

# Word Embedding

Amir Mohammad Fakhimi

# How do we have usable meaning in a computer?

**Previously commonest NLP solution:** Use, e.g., **WordNet**, a thesaurus containing lists of **synonym sets** and **hypernyms** (“is a” relationships)

*e.g., synonym sets containing “good”:*

```
from nltk.corpus import wordnet as wn
poses = { 'n': 'noun', 'v': 'verb', 's': 'adj (s)', 'a': 'adj', 'r': 'adv' }
for synset in wn.synsets("good"):
    print("{}: {}".format(poses[synset.pos()],
        ", ".join([l.name() for l in synset.lemmas()])))
```

```
noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good
adj: good
adj (sat): estimable, good, honorable, respectable
adj (sat): beneficial, good
adj (sat): good
adj (sat): good, just, upright
...
adverb: well, good
adverb: thoroughly, soundly, good
```

*e.g., hypernyms of “panda”:*

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(panda.closure(hyper))
```

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

# Problems with resources like WordNet

- A useful resource but missing nuance:
  - e.g., “**proficient**” is listed as a synonym for “**good**”  
This is only correct in some contexts
  - Also, WordNet list offensive synonyms in some synonym sets without any coverage of the connotations or appropriateness of words
- Missing new meanings of words:
  - e.g., **wicked, badass, nifty, wizard, genius, ninja, bombest**
  - Impossible to keep up-to-date!
- Subjective
- Requires human labor to create and adapt
- Can’t be used to accurately compute word similarity (see following slides)

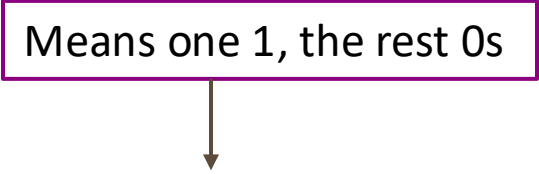
- What is “Embedding”?
- What is “Word Embedding”?
- What are static and contextual embeddings?

# Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols:

hotel, conference, motel – a **localist** representation

Means one 1, the rest 0s



Such symbols for words can be represented by **one-hot** vectors:

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

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

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

# Problem with words as discrete symbols

**Example:** in web search, if a user searches for “Seattle motel”, we would like to match documents containing “Seattle hotel”

But:

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

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

These two vectors are orthogonal

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

## Solution:

- Could try to rely on WordNet’s list of synonyms to get similarity?
  - But it is well-known to fail badly: incompleteness, etc.
- **Instead: learn to encode similarity in the vectors themselves**

# Representing words by their context



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

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

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

# Word vectors

We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts, measuring similarity as the vector dot (scalar) product  
Or Cosine Similarity (Normalized)

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

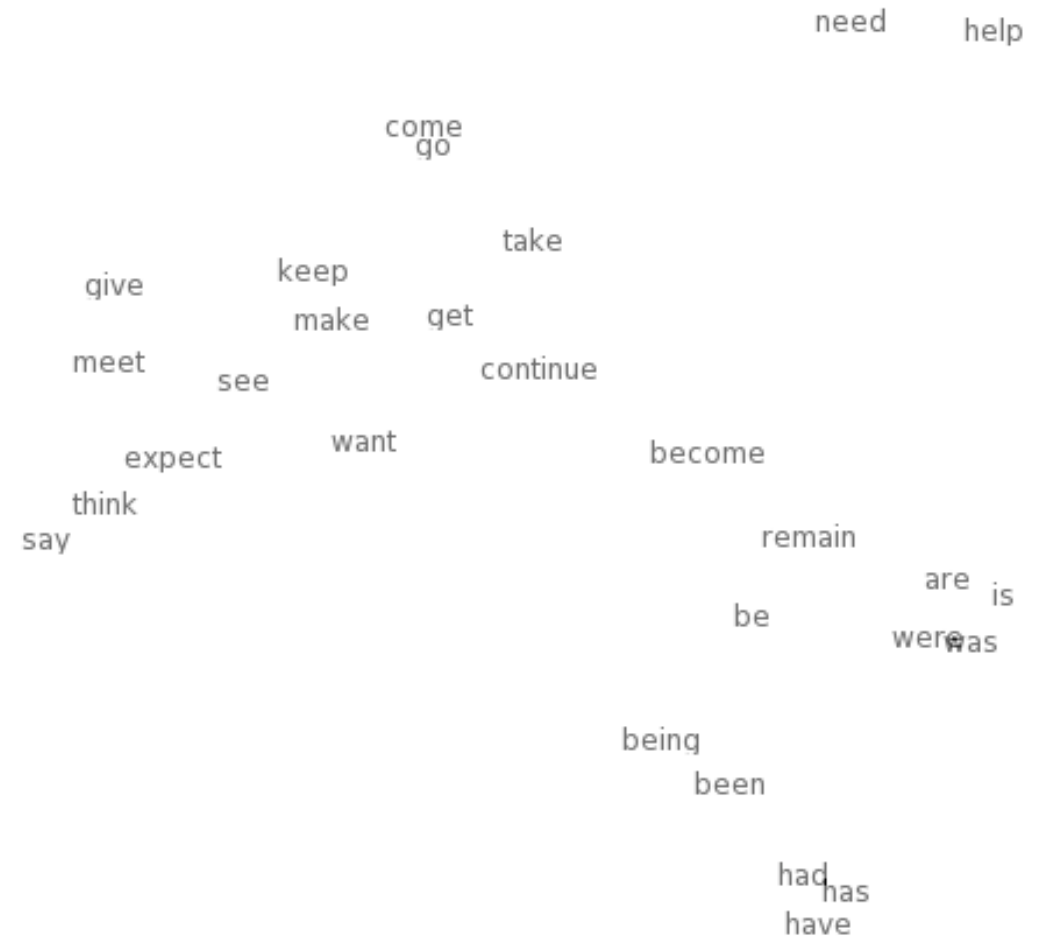
$$\textit{monetary} = \begin{pmatrix} 0.413 \\ 0.582 \\ -0.007 \\ 0.247 \\ 0.216 \\ -0.718 \\ 0.147 \\ 0.051 \end{pmatrix}$$

Note: word vectors are also called (word) embeddings or (neural) word representations  
They are a distributed representation



# Word meaning as a neural word vector – visualization

*expect* =

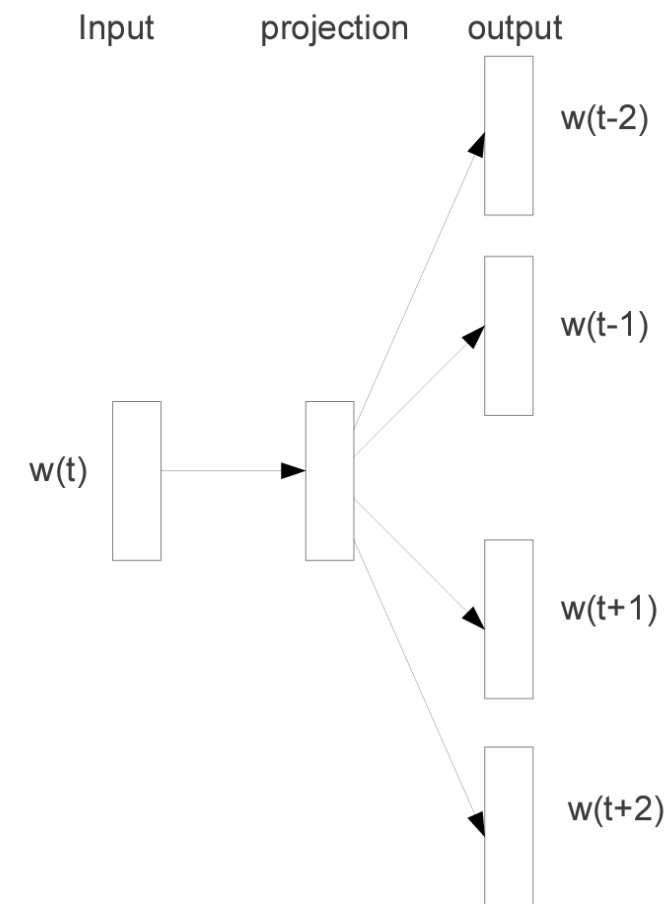
$$\begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \\ 0.487 \end{pmatrix}$$


### 3. Word2vec: Overview

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

Idea:

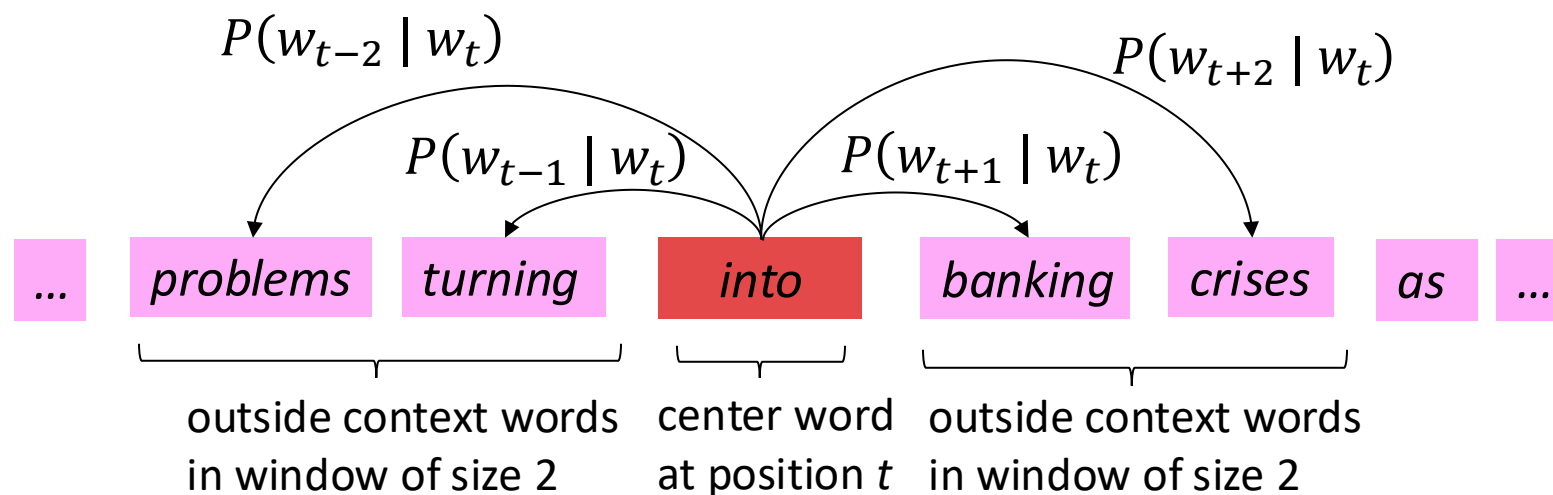
- We have a large corpus (“body”) of text: a long list of words
- Every word in a fixed vocabulary is represented by a **vector**
- Go through each position  $t$  in the text, which has a center word  $c$  and context (“outside”) words  $o$
- Use the **similarity of the word vectors** for  $c$  and 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



Skip-gram model  
(Mikolov et al. 2013)

# Word2Vec Overview

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



# Word2Vec: objective function

For each position  $t = 1, \dots, T$ , predict context words within a window of fixed size  $m$ , given center word  $w_t$ . Data likelihood:

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

$\theta$  is all variables  
to be optimized

sometimes called a *cost* or *loss* function

The **objective function**  $J(\theta)$  is the **(average) negative log likelihood**: We divided by  $T$  because the raw negative log-likelihood grows linearly with corpus length.

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

Minimizing objective function  $\Leftrightarrow$  Maximizing predictive accuracy

- *Likelihood*  $= P(w_1, \dots, w_T; \theta) = \prod_{t=1}^T P(w_t \mid w_1, \dots, w_{t-1}; \theta)$
- Instead of modelling each word conditioned on its entire history, Skip-Gram treats each position  $t$  as a “center” word  $w_t$  generating each context word  $w_{t+j}$  (for  $j = -m, \dots, +m, j \neq 0$ )

# Word2Vec: objective function

- We want to minimize the objective function:

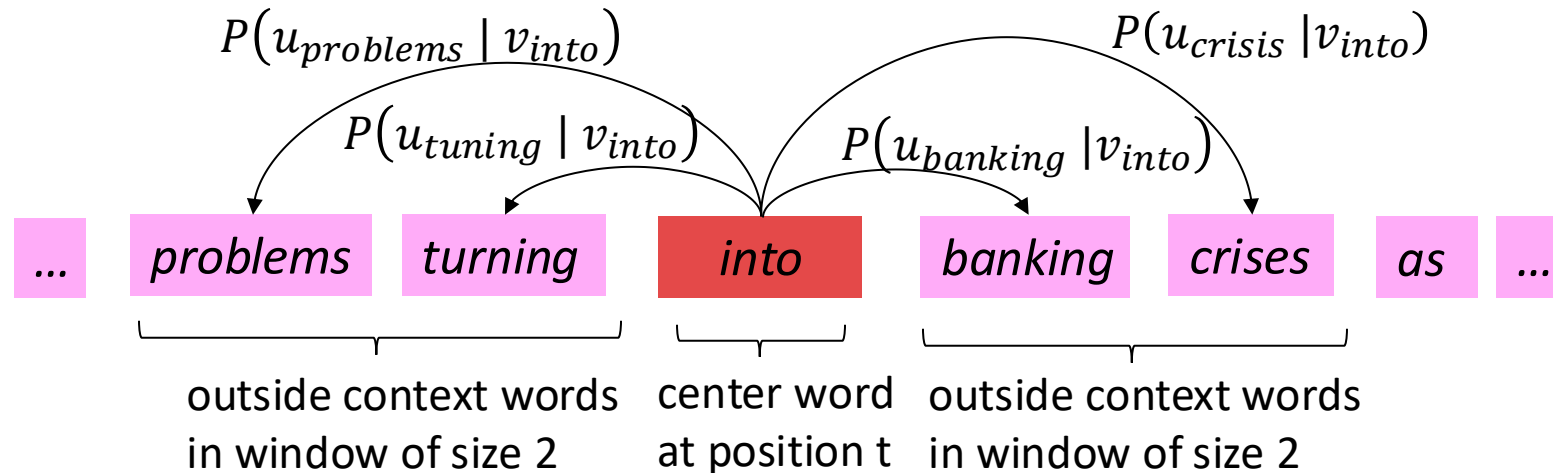
$$J(\theta) = -\frac{1}{T} \sum_{t=1}^T \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j} | w_t; \theta)$$

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

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

# Word2Vec with Vectors

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



# Word2Vec: prediction function

② Exponentiation makes anything positive

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

① Dot product compares similarity of  $o$  and  $c$ .

$$u^T v = u \cdot v = \sum_{i=1}^n u_i v_i$$

Larger dot product = larger probability

③ Normalize over entire vocabulary to give probability distribution

- This is an example of the **softmax function**  $\mathbb{R}^n \rightarrow (0,1)^n$  ← Open region

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

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

But sort of a weird name because it returns a distribution!

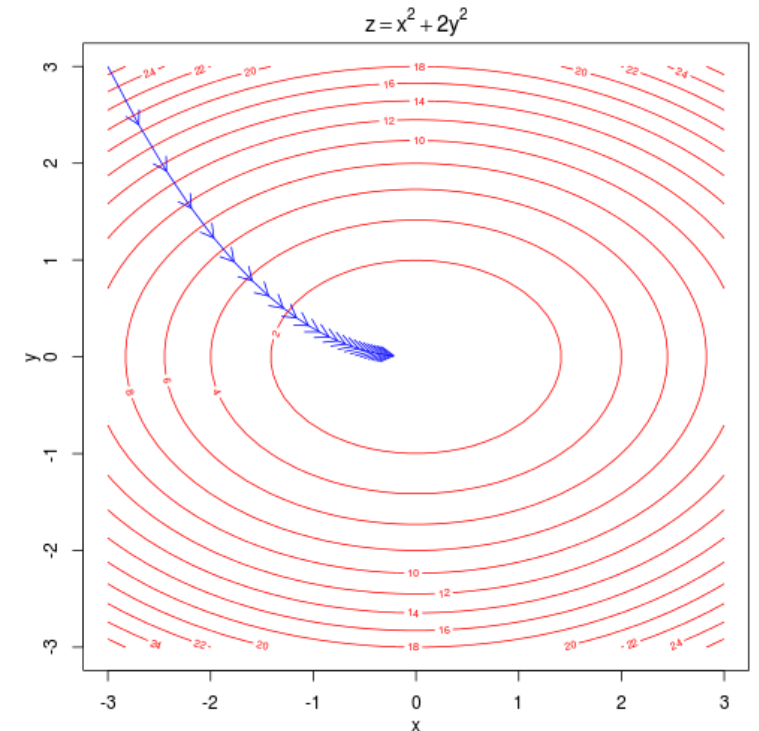


# To train the model: Optimize value of parameters to minimize loss

To train a model, we gradually adjust parameters to minimize a loss

- Recall:  $\theta$  represents **all** the model parameters, in one long vector
- In our case, with  $d$ -dimensional vectors and  $V$ -many words, we have  $\rightarrow$
- Remember: every word has two vectors

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



- We optimize these parameters by walking down the gradient (see right figure)
- We compute **all** vector gradients!

- **fastText:** fastText is a library and set of models developed by Facebook AI Research for learning word representations and performing text classification. FastText extends the Word2Vec idea by incorporating subword (character-level) information.
- → We'll see more in the workshop!

- Unsupervised fastText (embeddings)
  - **Subword modeling:** Instead of learning a single vector for each word type, fastText represents a word  $w$  as the sum of vectors for all of its character  $n$ -grams (plus the whole word).
  - E.g. for the word “where” and  $n$ -grams of length 3, you’d break it into  $\langle wh, whe, her, ere, re \rangle$  (with boundary symbols  $\langle$  and  $\rangle$ ), plus the special token for “where” itself.
  - **Embedding lookup:** each  $n$ -gram has its own vector. The final word embedding is:
    - $$v_w = \sum_{g \in \mathcal{G}_w} z_g$$
    - where  $\mathcal{G}_w$  is the set of  $n$ -grams for  $w$ , and  $z_g$  is the learned embedding for  $n$ -gram  $g$ .
    - We do not cover it’s training objective.

- Why subwords? (Even we do it in byte-level!)
  - **Rare words & morphology:** by sharing n-gram vectors across many words, fastText can build good embeddings for rare or even unseen (out-of-vocabulary) words, especially in morphologically rich languages.
  - **Better generalization:** words with similar character sequences end up close in embedding space.

- Supervised fastText (text classification)
  - fastText also provides a supervised model that's extremely fast and effective for document tagging:
  - Represent each document as the average of its word (or subword-augmented) embeddings.
  - Linear classifier: A single weight matrix  $W_{\text{clf}} \in \mathbb{R}^{C \times d}$  maps the d-dimensional average embedding into scores for C classes.
  - Softmax + cross-entropy loss trains both the embeddings and the classifier weights jointly.

# Information Retrieval

## بازنمایی متنی

	W1	W2	...	Wn
Doc 1				
Doc 2			۱ یا ۰	
⋮				
Doc m				

[1 0 1 0 1]

[1 0 1 0 0]

## بازنمایی متن با استفاده از one-hot

	W1	W2	...	Wn
W 1				
W 2				
⋮				
W n				

Window-size = 5

---

---

---



## بازنمایی متنی

	W1	W2	...	Wn
Doc 1				
Doc 2		Freq		
⋮				
Doc m				

## بازنمایی متنی با استفاده از tf-idf

	W1	W2	...	Wm
Doc 1				
Doc 2			TF-idf	
⋮				
Doc N				

$$\text{tf}(t, D) = \frac{\#(t, D)}{\max_{t' \in D} \#(t', D)}$$

$$\text{idf}(t) = \log \frac{N}{\sum_{D:t \in D} 1}$$

$$\text{tf. idf}(t, D) = \text{tf}(t, D) \cdot \text{idf}(t)$$

Inverse Document Frequency (IDF): Down weight very common words (“the”, “and”, etc.) and boost rare ones.

- Why log IDF?
  - **Dampens extremes:** Without the log, a term that appears in only one document would get weight  $N/1 = N$ , which can be enormous if you have tens of thousands of docs. Taking  $\log(N)$  brings that back into a more reasonable range.
  - **Turns ratios into differences:** Because  $\log(a/b) = \log a - \log b$ , IDF becomes an additive correction when you take  $\log(\text{TF} \times \text{IDF})$ . This often makes downstream models (which work additively in log-space) behave more naturally.
  - **Better reflects informativeness:** Doubling a term's df from  $1 \rightarrow 2$  halves its raw IDF from  $\frac{10^6}{1} = 1\,000\,000$  to  $500\,000$  (a 500 000-point drop), while doubling from  $100 \rightarrow 200$  halves it from  $10\,000$  to  $5\,000$  (only a 5 000-point drop). In log-IDF, both doublings subtract the same amount ( $\ln(10^6) - \ln(5 \times 10^5) = \ln 2 \approx 0.69$  and  $\ln(10^4) - \ln(5 \times 10^3) = \ln 2 \approx 0.69$ ), so the log makes each “ $\times 2$ ” step cost exactly the same, smoothing out extreme jumps.

## Sources:

- Stanford, CS224N (NLP Course), winter 2025, [Link](#)
- Sharif University of Technology, 40677 (NLP Course), Spring 2024