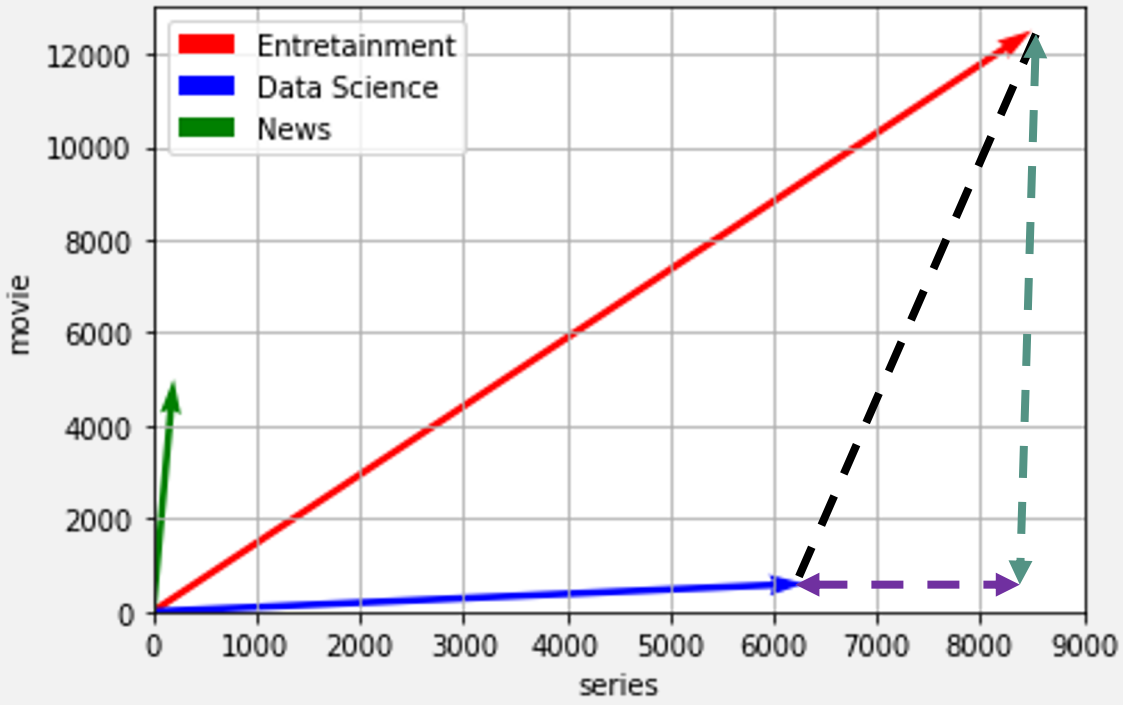


# Euclidean Distance

- $d(B,A) = \sqrt{(B_1 - A_1)^2 + (B_2 - A_2)^2}$

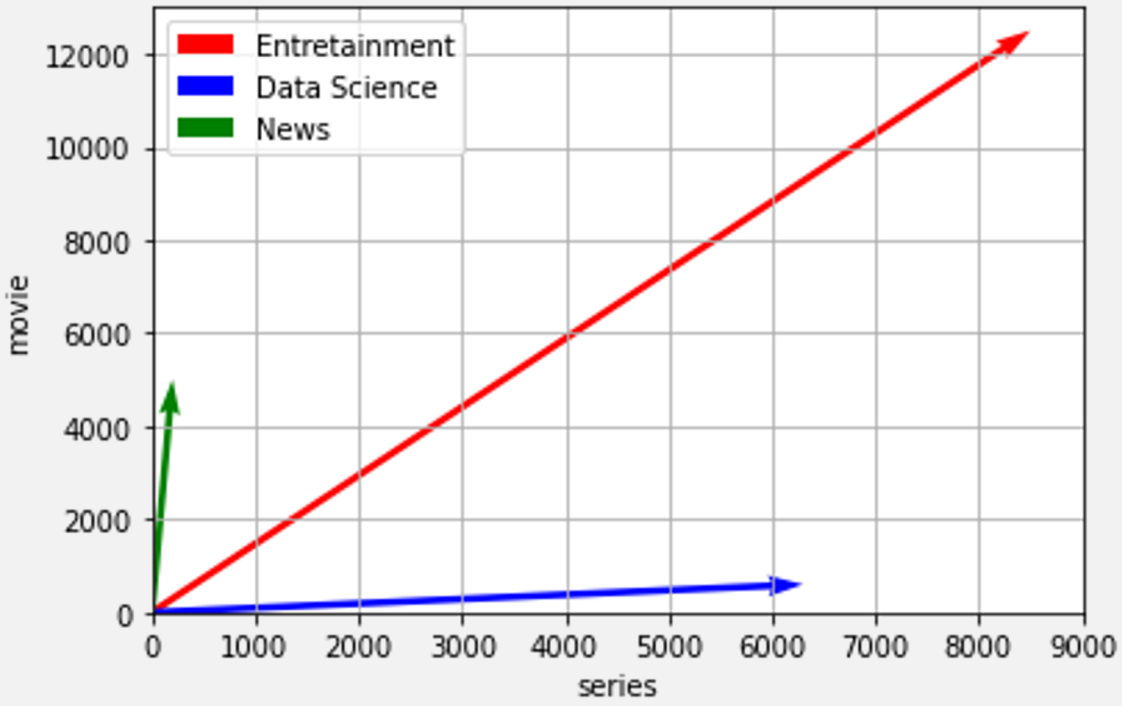
- $B_1 - A_1$
- $B_2 - A_2$

| | Entertainment | Data Science |
|---|---|---|
| series | 8500 | 6300 |
| movie | 12500 | 600 |



# Cosine Similarity

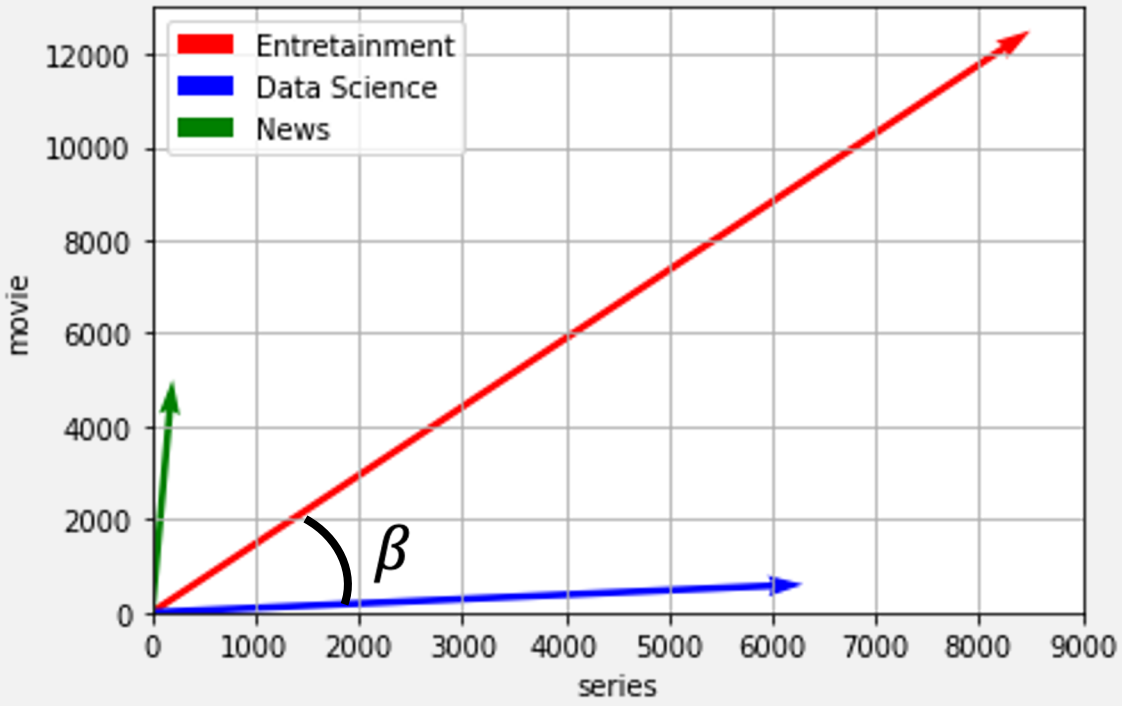
- Problem with Euclidean Distance:
  - The distance depends on the number of words in the corpus of the category
    - See how we have not a lot of words for News.
- The Cosine Similarity evaluates the inner angle between the vectors.
  - Thus evaluates their directions, not size.
  - Not biased by the size difference of the vector representations.



# Cosine Similarity

- $\hat{v} \cdot \hat{w} = \|\hat{v}\| \|\hat{w}\| \cos(\beta)$

$$\cos(\beta) = \frac{\hat{v} \cdot \hat{w}}{\|\hat{v}\| \|\hat{w}\|}$$



# Vector Spaces: Co-occurrence Matrix

- We can make vector representation of **words** analyzing their neighborhood

- Co-occurrence: the number of times two words appear together in the corpus within a given distance.

- "Tell me who your friends are, and I'll tell you who you are."
  - Very strong concept in linguistics

# Vector Spaces: Co-occurrence Matrix

- Word by word design.

I like action series.
I prefer to watch animation action series.

# Vector Spaces: Co-occurrence Matrix

- Word by word design.
- k = 2
- Number of times they co-occur in a distance k.

I like action series.
I prefer to watch animation action series.

| | action | like | animation | like | watch | to | prefer | I |
|---|---|---|---|---|---|---|---|---|
| series | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

# Vector Spaces: Co-occurrence Matrix

- Word by word design.
- k = 2
- Number of times they co-occur in a distance k.

I like action series.
I prefer to watch animation action series.

| | action | like | animation | like | watch | to | prefer | I |
|---|---|---|---|---|---|---|---|---|
| series | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

n = size of the vocabulary = *V*

# Vector Spaces: Co-occurrence Matrix

- Word by word design.
- k = 2 **-> Will impact this a lot!**
- Number of times they co-occur in a distance k.

I like action series.
I prefer to watch animation action series.

| | action | like | animation | like | watch | to | prefer | I |
|---|---|---|---|---|---|---|---|---|
| series | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

n = size of the vocabulary = *V*

# Vector Spaces: Word2Vec

- Perhaps the single most important algorithm in NLP
- A self supervised learning algorithm
  - All we need is the plain text and nothing else

- The main idea is very close to the co-occurrence matrix:
  - To use a word's surrounding/neighborhood
  - But this time, to try to predict it

# Vector Spaces: Word2Vec

- Closed vocabulary - $|V|$
- Fixed amount of dimensions - $d$ (50 ~ 1000, usually)
- Dense - all dimensions have a value and a meaning
  - Sadly, it probably won't be human interpretable
  - A vector for each word, all of them initialized with random values

- There are two methods to implement Word2Vec
  - We'll use the **Skip-Gram with Negative Sampling (SGNS)**

# Vector Spaces: Word2Vec

- Instead of counting words, we have a prediction task
- We take a target word and *k* words before and after it

"[...] this is very important to [...]"

- k = 2, so 4 context words for a single target word
- Using pairs of (w,c) vectors, we feed them to a **binary classifier** model

# Vector Spaces: Word2Vec

"[...] this is very important to [...]"

- In a more mathematical way:
  - There's the probability P(+|*w*,*c*) of the context word being in fact a context word for *w*
  - P(-|*w*,*c*) is the probability that it is **not** a context word for *w*
    - It is defined as: P(-|*w*,*c*) = 1 - P(+|*w*,*c*)
    - It will show up when we talk about the *negative sampling* part

# Vector Spaces: Word2Vec

"[...] this is very important to [...]"

- The classifier model:
  - Will try to output the P(+|*w*,*c*) value
  - But how does it come up with this value?
    - "A word is likely to occur near the target if its embedding vector is similar to the target embedding"
  - Similarity ~ $c \cdot w$ (same idea as the cosine similarity)
    - Is a value between [-1,1], so we need to transform it into a probability

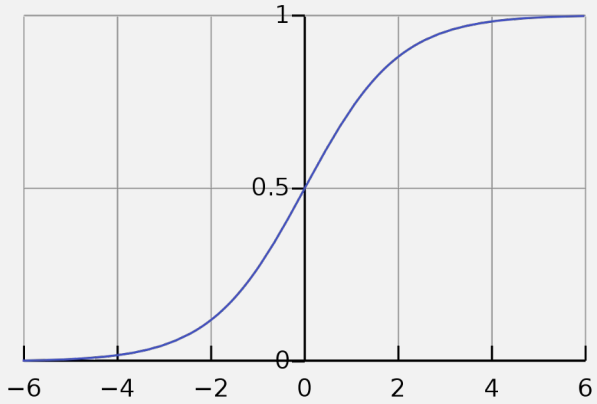
# Vector Spaces: Word2Vec

"[...] this is very important to [...]"

- The classifier model:
  - Will try to output the P(+|*w*,*c*) value using this definition:

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

# Vector Spaces: Word2Vec




"[...] this is very important to [...]"

$c$ $c$ $w$ $c$

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

- The only trainable parameters will be *c* and *w*
- In other words, we're trying to come up with **better vectors** for prediction
- Words that appear together will tend to have similar embeddings

# Vector Spaces: Word2Vec

"[...] this is very important to [...]"

- We don't have just one context word though:

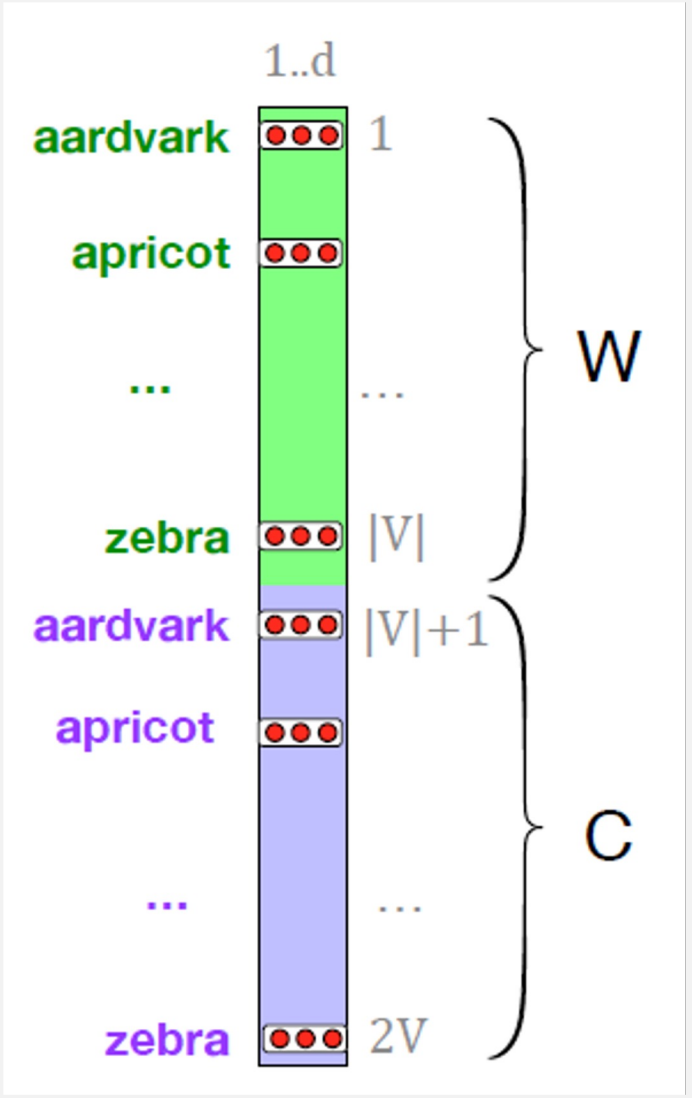
$$P(+|w, c_{1:L}) = \prod_{i=1}^{L} \sigma(\mathbf{c_i} \cdot \mathbf{w})$$

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

# Vector Spaces: Word2Vec

- OBS:
  - Technically, every word will have **two** embeddings
  - One for it as the target word
  - One for it as the context word

  - In the end, we can use either
    - Or even both! (By summing them)



# Vector Spaces: Word2Vec

Negatively sampled word!

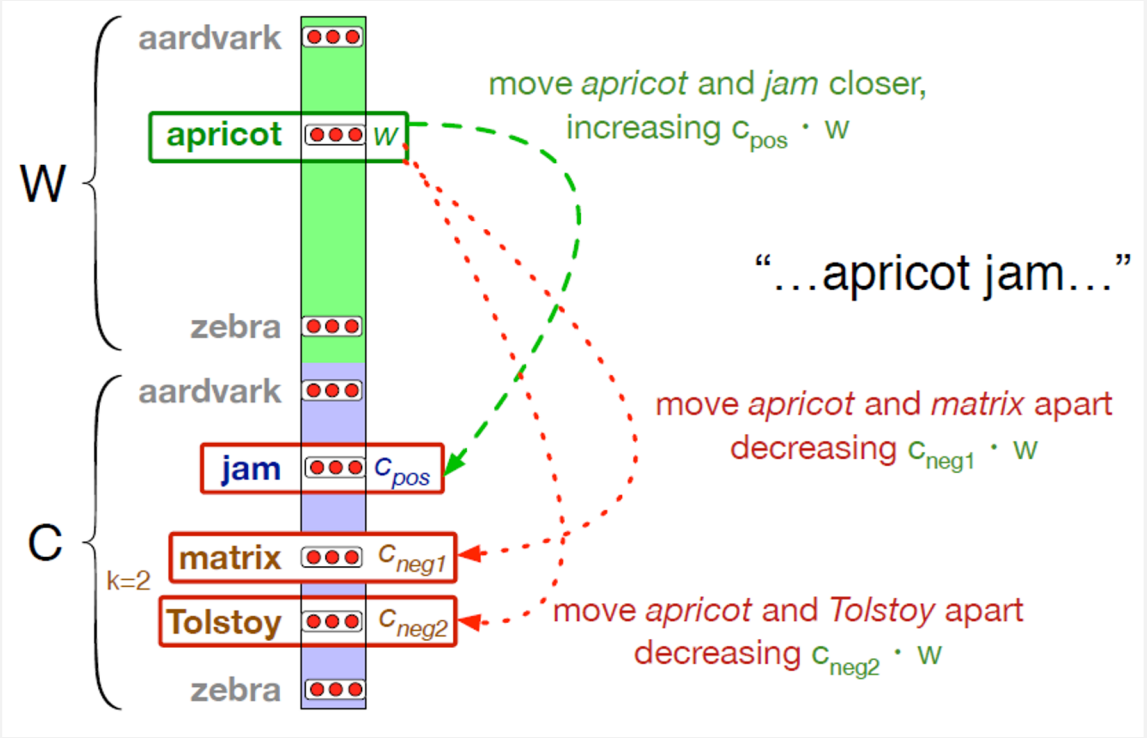
"[...] this is very important to [...]" + "watermelon"

- We'll also randomly sample negative examples
  - That is, words that are not in the context
- We want their similarity to be as negative as possible

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

# Vector Spaces: Word2Vec

- Negative sampling effects:
  - Inside the resulting Vector Space:
    - Will further separate the vector space
    - Similar words will continue spatially close
    - Unrelated words will be even more separated



# Vector Spaces Application

- We can simply scrap the Classifier model and use only the embedding
  - There are **many** pre-trained embeddings we can use:
    - GloVe, FastText, SpaCy's embeddings, etc

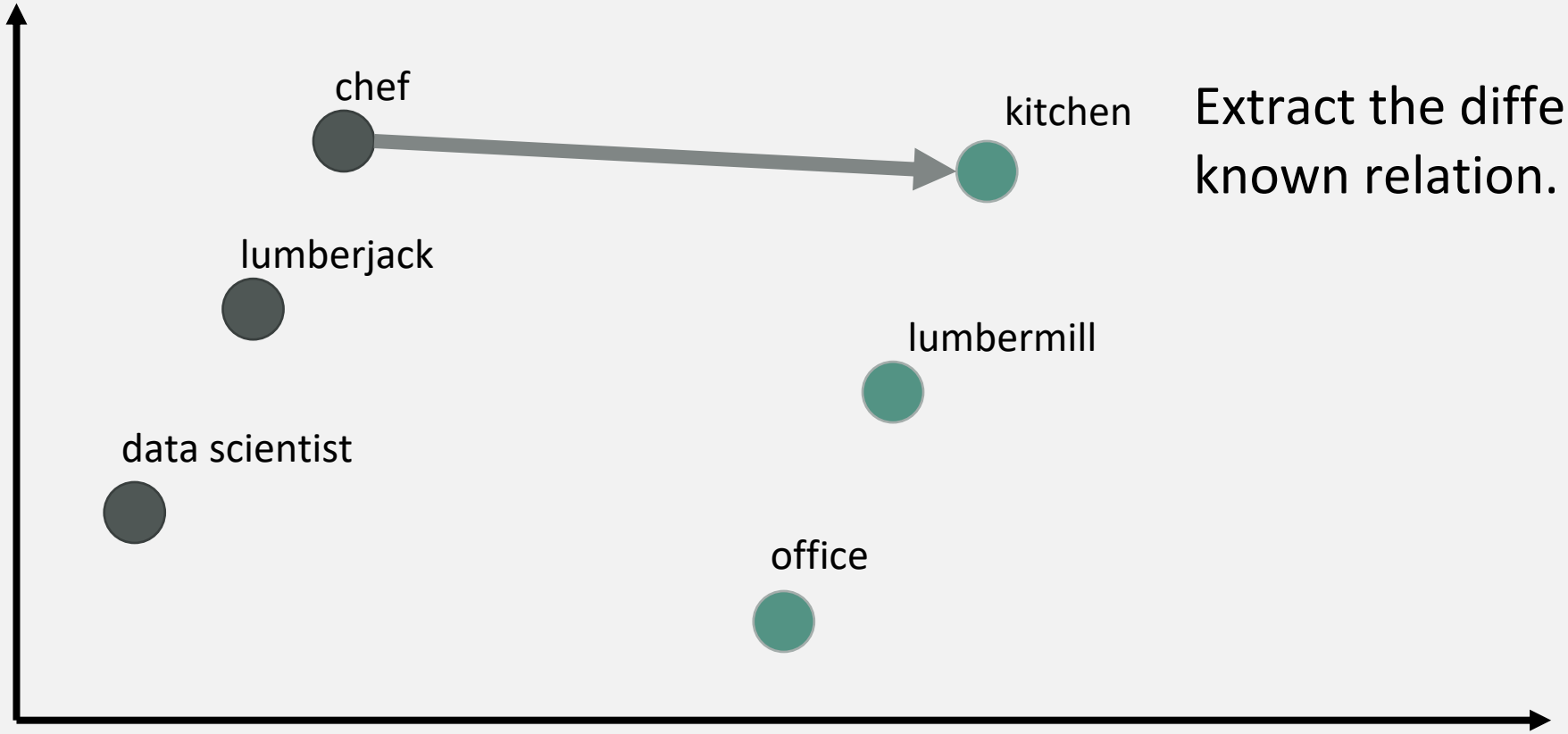
- We can use vector spaces and embedded words to extract unknown relations between words.
- Given that a vector space captures the relative meaning of works, the distance between two words is closely related to their relations.

# Vector Spaces Application

- Example: finding out workplaces.

# Vector Spaces Application

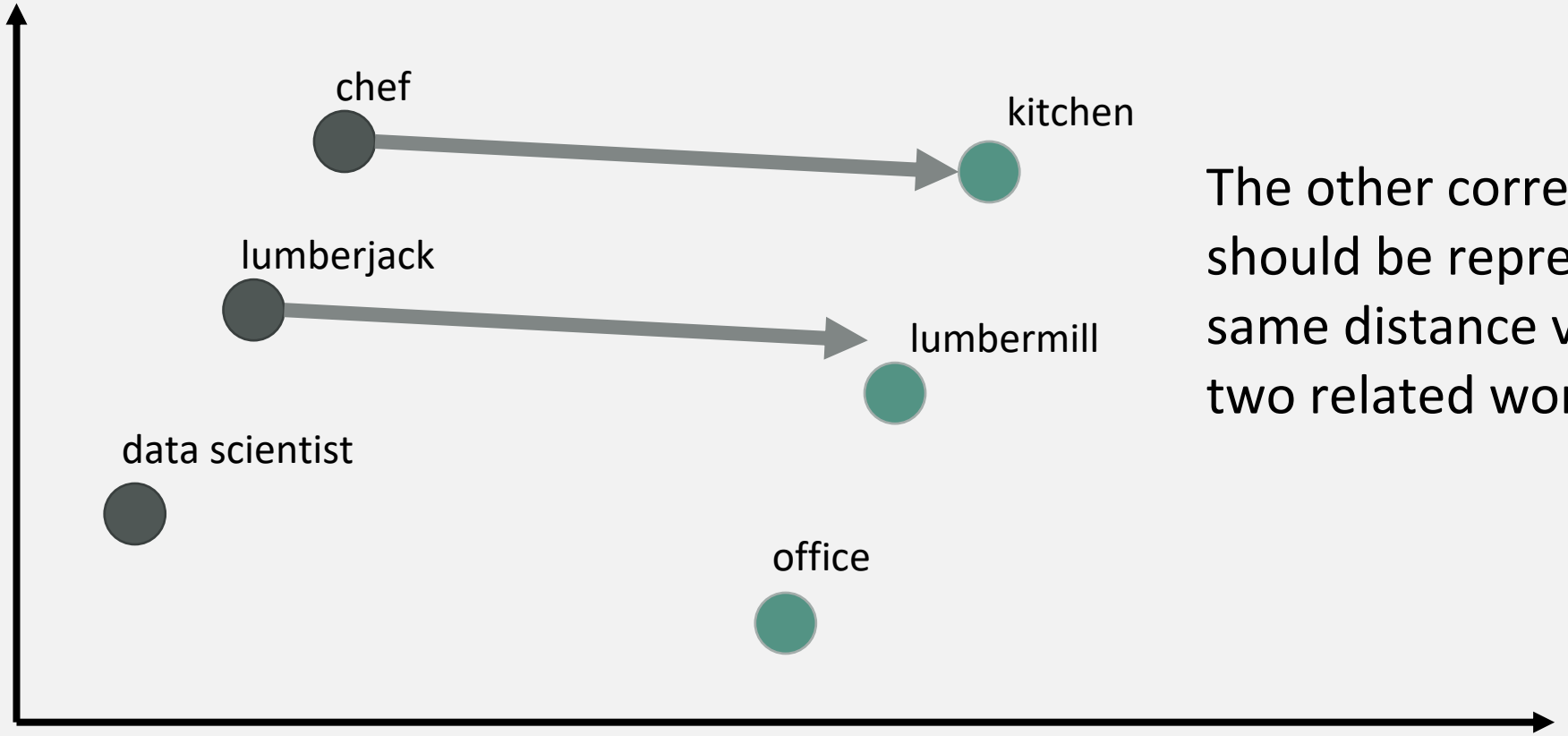
- Example: finding out workplaces.




Extract the difference from a known relation.

# Vector Spaces Application

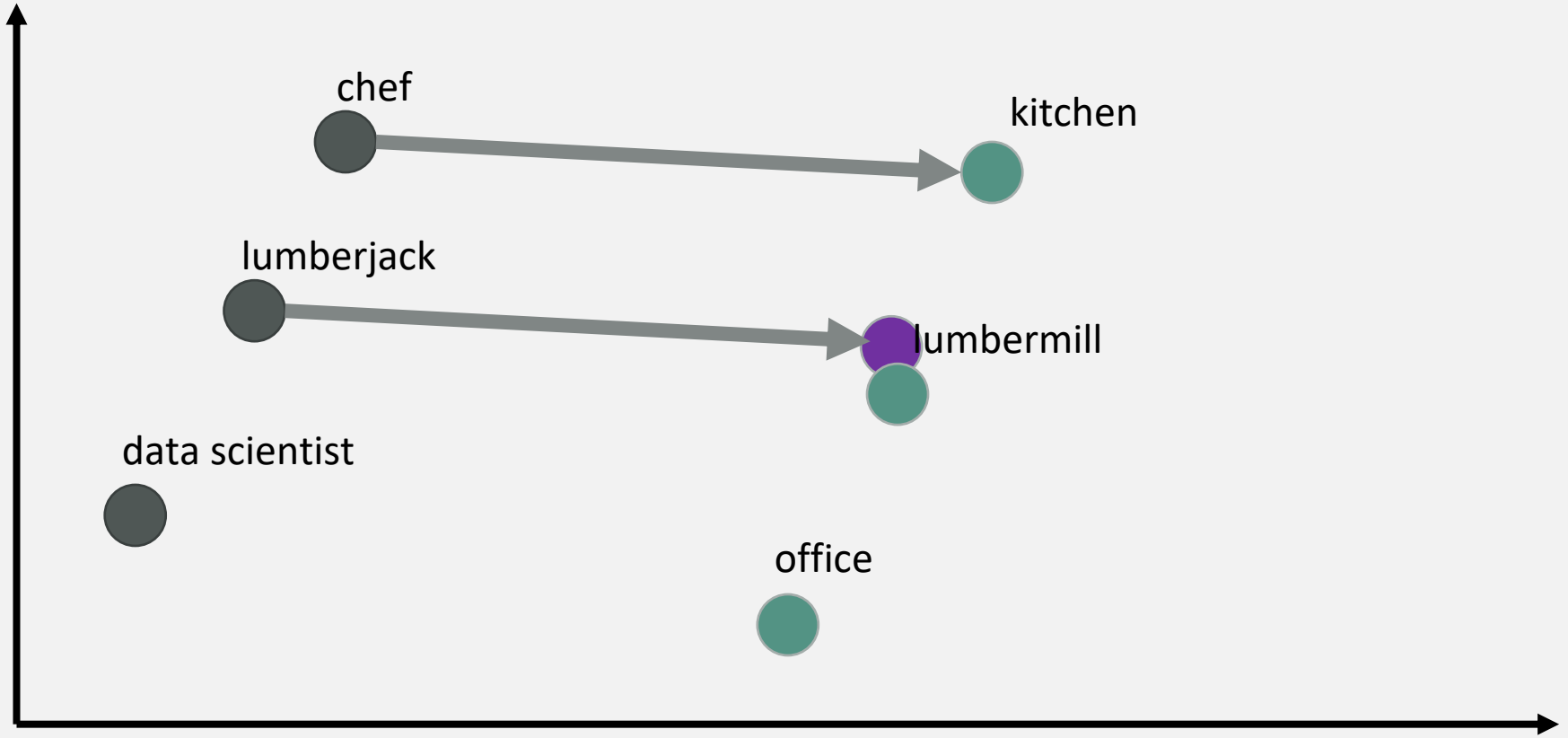
- Example: finding out workplaces.




The other correct relations should be represented by the same distance vector between two related words.

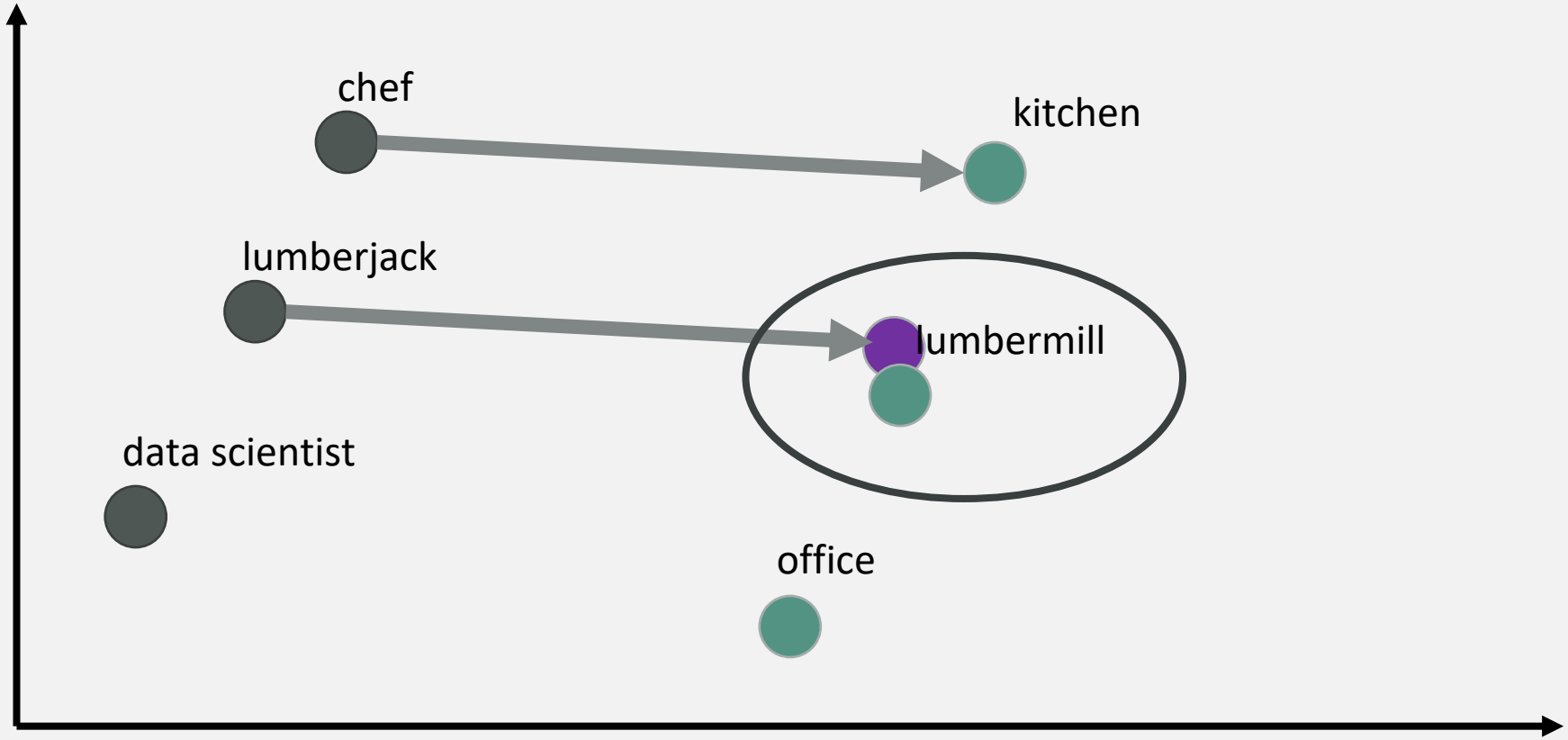
# Vector Spaces Application

- Example: finding out workplaces.



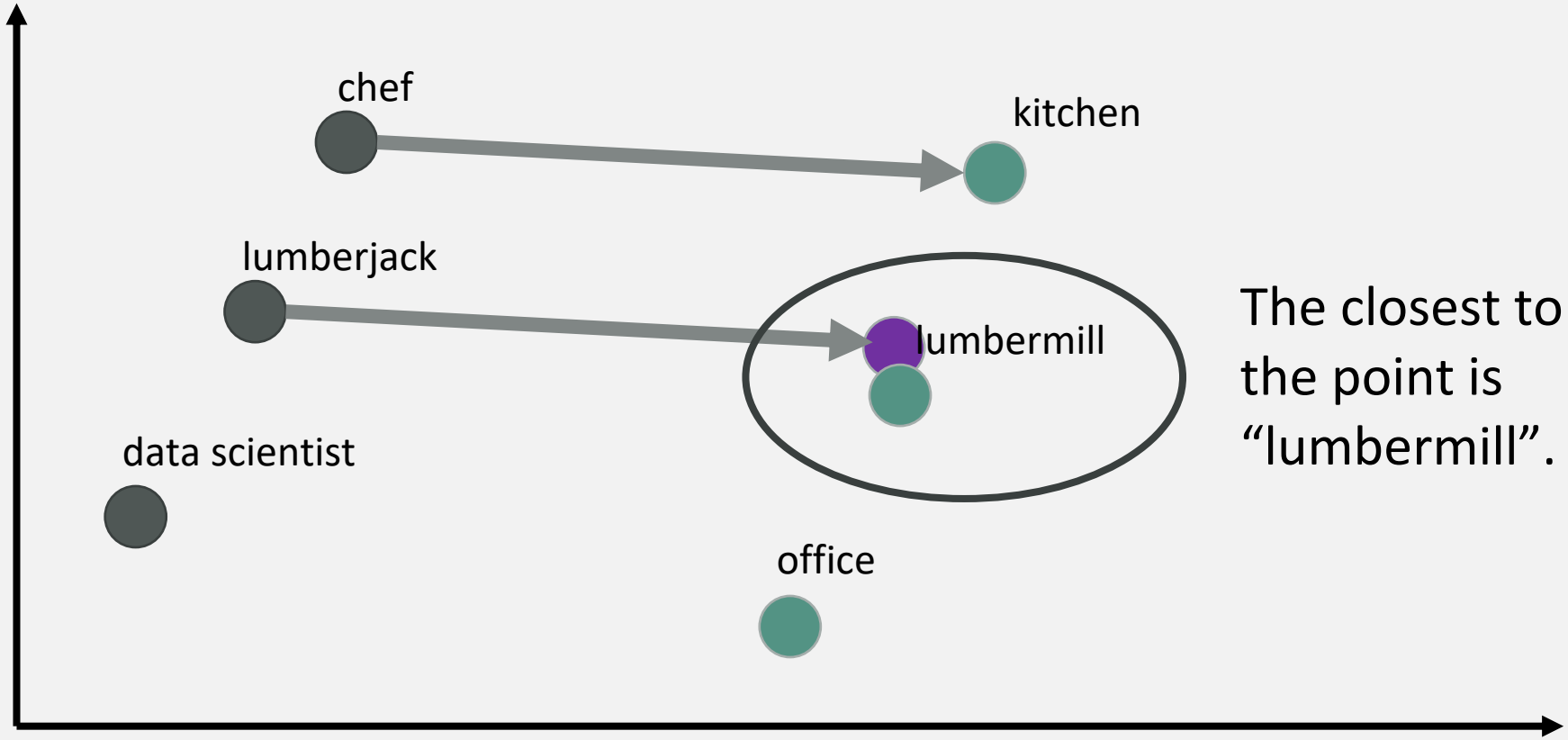
# Vector Spaces Application

- Example: finding out workplaces.



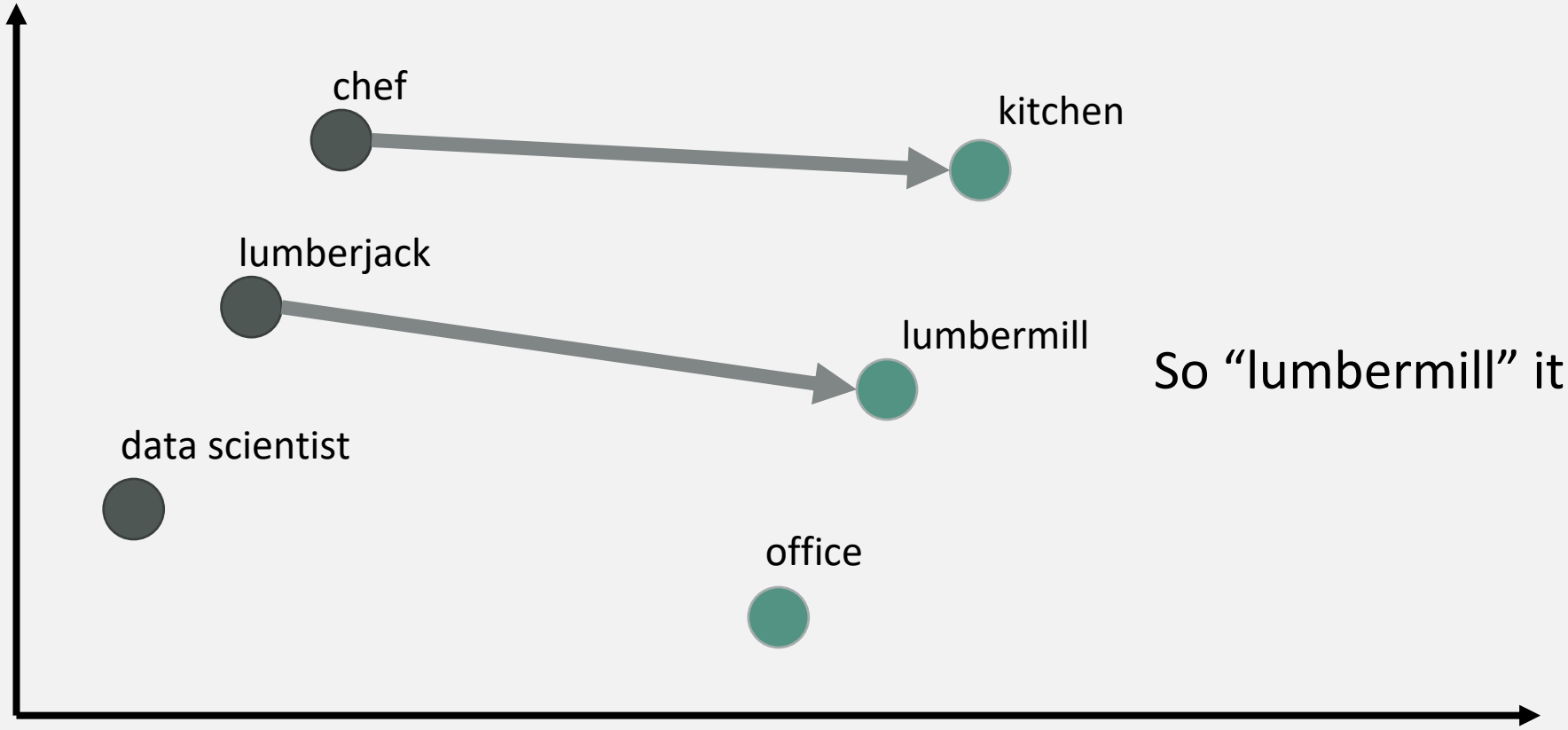
# Vector Spaces Application

- Example: finding out workplaces.



# Vector Spaces Application

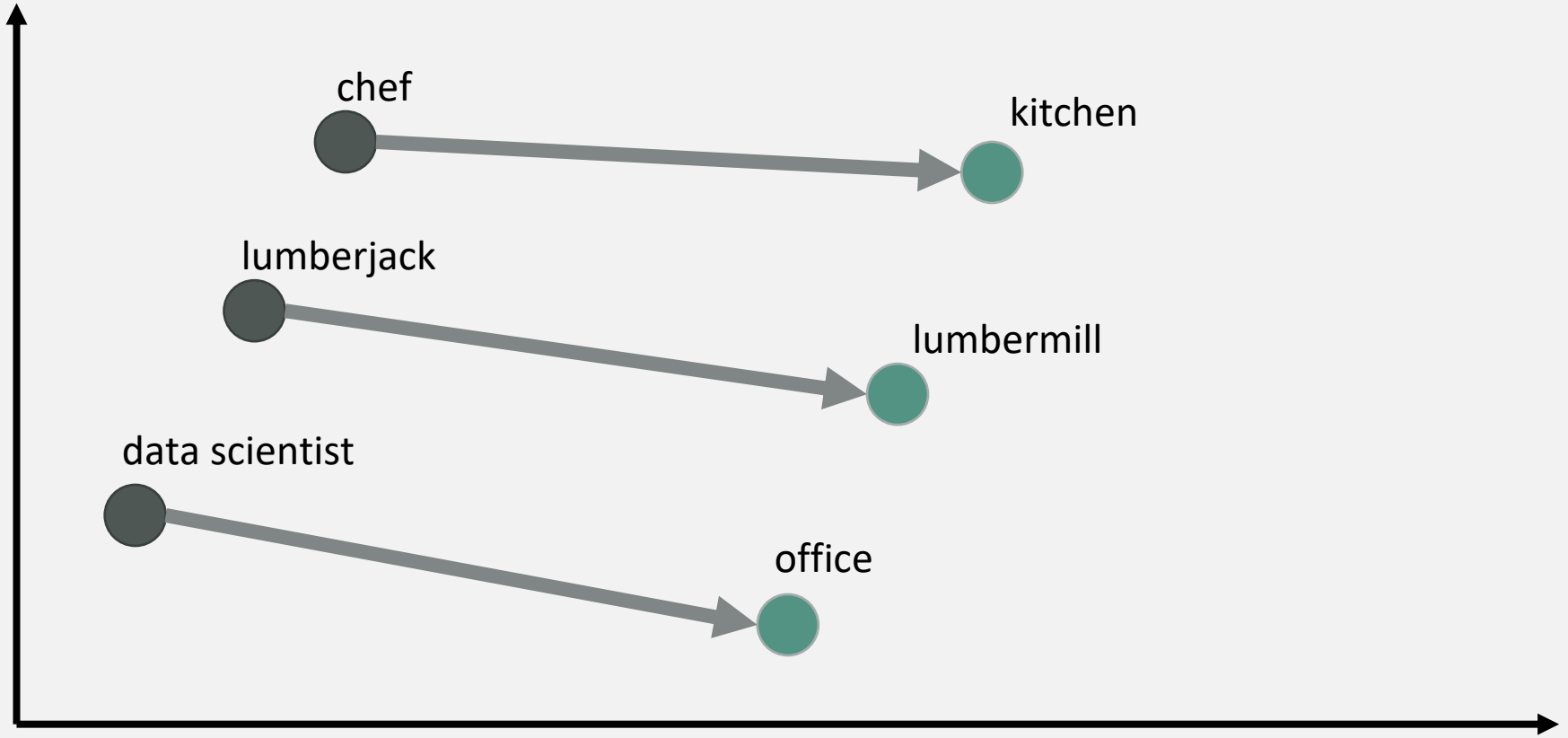
- Example: finding out workplaces.




So “lumbermill” it is.

# Vector Spaces Application

- Example: finding out workplaces.



# Vector Spaces: Interpretation

- Most of the cases we have vector spaces of high dimensions
- It is hard for human beings to visualize a vector space with more than 3 dimensions, let alone one with hundreds
- The dimensions will hardly ever translate to something human-interpretable
- The vector space will be **extremely** dependent on the corpus given
  - Also, they are **static**
    - They won't change their meaning given the context!
    - "Apple launches new Iphone" vs "I just ate an apple" -> They're the same vector!

# Word Embeddings – What are they good for?

- A practical use example of word embeddings and vector spaces is text **translation**.

# Word Embeddings – Translation

- Suppose we have a vector space with words in English:

| Word | Embedding |
|---|---|
| father | [0.2, -0.1, …, 0.8] |
| kitchen | [0.4, 1.2, …, -0.7] |
| … | … |
| dog | [-1.3, 0.9, …, 0.2] |

English

# Word Embeddings – Translation

- Suppose we have a vector space with words in English:
- And another one for German

| Word | Embedding |
|---|---|
| father | [0.2, -0.1, …, 0.8] |
| kitchen | [0.4, 1.2, …, -0.7] |
| … | … |
| dog | [-1.3, 0.9, …, 0.2] |

English

| Word | Embedding |
|---|---|
| Vater | [-2.2, -1.1, …, 0.2] |
| Küche | [1.2, -1.0, …, -1.7] |
| … | … |
| Hund | [0.9, 2.1, …, -0.3] |

German

# Word Embeddings – Translation

- Suppose we have a vector space with words in English:
- And another one for German
- (In theory) there could be a vector that transforms the English vector space to the German one (through vector multiplication)

| Word | Embedding |
|---|---|
| father | [0.2, -0.1, …, 0.8] |
| kitchen | [0.4, 1.2, …, -0.7] |
| … | … |
| dog | [-1.3, 0.9, …, 0.2] |

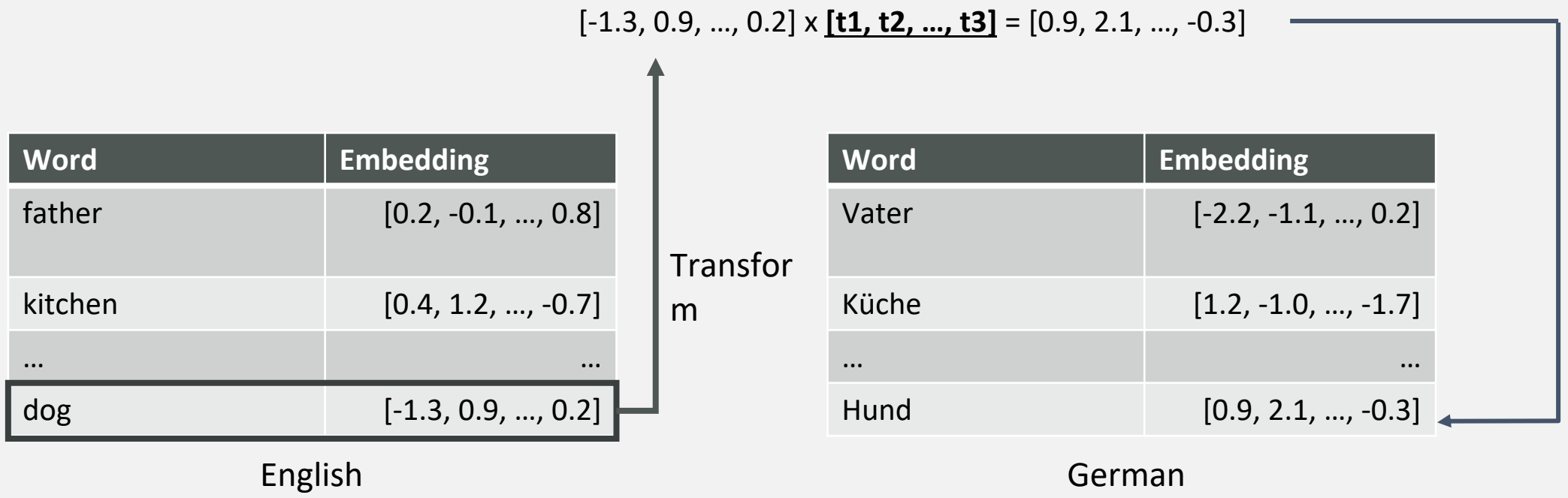
English

| Word | Embedding |
|---|---|
| Vater | [-2.2, -1.1, …, 0.2] |
| Küche | [1.2, -1.0, …, -1.7] |
| … | … |
| Hund | [0.9, 2.1, …, -0.3] |

German

# Word Embeddings – Translation

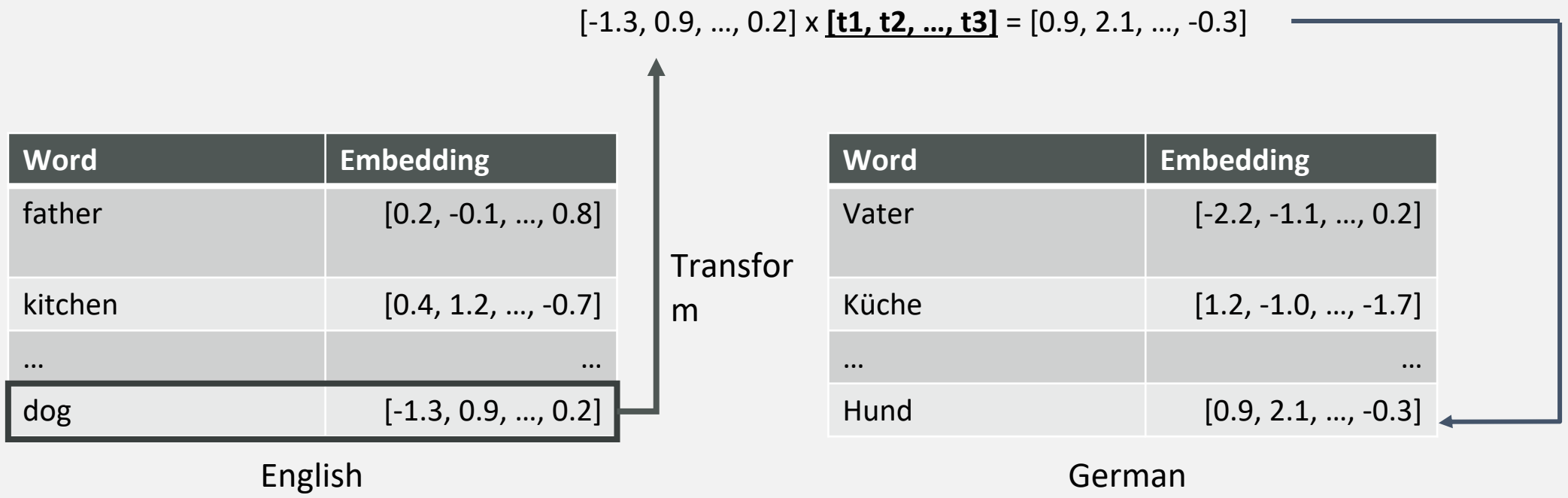
By multiplying the English vector by the transformation vector we should be able to land close or at the German embedding



# Word Embeddings – Translation

From a linguistics point of view there's probably no one vector that's able to do that

- Results will probably be very poor, even if we find a "best" vector



# Word Embeddings – Translation

- If we have aligned vocabularies, we can create a transformation **matrix**
  - Each vector will have its own transformation vector
  - There are algorithms to ensure the translation will be as sane as possible

$$\begin{pmatrix} [\text{father}] \\ [\text{kitchen}] \\ \dots \\ [\text{dog}] \end{pmatrix} \rightarrow \begin{pmatrix} [\text{Vater}] \\ [\text{Küche}] \\ \dots \\ [\text{Hund}] \end{pmatrix}$$

# Word Embeddings – Translation

- Objective: find a matrix that transforms vectors in one vector space to vectors in another vector space:

$$XR \approx Y$$

- Needs lots of training data.
- In our example:
  - X is a matrix with English word vectors.
  - Y is a matrix with French word vectors.
  - The words shall be aligned.

$$\begin{pmatrix} [\text{father}] \\ [\text{kitchen}] \\ \dots \\ [\text{dog}] \end{pmatrix} \rightarrow \begin{pmatrix} [\text{Vater}] \\ [\text{Küche}] \\ \dots \\ [\text{Hund}] \end{pmatrix}$$

# Word Embeddings – Translation

- Objective: find a matrix that transforms vectors in one vector space to vectors in another vector space:

$$XR \approx Y$$

- Objective: minimize the distance between $XR$ and $Y$.

$$Loss = \|XR - Y\|_F$$

# Word Embeddings – Translation

1. Initialize with a random R
2. Execute in a loop:

$$Loss = \|XR - Y\|_F$$

$$gradient = g = \frac{dLoss}{dR}$$

$$R = R - \alpha g$$

# Word Embeddings – Translation

1. Initialize with a random R
2. Execute in a loop:

$$Loss = \|XR - Y\|_F$$ Compute loss

$$gradient = g = \frac{dLoss}{dR}$$

$$R = R - \alpha g$$

# Word Embeddings – Translation

1. Initialize with a random R
2. Execute in a loop:

$$Loss = \|XR - Y\|_F$$ Compute loss

$$gradient = g = \frac{dLoss}{dR}$$ Compute gradient

$$R = R - \alpha g$$

# Word Embeddings – Translation

1. Initialize with a random R
2. Execute in a loop:

$$Loss = \|XR - Y\|_F$$ Compute loss

$$gradient = g = \frac{dLoss}{dR}$$ Compute gradient

$$R = R - \alpha g$$ Update R

$\alpha$ = learning rate

# Word Embeddings – Translation

- $$Loss = \|XR - Y\|_F$$

- Frobenius norm:

$$\|X\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

# Word Embeddings – Translation

- In practice, it's easier to use the square of the Frobenius norm:

$$Loss = \|XR - Y\|_F^2$$

- So that it cancels the square root.

$$g = \frac{dLoss}{dR} = \frac{2}{m}\left(X^T(XR - Y)\right)$$

- $m = number\ of\ words\ in\ the\ training\ set$
- This is easier to implement in Python.

# Autocorrection

Using embeddings to disambiguate

# Autocorrection

- Let's say you already have a errors-amount based correction system
- The input sentence was *"Do you have a **pem**?"*
  - We humans know by context that it's likely that the word was ***pen***
  - For a machine "pen" and "pea" are both 1 substitution apart

- We can get all 1-error apart words in our vocabulary
  - Compute the cosine similarity between them and the k previous words in the sentence
  - Get the most probable one
- By doing this, our correction guess is starting to take context into account

# Thank you!