

COP509

# Natural Language Processing

---

Vector Space Model

Complete Worked Example with Shakespeare

Prof. Georgina Cosma  
Department of Computer Science  
Loughborough University

# What This Lecture Covers

**One complete example from start to finish.**

We will work through **every calculation** step by step:

## Part A: Build the Index (once)

1. Count term frequencies
2. Calculate TF (log frequencies)
3. Calculate IDF (rarity)
4. Combine into TF-IDF
5. Normalise document vectors

## Part B: Answer a Query

6. Create query vector
7. Normalise query vector
8. Compute dot products
9. Rank the results

## Key Insight

**Weight** = Local (TF) × Global (IDF) × Length Normalisation

# The Problem We Are Solving

## Task

Given a **query** and a **collection of documents**, rank the documents from most relevant to least relevant.

## Our Example:

**Query:** "BRUTUS CAESAR"

**Documents:** Three Shakespeare plays

- ▶  $d_1$ : *Julius Caesar*
- ▶  $d_2$ : *Anthony and Cleopatra*
- ▶  $d_3$ : *The Tempest*

**Question:** Which play is most relevant to the query "BRUTUS CAESAR"?

**Expected answer:** Julius Caesar (it's literally about Brutus and Caesar!)

# Part A

## Building the Index

(Done once when documents are added to the system)

# Step 1

## Count Term Frequencies

How many times does each word appear in each document?

## Step 1: Raw Term Counts

We count how many times each word appears in each document:

<b>Term</b>	<b>Julius Caesar</b>	<b>Ant &amp; Cleo</b>	<b>Tempest</b>
BRUTUS	40	5	0
CAESAR	50	30	0
MERCY	2	5	8

**What this tells us:**

- ▶ BRUTUS appears 40 times in Julius Caesar, 5 times in Ant & Cleo, 0 times in Tempest
- ▶ CAESAR appears 50 times in Julius Caesar, 30 times in Ant & Cleo
- ▶ MERCY appears in all three plays

## Step 2

Calculate Term Frequency (TF)

Use logarithms to dampen the raw counts

## Step 2: Why Use Logarithms?

**Problem:** A word appearing 100 times is NOT 100 times more important than a word appearing once.

**Example:** If “CAESAR” appears 100 times and “BRUTUS” appears 10 times, is CAESAR really  $10\times$  more important? Probably not!

**Solution:** Use **logarithms** to “dampen” large counts.

### What Logarithms Do

Raw Count	$\log_{10}(\text{count})$	$\text{TF} = 1 + \log$
1	0	1.00
10	1	2.00
100	2	3.00
1000	3	4.00

A  $1000\times$  increase in count only gives a  $4\times$  increase in weight!

## Step 2: The TF Formula

### Term Frequency Formula

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

**Why add 1?** So that a word appearing once gets weight 1, not 0.

**Let's calculate TF for BRUTUS:**

Julius Caesar (count = 40):  $\text{tf} = 1 + \log_{10}(40) = 1 + 1.60 = \mathbf{2.60}$

Ant & Cleo (count = 5):  $\text{tf} = 1 + \log_{10}(5) = 1 + 0.70 = \mathbf{1.70}$

Tempest (count = 0):  $\text{tf} = \mathbf{0}$

## Step 2: Complete TF Calculations

**Applying the formula to all terms:**

### **CAESAR:**

- ▶ Julius Caesar:  $1 + \log_{10}(50) = 1 + 1.70 = 2.70$
- ▶ Ant & Cleo:  $1 + \log_{10}(30) = 1 + 1.48 = 2.48$
- ▶ Tempest: 0

### **MERCY:**

- ▶ Julius Caesar:  $1 + \log_{10}(2) = 1 + 0.30 = 1.30$
- ▶ Ant & Cleo:  $1 + \log_{10}(5) = 1 + 0.70 = 1.70$
- ▶ Tempest:  $1 + \log_{10}(8) = 1 + 0.90 = 1.90$

## Step 2: The TF Matrix

Term	Julius Caesar	Ant & Cleo	Tempest
BRUTUS	2.60	1.70	0
CAESAR	2.70	2.48	0
MERCY	1.30	1.70	1.90

**This is the LOCAL component.**

It tells us how important each term is **within each document**.

But there's a problem: MERCY appears in **all** documents, so it's not useful for distinguishing between them!

## Step 3

Calculate Inverse Document Frequency (IDF)

Rare words are more useful for finding relevant documents

## Step 3: Why Do We Need IDF?

**Problem:** Some words appear in many documents (common), others appear in few (rare).

**Key Insight:** **Rare words are more useful** for finding relevant documents!

**In our example:**

- ▶ BRUTUS appears in **2 out of 3** documents – fairly rare
- ▶ CAESAR appears in **2 out of 3** documents – fairly rare
- ▶ MERCY appears in **3 out of 3** documents – **not useful!**

**Think about it:** If a word appears in EVERY document, it can't help us distinguish between them. It's like searching for "the" – useless!

## Step 3: Document Frequency (df)

**Document Frequency** = number of documents containing the term

Term	df	Which documents?
BRUTUS	2	Julius Caesar, Ant & Cleo
CAESAR	2	Julius Caesar, Ant & Cleo
MERCY	3	Julius Caesar, Ant & Cleo, Tempest

**Note:** We count **how many documents** contain the term, not how many times it appears total.

## Step 3: The IDF Formula

### Inverse Document Frequency Formula

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

where:

- ▶  $N$  = total number of documents (in our case,  $N = 3$ )
- ▶  $\text{df}_t$  = number of documents containing term  $t$

### What this does:

- ▶ If  $\text{df}$  is small (rare word)  $\rightarrow N/\text{df}$  is large  $\rightarrow$  IDF is large
- ▶ If  $\text{df}$  is large (common word)  $\rightarrow N/\text{df}$  is small  $\rightarrow$  IDF is small
- ▶ If  $\text{df} = N$  (word in every doc)  $\rightarrow N/\text{df} = 1 \rightarrow \text{IDF} = 0!$

## Step 3: Calculate IDF Values

$N = 3$  (we have 3 documents)

**BRUTUS:**  $df = 2$

$$\text{idf}_{\text{BRUTUS}} = \log_{10} \left( \frac{3}{2} \right) = \log_{10}(1.5) = \mathbf{0.18}$$

**CAESAR:**  $df = 2$

$$\text{idf}_{\text{CAESAR}} = \log_{10} \left( \frac{3}{2} \right) = \log_{10}(1.5) = \mathbf{0.18}$$

**MERCY:**  $df = 3$  (appears in ALL documents!)

$$\text{idf}_{\text{MERCY}} = \log_{10} \left( \frac{3}{3} \right) = \log_{10}(1) = \mathbf{0}$$

## Step 3: What IDF Values Mean

Term	df	$N/df$	IDF
BRUTUS	2	$3/2 = 1.5$	<b>0.18</b>
CAESAR	2	$3/2 = 1.5$	<b>0.18</b>
MERCY	3	$3/3 = 1.0$	<b>0</b>

### Key Insight

MERCY has  $IDF = 0$  because it appears in **every** document!

A word that appears everywhere has **zero discriminating power**.

**Rule:** Higher IDF = rarer word = more useful for finding relevant documents

## Step 4

Calculate TF-IDF Weights

Multiply TF (local) by IDF (global)

## Step 4: The TF-IDF Formula

### TF-IDF Formula

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

This is: **Local**  $\times$  **Global**

#### What this achieves:

- ▶ High weight if term is frequent in THIS document (high TF)
- ▶ High weight if term is rare ACROSS documents (high IDF)
- ▶ Low weight if term is in every document ( $\text{IDF} = 0 \rightarrow \text{weight} = 0$ )

## Step 4: Calculate TF-IDF for BRUTUS

**BRUTUS** (IDF = 0.18)

Julius Caesar:  $\text{tf-idf} = 2.60 \times 0.18 = \mathbf{0.47}$

Ant & Cleo:  $\text{tf-idf} = 1.70 \times 0.18 = \mathbf{0.31}$

Tempest:  $\text{tf-idf} = 0 \times 0.18 = \mathbf{0}$

**Note:** The Tempest gets 0 because BRUTUS doesn't appear there (TF = 0).

## Step 4: Calculate TF-IDF for CAESAR and MERCY

### **CAESAR** (IDF = 0.18)

Julius Caesar:  $2.70 \times 0.18 = 0.49$

Ant & Cleo:  $2.48 \times 0.18 = 0.45$

Tempest:  $0 \times 0.18 = 0$

### **MERCY** (IDF = 0)

Julius Caesar:  $1.30 \times 0 = 0$

Ant & Cleo:  $1.70 \times 0 = 0$

Tempest:  $1.90 \times 0 = 0$

**MERCY is 0 everywhere** because it appears in all documents (IDF = 0)!

## Step 4: The Complete TF-IDF Matrix

Term	Julius Caesar	Ant & Cleo	Tempest
BRUTUS	<b>0.47</b>	0.31	0
CAESAR	<b>0.49</b>	0.45	0
MERCY	0	0	0

**This is our TF-IDF Weight Matrix!**

### Observations:

- ▶ Julius Caesar has the highest weights for BRUTUS and CAESAR
- ▶ MERCY is useless (all zeros) – it can't help us distinguish documents
- ▶ The Tempest has all zeros for our query terms

# Step 5

## Normalise Document Vectors

Make all documents comparable regardless of length

## Step 5: Documents as Vectors

Each column of the TF-IDF matrix is a **document vector**:

**Vector format:** (BRUTUS weight, CAESAR weight, MERCY weight)

$$\vec{d}_1 \text{ (Julius Caesar)} = (0.47, 0.49, 0)$$

$$\vec{d}_2 \text{ (Ant \& Cleo)} = (0.31, 0.45, 0)$$

$$\vec{d}_3 \text{ (Tempest)} = (0, 0, 0)$$

**What is a vector?** Just an ordered list of numbers!

Each number represents the importance of that term in that document.

## Step 5: Why Normalise?

**Problem:** Longer documents have more words, so they might get higher scores unfairly.

**Solution:** Divide each vector by its **length** so all vectors have length 1.

### Vector Length Formula

$$|\vec{d}| = \sqrt{d_1^2 + d_2^2 + d_3^2 + \dots}$$

This is Pythagoras' theorem extended to multiple dimensions!

**Simple example:** If  $\vec{v} = (3, 4)$

$$|\vec{v}| = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

## Step 5: Calculate Document Lengths

**Julius Caesar:**  $\vec{d}_1 = (0.47, 0.49, 0)$

$$|\vec{d}_1| = \sqrt{0.47^2 + 0.49^2 + 0^2} = \sqrt{0.22 + 0.24 + 0} = \sqrt{0.46} = \mathbf{0.68}$$

**Anthony & Cleopatra:**  $\vec{d}_2 = (0.31, 0.45, 0)$

$$|\vec{d}_2| = \sqrt{0.31^2 + 0.45^2 + 0^2} = \sqrt{0.10 + 0.20 + 0} = \sqrt{0.30} = \mathbf{0.55}$$

**Tempest:**  $\vec{d}_3 = (0, 0, 0)$

$$|\vec{d}_3| = \sqrt{0^2 + 0^2 + 0^2} = \mathbf{0}$$

## Step 5: Normalise Each Vector

**Normalisation formula:** Divide each component by the length.

$$\hat{d} = \frac{\vec{d}}{|\vec{d}|} = \left( \frac{d_1}{|\vec{d}|}, \frac{d_2}{|\vec{d}|}, \frac{d_3}{|\vec{d}|} \right)$$

**Julius Caesar:**

$$\hat{d}_1 = \frac{(0.47, 0.49, 0)}{0.68} = \left( \frac{0.47}{0.68}, \frac{0.49}{0.68}, \frac{0}{0.68} \right) = (0.69, 0.72, 0)$$

**Verify length = 1:**

$$\sqrt{0.69^2 + 0.72^2 + 0^2} = \sqrt{0.48 + 0.52} = \sqrt{1.0} = 1 \quad \checkmark$$

## Step 5: All Normalised Document Vectors

**Before normalisation:**

$$\vec{d}_1 = (0.47, 0.49, 0) \quad |\vec{d}_1| = 0.68$$

$$\vec{d}_2 = (0.31, 0.45, 0) \quad |\vec{d}_2| = 0.55$$

$$\vec{d}_3 = (0, 0, 0) \quad |\vec{d}_3| = 0$$

**After normalisation** (divide by length):

$$\hat{d}_1 \text{ (Julius Caesar)} = (0.69, 0.72, 0)$$

$$\hat{d}_2 \text{ (Ant \& Cleo)} = (0.57, 0.82, 0)$$

$$\hat{d}_3 \text{ (Tempest)} = (0, 0, 0)$$

**Now all vectors have length 1!** (Called “unit vectors”)

# Part B

## Answering a Query

(Done every time a user searches)

## Step 6

### Create the Query Vector

Represent the query in the same way as documents

## Step 6: Query as a Vector

**Query:** “BRUTUS CAESAR”

The query becomes a vector in the **same space** as the documents.

**Simple approach:** Give weight 1 to each query term, 0 to others.

$$\vec{q} = ( \underbrace{1}_{\text{BRUTUS}}, \underbrace{1}_{\text{CAESAR}}, \underbrace{0}_{\text{MERCY}} ) = (1, 1, 0)$$

**Why this works:** We want to find documents where BRUTUS and CAESAR are important. By giving them equal weight (1), we’re saying they’re equally important in the query.

## Step 7

Normalise the Query Vector

Same process as for documents

## Step 7: Normalise the Query

**Query vector:**  $\vec{q} = (1, 1, 0)$

**Step 1: Calculate length**

$$|\vec{q}| = \sqrt{1^2 + 1^2 + 0^2} = \sqrt{1 + 1 + 0} = \sqrt{2} = \mathbf{1.41}$$

**Step 2: Divide by length**

$$\hat{q} = \frac{(1, 1, 0)}{1.41} = \left( \frac{1}{1.41}, \frac{1}{1.41}, \frac{0}{1.41} \right) = \mathbf{(0.71, 0.71, 0)}$$

**Verify:**  $\sqrt{0.71^2 + 0.71^2 + 0^2} = \sqrt{0.50 + 0.50} = 1 \checkmark$

# Step 8

## Compute Dot Products

For normalised vectors, dot product = similarity!

## Step 8: What is a Dot Product?

The **dot product** multiplies corresponding elements and adds them up.

### Dot Product Formula

$$\vec{a} \cdot \vec{b} = a_1 \times b_1 + a_2 \times b_2 + a_3 \times b_3$$

**Example:**

$$(2, 3, 1) \cdot (4, 0, 2) = (2 \times 4) + (3 \times 0) + (1 \times 2) = 8 + 0 + 2 = 10$$

### Key Insight

For **unit vectors** (length = 1), the dot product equals the **cosine similarity**!

This means: **dot product = how similar the vectors are**

## Step 8: Calculate Dot Products

**Normalised query:**  $\hat{q} = (0.71, 0.71, 0)$

**Julius Caesar:**  $\hat{d}_1 = (0.69, 0.72, 0)$

$$\begin{aligned}\hat{q} \cdot \hat{d}_1 &= (0.71 \times 0.69) + (0.71 \times 0.72) + (0 \times 0) \\ &= 0.49 + 0.51 + 0 = \mathbf{1.00}\end{aligned}$$

**Ant & Cleo:**  $\hat{d}_2 = (0.57, 0.82, 0)$

$$\begin{aligned}\hat{q} \cdot \hat{d}_2 &= (0.71 \times 0.57) + (0.71 \times 0.82) + (0 \times 0) \\ &= 0.40 + 0.58 + 0 = \mathbf{0.99}\end{aligned}$$

**Tempest:**  $\hat{d}_3 = (0, 0, 0)$

$$\hat{q} \cdot \hat{d}_3 = 0 + 0 + 0 = \mathbf{0}$$

## Step 9

Rank the Documents

Highest similarity wins!

## Step 9: Final Ranking

**Query:** “BRUTUS CAESAR”

Rank	Document	Similarity Score
1	Julius Caesar	1.00
2	Anthony & Cleopatra	0.99
3	The Tempest	0.00

**Julius Caesar wins!** This makes sense because:

- ▶ It has the highest TF-IDF weights for both BRUTUS and CAESAR
- ▶ The play is literally about these characters!

The Tempest scores 0 because it contains neither query term.

# Summary

Everything in One Place

# Why Pre-Normalisation is Smart

## The old way:

At query time, calculate:

$$\text{similarity} = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}| \times |\vec{d}|}$$

Problem: Must calculate  $|\vec{d}|$  for every document, every time!

## The smart way:

Pre-normalise documents once:

$$\hat{d} = \frac{\vec{d}}{|\vec{d}|}$$

Then at query time:

$$\text{similarity} = \hat{q} \cdot \hat{d}$$

Just multiply and add – no square roots!

## Why It Works

$$\frac{\vec{q} \cdot \vec{d}}{|\vec{q}| \times |\vec{d}|} = \frac{\vec{q}}{|\vec{q}|} \cdot \frac{\vec{d}}{|\vec{d}|} = \hat{q} \cdot \hat{d}$$

Same answer, much faster!

# The Complete Process: Summary

Step	Name	What You Do
1	Raw Counts	Count term occurrences in each document
2	TF	$tf = 1 + \log_{10}(\text{count})$
3	IDF	$idf = \log_{10}(N/df)$
4	TF-IDF	$\text{weight} = tf \times idf$
5	Normalise Docs	$\hat{d} = \vec{d}/ \vec{d} $
— <i>Index built. Now answer queries:</i> —		
6	Query Vector	Weight 1 for each query term
7	Normalise Query	$\hat{q} = \vec{q}/ \vec{q} $
8	Dot Products	$\text{score} = \hat{q} \cdot \hat{d}$ for each document
9	Rank	Sort documents by score (highest first)

**1. Term Frequency (Local importance):**

$$\text{tf}_{t,d} = 1 + \log_{10}(\text{count}) \quad \text{or } 0 \text{ if count} = 0$$

**2. Inverse Document Frequency (Global importance):**

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

**3. TF-IDF Weight:**

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

**4. Vector Length:**

$$|\vec{d}| = \sqrt{\sum_i d_i^2}$$

**5. Similarity (for normalised vectors):**

$$\text{similarity} = \hat{q} \cdot \hat{d} = \sum_i \hat{q}_i \times \hat{d}_i$$

## We worked through:

1. **TF-IDF weighting** — term frequency  $\times$  inverse document frequency
2. **Cosine similarity** via the dot product of normalised vectors
3. **Document length normalisation** — so long and short documents are compared fairly

$$\text{Weight} = \text{Local (TF)} \times \text{Global (IDF)} \times \text{Length Normalisation}$$

We used three Shakespeare plays and the query “BRUTUS CAESAR” to demonstrate the complete pipeline: count terms  $\rightarrow$  compute TF-IDF weights  $\rightarrow$  normalise  $\rightarrow$  dot product  $\rightarrow$  rank.

## Key Result

“BRUTUS” and “CAESAR” have high IDF because they are rare across documents. Common words like “the” contribute nothing. The dot product naturally ranks *Julius Caesar* highest.

# What TF-IDF Cosine Similarity Actually Does

**TF-IDF cosine similarity is exact lexical matching.**

It works when the **same words** appear in both texts:

Recommendation: “*The PHSO recommended **compensation** for **women**...*”

Response: “*We have decided against **compensation** for **women**...*”

→ Shared terms “compensation” and “women” produce a **high cosine similarity score**. ✓

It **fails** when different words express the same meaning:

Recommendation: “*financial **remedy** for affected **individuals***”

Response: “*monetary **payment** to those **impacted***”

→ TF-IDF sees **zero overlap** on the key terms. Cosine similarity will be **low**. ✕

## The Limitation

TF-IDF matches **words**, not **meaning**. It cannot handle synonyms, paraphrasing, or rephrased content.

**If better methods exist, why learn TF-IDF cosine similarity?**

- 1. It's the foundation of all text similarity.** Vector representation, term weighting, normalisation, and cosine similarity are the building blocks. Every advanced method — LSI, sentence embeddings, RAG — builds on these concepts.
- 2. It combines with machine learning.** In the sentiment analysis lab, you used bag-of-words vectors as input features for a neural network. TF-IDF representations are the foundation for that kind of pipeline — represent text as a vector, then use it downstream.
- 3. The principles underpin production systems everywhere.** Search engines, spam filters, plagiarism detection, recommendation systems — all use term weighting and IDF concepts, often combined with other techniques like BM25 or learning-to-rank models.
- 4. It works when pre-trained models aren't available.** Specialised domains (legal, medical, classified), low-resource languages, or constrained environments — TF-IDF cosine works out of the box with no external dependencies.

# What's Coming Next

TF-IDF cosine similarity is the **starting point**. We will cover two techniques that go beyond exact word matching:

## Next Week: Latent Semantic Indexing (LSI)

- ▶ Applies SVD to the TF-IDF matrix to discover **hidden semantic dimensions**
- ▶ “remedy” and “payment” collapse onto the same latent concept
- ▶ Addresses the synonym problem **without** needing any pre-trained model
- ▶ Still uses cosine similarity to compare the resulting vectors

## Later: Sentence Embeddings

- ▶ Encode entire passages as **dense vectors that capture meaning**
- ▶ Handle synonyms, paraphrasing, and rephrasing naturally
- ▶ Pre-trained models (e.g. sentence-transformers) available off the shelf
- ▶ This is how modern **RAG** (Retrieval-Augmented Generation) systems work
- ▶ Still uses cosine similarity to compare vectors

**Common thread:** all three approaches produce vectors and compare them with **cosine similarity**. Only how the vectors are built changes.

## Your Task

Given six pairs of UK Government documents, build a system that can **search**, **extract**, and **align** recommendations with their responses.

Task	Marks	What You Can Use
Task 1: Search	15	TF-IDF cosine (keyword search across all 12 PDFs) <b>or</b> sentence embeddings (semantic search, RAG-style)
Task 2a: Extraction	35	Pattern matching and structural cues: modal verbs ("should", "must", "recommend"), numbered headings, sentence position
Task 2b: Alignment	25	TF-IDF cosine works for <b>exact word overlap</b> only. For paraphrased content, use LSI or sentence embeddings
Task 2c: Evaluation	5	Compare outputs against ground truth; analyse where and <b>why</b> errors occur

**Key insight:** TF-IDF cosine similarity is a strong baseline for **search** (Task 1), but for **alignment** (Task 2b) you will likely need LSI or **sentence embeddings** to handle cases where recommendations and responses use different vocabulary.

## What we covered:

- ▶ TF-IDF weighting, cosine similarity, document length normalisation
- ▶ These are **exact lexical matching** — same words → high similarity

## Why it matters:

- ▶ Foundation for all text similarity techniques
- ▶ Combines with machine learning (as in the BoW sentiment analysis lab)
- ▶ Principles underpin production systems; works without pre-trained models

## What's next:

- ▶ **LSI** (next week) — handles synonyms via latent semantic dimensions
- ▶ **Sentence embeddings** (later) — captures meaning, powers RAG systems

$$\text{similarity} = \hat{q} \cdot \hat{d} = \sum_i \hat{q}_i \times \hat{d}_i$$

Whether vectors come from TF-IDF, LSI, or sentence embeddings — the comparison step is always the same cosine similarity.

# Questions?

Prof. Georgina Cosma

`g.cosma@lboro.ac.uk`