Part 4

LSI for Information Retrieval

Why Use LSI for Information Retrieval?

LSI takes documents that are semantically similar (talk about the same topics)...

1. ...but are **not similar in the vector space** (they use different words)...
2. ...and re-represents them in a **reduced vector space**...
3. ...in which they have **higher similarity**.

Standard Vector Space

Synonyms contribute **nothing** to document similarity.

Desired Effect of LSI

Synonyms contribute **strongly** to document similarity.

Thus, LSI addresses the problems of synonymy and semantic relatedness.

LSI is able to do this because the mathematics computes the co-occurrences of terms in documents — documents can be identified as similar by analysing the co-occurrences of their terms.

Implementing LSI for Retrieval

Offline (indexing phase):

1. Compute SVD: $C = U\Sigma V^T$
2. Choose k (number of dimensions to retain)
3. Compute reduced matrices: U_k, Σ_k, V_k^T
4. Store U_k, Σ_k, V_k^T for query-time use

Online (query time):

1. Represent query q as a term vector
2. Project query into reduced space:

$$q_k^T = q^T U_k \Sigma_k^{-1}$$

3. Compute similarity of q_k with all reduced document vectors in V_k^T
4. Rank documents by similarity
5. Return top-N documents

Derivation of the Query Projection Formula

From $C_k = U_k \Sigma_k V_k^T$ it follows that:

$$\Sigma_k^{-1} U_k^T C = V_k^T$$

Applying the same transformation to query vector $\bar{q}$ gives:

$$\bar{q}_k^T = \Sigma_k^{-1} U_k^T \bar{q}^T$$

This places the query in the same normalised semantic space as the document vectors in V_k^T.

Query Example

Query: "person jumps over hedge"

Step 1: Represent query as a term vector

Term	q
person	1
jumps	1
over	1
fox	0
owl	0
hedge	1

Step 2: Project into reduced space using $q_k^T = q^T U_k \Sigma_k^{-1}$

Step 3: Compute cosine similarity between q_k and all document vectors in V_k^T

Step 4: Rank and return documents

Key Point

Documents that contain synonyms of the query terms (e.g. "leaps" instead of "jumps") will now score higher than in a standard vector space model, because synonyms are mapped to nearby positions in the reduced semantic space.

Why Cosine Similarity for LSI Retrieval?

In Step 3 we ranked documents using cosine similarity — but why not just use dot product or Euclidean distance?

What cosine similarity measures

For two vectors **a** and **b**:

$$\text{sim}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|} = \cos\theta$$

It measures the **angle** between vectors, not their length.

Values range from -1 (opposite directions) through 0 (orthogonal, no relation) to $+1$ (identical direction, maximum similarity).

Concrete example

Document d_A: mentions "ship" once $\rightarrow$ short vector
Document d_B: mentions "ship" ten times $\rightarrow$ long vector

Both point in the *same direction* in semantic space.
Cosine $= 1.0$ for both. **Length does not unfairly inflate the score.**

Why not Euclidean distance?

Euclidean distance penalises long documents simply because their vectors extend further from the origin — a document mentioning "ship" 20 times would be ranked *further* from "ocean" than one mentioning it once, which is wrong.

In the reduced LSI space:

After projecting the query and all documents into k dimensions, we have dense, real-valued vectors. The cosine of the angle between the query vector q_k and each document vector d_k gives a score in $[-1, +1]$ that reflects semantic alignment, regardless of document length.

Note

How many top documents to return, and how to evaluate retrieval quality using recall and precision, will be covered in a later lecture.

Graphical Illustration of LSI: The Deerwester Corpus

The canonical LSI example from Deerwester et al. (1990):

c_1	Human machine interface for lab abc computer applications
c_2	A survey of user opinion of computer system response time
c_3	The EPS user interface management system
c_4	System and human system engineering testing of EPS
c_5	Relation of user perceived response time to error measurement
m_1	The generation of random binary unordered trees
m_2	The intersection graph of paths in trees
m_3	Graph minors IV widths of trees and well quasi ordering
m_4	Graph minors: a survey

Two document groups

c_1–c_5: Human-computer interaction
(human, interface, system, EPS, user, response)

m_1–m_4: Graph theory & maths
(trees, graph, minors)

The matrix C (term $\times$ doc):

	c_1	c_2	c_3	c_4	c_5	m_1	m_2	m_3	m_4
human	1	0	0	1	0	0	0	0	0
interface	1	0	1	0	0	0	0	0	0
computer	1	1	0	0	0	0	0	0	0
user	0	1	1	0	1	0	0	0	0
system	0	1	1	2	0	0	0	0	0
response	0	1	0	0	1	0	0	0	0
time	0	1	0	0	1	0	0	0	0
EPS	0	0	1	1	0	0	0	0	0
survey	0	1	0	0	0	0	0	0	1
trees	0	0	0	0	0	1	1	1	0
graph	0	0	0	0	0	0	1	1	1
minors	0	0	0	0	0	0	0	1	1

The test query

$q =$ "human computer interaction"

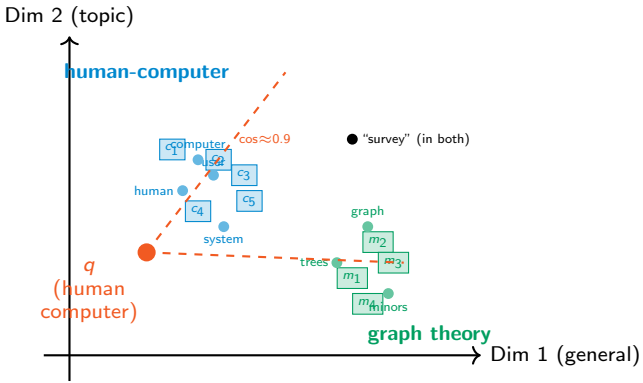
Challenge: c_3 and c_5 share *no* terms with q, yet are about the same topic.

LSI should retrieve them.

After reducing to $k = 2$, the documents are plotted in a 2D semantic space. The next slide shows the result.

Graphical Illustration of LSI: The 2D Plot

After reducing to $k = 2$, every document/term occupies a point in semantic space:



Reading the plot

Squares = documents
Circles = terms

The dotted cone = region within cosine 0.9 of query q.

All c_1–c_5 are near q, even c_3/c_5 which share **no terms** with q.

None of m_1–m_4 are near q.

Key result

LSI retrieves thematically related documents even without exact term overlap. "survey" appears near both clusters — it occurs in both c_2 and m_4.

The Computational Challenge

Fundamental problem: LSI requires computing similarity between the query and **all documents**.

Three Practical Issues

1. **Expensive at query time:** Must compute $\mathrm{sim}(q_k, d_k)$ for every document — no efficient indexing structure (unlike an inverted index).
2. **Display threshold:** How many documents do you show? You must define a rank cutoff or similarity threshold. Regardless, LSI must still compare against every document.
3. **Expensive at index time:** SVD is $O(mn^2)$ for m terms, n documents. Difficult to update incrementally — adding new documents requires recomputing the SVD.

Practical solutions:

- **Hybrid approach:** Use an inverted index for initial retrieval, then LSI for re-ranking the top-N results
- **Clustering:** Organise documents into clusters at index time; at query time search only the nearest cluster (next slide)
- **Modern alternative:** Dense retrieval with learned embeddings (BERT) + approximate nearest-neighbour search (FAISS, HNSW)

Using Clustering to Speed Up Retrieval

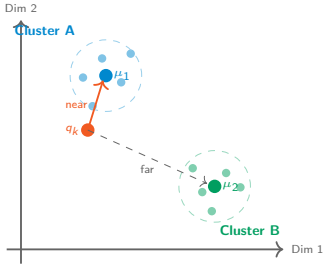
Core idea: instead of comparing the query to *every* document, pre-group documents into clusters. At query time, find the nearest cluster first, then only search within it.

Index-time (done once, offline):

1. Compute LSI reduced vectors for all documents (V_k^T)
2. Cluster the document vectors — e.g. k-means into C clusters
3. Compute and store the **centroid** of each cluster: $\mu_1, \mu_2, \ldots, \mu_C$

Query time:

1. Project query into the reduced space: q_k
2. Compute $\mathrm{sim}(q_k, \mu_j)$ for each of the C centroids ($C \ll n$)
3. Identify the nearest cluster(s)
4. Compute $\mathrm{sim}(q_k, d_k)$ **only for documents in those clusters**
5. Rank and return results



Query q_k is close to μ_1 (Cluster A). Only documents in Cluster A need their full cosine similarity computed. Cluster B is skipped entirely.

Trade-off

Speed gain: similarity computed against $\sim n/C$ documents instead of all n. **Cost:** some relevant documents near a cluster boundary may be missed. This is an *approximate* method.

LSI as Soft Clustering

The full decomposition reveals cluster structure:

U	1	**2**	3	4	5
ship	−0.44	−0.30	0.57	0.58	0.25
boat	−0.13	−0.33	−0.59	0.00	0.73
ocean	−0.48	−0.51	−0.37	0.00	−0.61
wood	−0.70	+0.35	0.15	−0.58	0.16
tree	−0.26	+0.65	−0.41	0.58	−0.09

Σ	1	2
1	2.16	0.00
2	0.00	1.59

V^T	d_1	d_2	d_3	d_4	d_5	d_6
1	−0.75	−0.28	−0.20	−0.45	−0.33	−0.12
2	−0.29	−0.53	−0.19	0.63	0.22	0.41

Water topic (Dim 2 < 0) Land topic (Dim 2 > 0)

Dimension 2: "soft membership"

Term/Doc	Value	Label
Terms:		
ocean	−0.51	Water (strong)
boat	−0.33	Water
ship	−0.30	Water
wood	+0.35	Land
tree	+0.65	Land (strong)
Documents:		
d_2 (boat, ocean)	−0.53	Water
d_1 (ship, ocean, **wood**)	−0.29	Mixed!
d_4 (wood, tree)	+0.63	Land

d_1 contains "wood" alongside water terms — its *intermediate* value reflects this mixed content. SVD discovered these groups **with no prior knowledge of word meanings**.

Connection to clustering: the sign and magnitude of each document's value on Dimension 2 directly identifies which cluster it belongs to — LSI provides cluster membership *as a by-product of the decomposition*, without running a separate clustering algorithm.

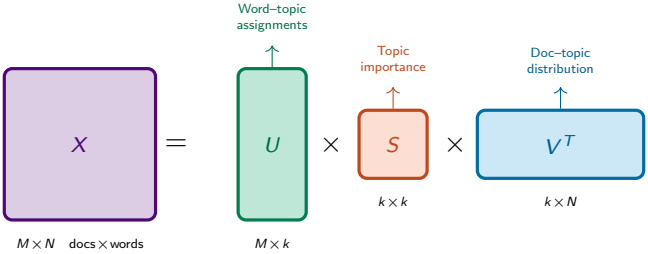
LSI is a **decomposition of** C **into**: a representation of the terms, a representation of the documents, and a representation of the importance of the "semantic" dimensions.

Part 5

From LSI to Modern Embeddings

LSI for Topic Modelling

While LSI is also used for **topic modelling**, SVD reveals three complementary views of the data:



U — Term-to-topic

Each row is a term.
Each column is a latent topic.
High values = that term is important in that topic.

S — Topic strength

Diagonal values rank topics by how much variance they explain.
Largest = most dominant topic in the corpus.

V^T — Doc-to-topic

Each column is a document.
Each row is a topic.
Values = soft membership of each document in each topic.

More sophisticated probabilistic models (LDA) build on this foundation, adding probability interpretations.

LSA vs LDA

	LSA (Latent Semantic Analysis)	LDA (Latent Dirichlet Allocation)
Method	Matrix factorisation (SVD)	Probabilistic generative model
Input	Term–document matrix (BoW or TF-IDF)	Bag-of-words (document–term counts)
Output	Dense real-valued vectors	Probability distributions
Topics	Linear combinations of terms	Probability distributions over terms
Documents	Linear combinations of topics	Probability distributions over topics
Interpretability	Harder (values can be negative)	Easier (probabilities sum to 1)
Main use	Dimensionality reduction, retrieval	Topic discovery and modelling

LSA applies linear algebra to uncover latent structure in a word–document matrix. It *reduces* matrix dimensions.

LDA models documents as mixtures of latent topics with probabilistic semantics. It *generates* topic–word distributions.

This is How Embeddings Are Created

Core Message

LSI/LSA shows you how word and document embeddings can be created from scratch when no pretrained model is available. The resulting vectors *are* the embeddings.

- For large datasets, **pretrained models** (Word2Vec, GloVe, BERT) do a better job at capturing semantic meaning because they are trained on massive corpora.
- However, some applications **may not have a suitable pretrained model** — for example, a domain-specific vocabulary (legal, medical), or a specific dialect (e.g. Cypriot Greek) where no pretrained model exists.
- In those cases, LSA-style approaches — building a term-document matrix from your own corpus and applying SVD — give you **domain-specific embeddings** with full transparency over every step.
- Word2Vec is an *algorithm* for generating word embeddings, not just a pretrained model. Pretrained Word2Vec models exist (e.g. via Gensim) but training from scratch on your own corpus is entirely viable for specialised domains.

The maths behind LSA are still used in NLP and information retrieval today.

Word2Vec: Algorithm vs Pretrained Model

Word2Vec itself is an algorithm for generating word embeddings — not just a pretrained model.

Pretrained Word2Vec Models

- Models trained on large corpora are available for direct use
- Useful for tasks like word similarity, word analogy, and semantic search without training from scratch
- Available via **Gensim** in Python: captures semantic relationships between words

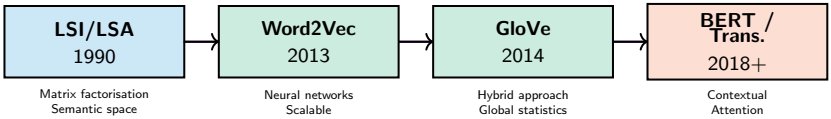
Training Your Own

- For specialised domains (medical, legal, specific dialects), pretrained models may not capture the right vocabulary
- You can train Word2Vec on your own corpus — just as you can build an LSA model on your own term-document matrix
- LSA gives you **full transparency**; Word2Vec gives you **better scalability**

- The **maths behind LSA** underpin Word2Vec: both exploit co-occurrence structure to build semantic spaces
- Large Language Models (GPT, BERT) are pre-trained on massive datasets and do a better job at capturing semantic meaning — but they are not always available for every domain or dialect

The Evolution of Semantic Embeddings

Common principle: Compress text into a semantic space where similar meanings are close



Key innovation at each stage:

- **Word2Vec (2013):** replaced matrix algebra with a small neural network; much faster to train on large corpora; learns from local context windows around each word
- **GloVe (2014):** combined Word2Vec's speed with LSA's use of *global* co-occurrence statistics across the whole corpus simultaneously
- **BERT / Transformers (2018+):** added *context-sensitivity* — the same word gets a *different* embedding depending on the sentence it is in ("bank" the river vs. "bank" the institution)

Word2Vec, GloVe and BERT: Pretrained or Train Your Own?

All four methods are algorithms — not just files you download. The difference is how much compute you need to train one yourself.

Word2Vec and GloVe — easy to train yourself

Both are open algorithms. You give them a text corpus and they produce word vectors. No GPU required.

- **Word2Vec:** one line in Python via Gensim: `Word2Vec(sentences)`. Trains in minutes on a domain corpus.
- **GloVe:** Stanford provides the C source code. Point it at your corpus, run it, get vectors.

Pretrained versions (Google News, Wikipedia) are available if you just want to download and use them, but they are optional.

BERT / Transformers — pretrained in practice

BERT is also an algorithm, but training it from scratch is expensive:

- Original training: 4 days on 64 TPUs, on billions of words
- For most people this is not feasible

In practice you always start from a pretrained checkpoint, then **fine-tune** on your domain data for a few hours on a single GPU.
Domain-specific BERTs (BioBERT for medicine, LegalBERT for law) were made exactly this way.

Key point for this course

LSI and Word2Vec are the most accessible: you can run them yourself on any text, inspect every number, and understand exactly what happened. BERT is powerful but a black box by comparison.

Connecting LSI Concepts to Modern Methods

LSI Concept	Modern Equivalent	How It Evolved
Semantic dimensions in U	Embedding dimensions	Now learned end-to-end, not via SVD
Dimensionality reduction via Σ	Attention mechanisms	Transformers learn which dimensions to emphasise
Document vectors in V^T	Document embeddings	Now computed via pooling of contextual embeddings
Synonyms map to similar positions	Distributional similarity	"You shall know a word by the company it keeps" (Firth, 1957)
Query projection into reduced space	Dense retrieval	Queries and documents embedded in the same space

The core insight remains constant: compress high-dimensional sparse representations into low-dimensional dense representations. In the compressed space, semantic similarity is explicit.

Modern Dense Retrieval

How modern systems address LSI's scalability limitations:

Dense retrieval with learned embeddings:

- Encode documents with a Transformer encoder (e.g. BERT): $d_i \rightarrow \mathbf{e}_i \in \mathbb{R}^{768}$
- Encode the query in the same space: $q \rightarrow \mathbf{e}_q \in \mathbb{R}^{768}$
- Retrieve via **Approximate Nearest Neighbour (ANN)** search (FAISS, HNSW, ScaNN)

Key differences from LSI:

Scalable	ANN enables sub-linear search over billions of vectors
Contextual	Transformer embeddings capture word meaning in context
Task-optimised	Models can be fine-tuned for specific retrieval objectives
Incremental	New documents can be added without recomputing the whole space

Key Takeaway

The core idea — mapping text into a shared semantic vector space and retrieving via similarity — originates with LSI. Modern systems inherit this principle while addressing its limitations.

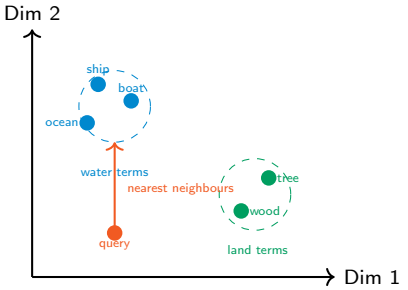
An embedding is a vector of numbers that represents meaning. The closer two vectors are in that space, the more semantically related the texts.

What LSI gives you

- Each row of V^T is a **document embedding** in the k-dimensional semantic space
- Each row of U is a **word embedding** in the same space
- Compute cosine similarity to find related documents

What modern models give you

- **Same principle** — vectors in a shared semantic space
- Trained on orders of magnitude more data
- Context-sensitive (BERT, Sentence-BERT)
- Off-the-shelf, no manual corpus needed



Cosine similarity in this 2D space retrieves semantically related documents even when no words are shared.

Python: LSI Embeddings with scikit-learn

Building LSI document embeddings on the ship/boat/ocean example:

Python | LSI with TruncatedSVD

```python
# pip install scikit-learn numpy
import numpy as np
from sklearn.decomposition import TruncatedSVD
from sklearn.metrics.pairwise import cosine_similarity

# Step 1: term-document matrix C (rows=terms, cols=docs)
#         terms: ship, boat, ocean, wood, tree
C = np.array([
    [1, 0, 1, 0, 0, 0],   # ship
    [0, 1, 0, 0, 0, 0],   # boat
    [1, 1, 0, 0, 0, 0],   # ocean
    [1, 0, 0, 1, 1, 0],   # wood
    [0, 0, 0, 1, 0, 1],   # tree
])

# Step 2: reduce to k=2 semantic dimensions
svd = TruncatedSVD(n_components=2, random_state=42)
C2  = svd.fit_transform(C.T)  # shape: (n_docs, k)

# Step 3: cosine similarity between d2 (boat/ocean) and d3 (ship)
sim_original = cosine_similarity(C.T[[1]], C.T[[2]])[0,0]
sim_lsi      = cosine_similarity(C2[[1]], C2[[2]])[0,0]

print(f"Original space:  d2· d3 = {sim_original:.3f}")  # 0.000
print(f"LSI space (k=2): d2· d3 = {sim_lsi:.3f}")       # ¯0.937
```

Key API: `TruncatedSVD(n_components=k)` performs the truncated SVD efficiently using randomised algorithms — no full decomposition required.

`fit_transform(C.T)` returns document embeddings directly — equivalent to $V_k\Sigma_k$ in our notation.

The same retrieval task using a pretrained BERT-based model:

Python | Sentence-BERT

```python
# pip install sentence-transformers
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity

model = SentenceTransformer('all-MiniLM-L6-v2')

# Documents and a query | full sentences, not just terms
documents = [
    "The ship sailed across the ocean carrying cargo",     # d1
    "A boat drifted across the calm ocean water",          # d2
    "The old wooden ship navigated through rough seas",    # d3
    "The carpenter cut wood and planted a tree",           # d4
    "Timber from forest trees was used for furniture",     # d5
]
query = "vessel travelling over water"  # no shared words with any doc!

# Embed everything into the SAME semantic space
doc_embeddings  = model.encode(documents)   # shape: (5, 384)
query_embedding = model.encode([query])     # shape: (1, 384)

# Rank by cosine similarity
scores = cosine_similarity(query_embedding, doc_embeddings)[0]
ranked = sorted(enumerate(scores), key=lambda x: -x[1])
for idx, score in ranked:
    print(f"  d{idx+1} ({score:.3f}): {documents[idx][:50]}...")
```

Connection to LSI: same idea — embed query and documents into a shared semantic space, rank by cosine similarity. The difference is *how* the space is learned.

Output: d2, d1, d3 rank highest — "vessel" and "water" match thematically even though neither word appears in those documents.

Part 6

Summary and Key Takeaways

Key Concepts to Remember

Mathematical foundations:

1. SVD decomposes C into $U\Sigma V^T$
   - U = terms in semantic space
   - Σ = importance of each dimension
   - V^T = documents in semantic space
2. Dimensionality reduction improves semantic matching
   - Keeping top-k dimensions removes noise
   - Forces synonyms to share dimensions
   - Eckart-Young guarantees optimality
3. LSI resolves the synonymy problem
   - Semantic relationships emerge from co-occurrence
   - No manual synonym dictionary needed

Practical implications:

1. Semantic spaces work
   - Continuous vector representations are powerful
   - Cosine similarity captures semantic relationships
2. Compression reveals structure
   - High-dimensional sparse vectors contain noise
   - Lossy compression can *improve* performance
3. Co-occurrence encodes meaning
   - Distributional hypothesis (Firth, 1957)
   - Foundation of all embedding approaches
4. LSI has real limitations
   - Scalability, no word order, static representations
   - Neural methods address these directly

Resources for Further Study

Original papers:

- Deerwester et al. (1990): "Indexing by Latent Semantic Analysis"
  https://doi.org/10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9
- Hofmann (1999): "Probabilistic Latent Semantic Indexing"
  https://doi.org/10.1145/312624.312649

Practical implementations:

- **Gensim** Python library: includes a full LSI/LSA implementation (`gensim.models.LsiModel`)
- **scikit-learn**: `TruncatedSVD` for LSI-style dimensionality reduction
- **Word space** tool: LSI applied to words (distributional semantics), Elastic Search () Elasticsearch is an open source search and analytics engine based on the Apache Lucene library. Developers can use Elasticsearch to add extremely scalable search capabilities to their applications.
- ****Elastic Search**** – New ways to use Elastic Search:
  https://github.com/elastic/elasticsearch-labs?tab=readme-ov-file

Why does LSI still matter in 2025?

LSI was published in 1990. BERT, GPT, and sentence-transformers have far surpassed it in performance. And yet every modern embedding system rests on ideas LSI introduced first.

What LSI gave us

- The idea of a **semantic vector space** where similar words sit close together
- Proof that **compression can improve retrieval** — less is sometimes more
- **Unsupervised** discovery of meaning from raw co-occurrence, with no labels
- A fully **transparent, step-by-step** process you can verify by hand

What came next (and why)

- **Word2Vec / GloVe** — same idea, scales to billions of words
- **BERT** — same idea, but the vector changes depending on context ("bank" the river vs. "bank" the institution)
- **Dense retrieval (FAISS)** — same idea, but fast enough for production

The mathematics changed. The goal — finding meaning in text — never did.

Questions?

Prof. Georgina Cosma

g.cosma@lboro.ac.uk

Department of Computer Science, Loughborough University