Universität Basel

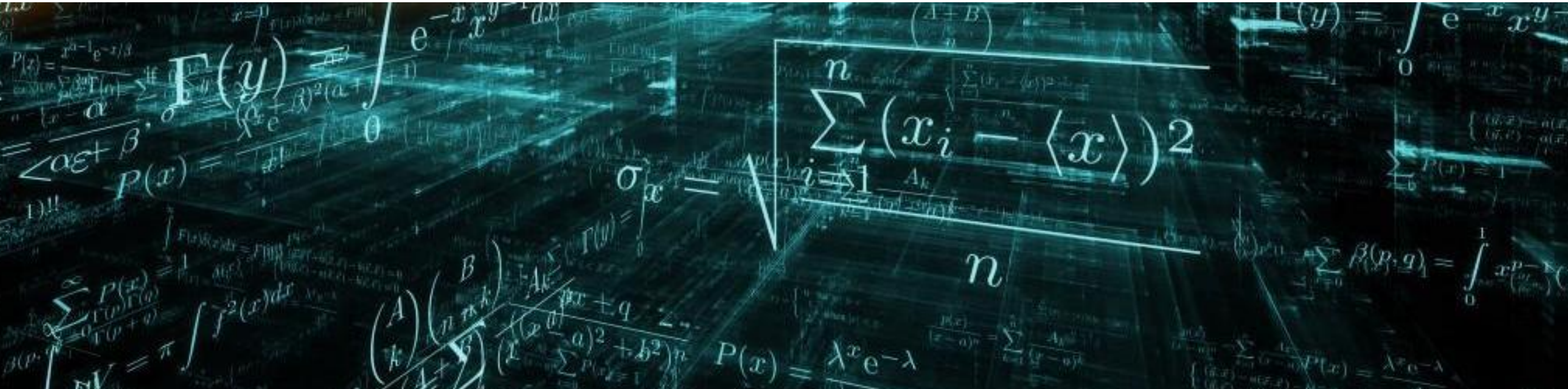
Computer Science / 15731-01 / 2025

# Multimedia Retrieval

## Chapter 6: Semantic Search

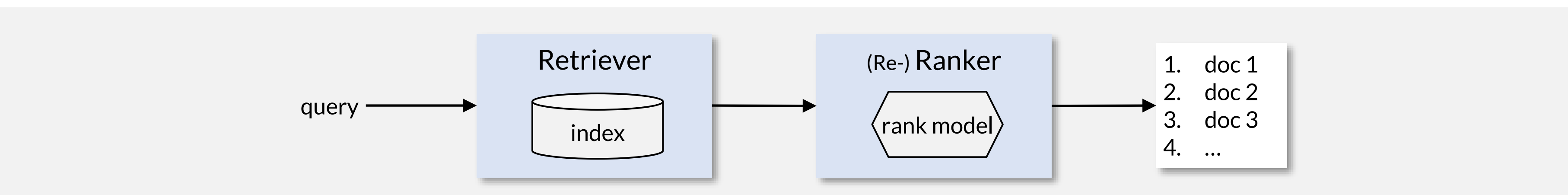


Dr. Roger Weber, roger.weber@gmail.com

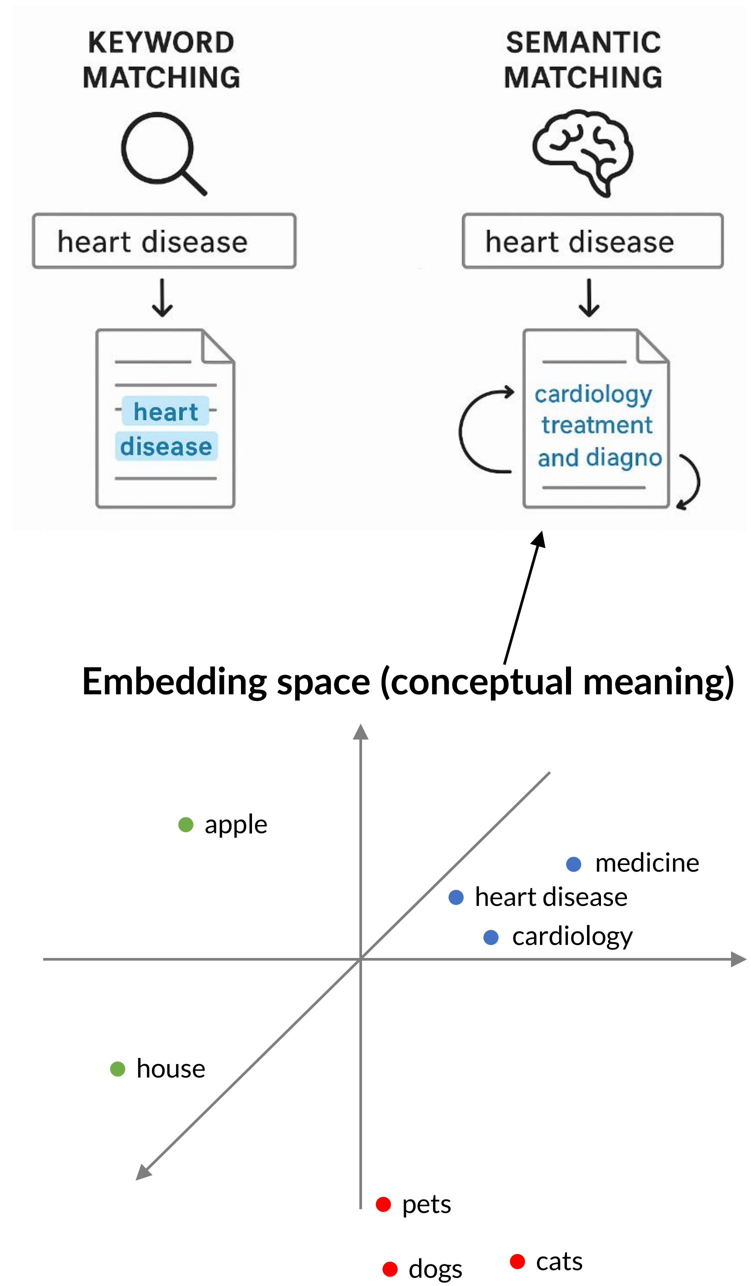


# 6.1 Introduction

- Over the early decades of information retrieval, systems relied primarily on keyword matching and statistical models such as the vector space model or BM25. These methods assumed independence among terms and treated queries as simple collections of words with only these words impacting the ranking. While efficient and surprisingly effective, such approaches offered only a shallow understanding of language: they could match keywords but not meaning.
- Over the past twenty years, two separate research directions have led to major advances in semantic understanding for information retrieval:
  - **Semantic relationships between terms and texts**, and
  - **User intent understanding** and (automated) **generation of direct responses** to queries
- This chapter is one half of a complementary pair. It focuses on **what** words and documents mean and on how to model their semantic relationships, for instance whether a document matches a query. Chapter 8 builds on this foundation by examining **why** users issue queries, users' intent, and how systems can respond with the most relevant and useful answers.
- This chapter looks at how semantic understanding in information retrieval has evolved and how models now represent and relate terms and texts beyond exact word overlap. We already covered linguistic preprocessing such as stemming and synonym expansion. Next, we examine mathematical methods like Latent Semantic Indexing (LSI) and modern transformer based models that improve how documents are retrieved and ranked for a query. We consider not only **semantic matching in the retriever phase** but also **how to better rank** and order candidate documents by how well they fit the query and the user's intent. These advances changed retrieval pipelines: semantic retrievers maintain efficiency while increasing recall, while semantic rankers use deeper contextual understanding to refine search results and thus improve precision.



- **Keyword vs. Semantic Matching:** let us consider the example query **"heart disease"**
  – With **exact keyword matching**, the search engine can only retrieve documents that contain the keywords **"heart"** and **"disease"** exactly as they appear. This approach is limited because it cannot account for variations in wording or terminology.
  – **Stemming and linguistic transformations** can improve results by allowing the system to recognize variations such as **"diseases"** or **"hearts"**, capturing slightly broader matches while still relying on surface-level text.
  – **Manually developed ontologies** can further enhance retrieval by expanding the query to include related terms, such as **"cardiovascular disorder"** or **"coronary artery disease"**. This allows the search to capture documents that are conceptually related, even if they do not contain the exact query keywords.
  – **Semantic matching**, on the other hand, goes beyond surface forms and manually curated synonyms. It captures the **conceptual meaning** of the query by mapping words and phrases to **dense vector representations** in a continuous space. These vector embeddings encode semantic relationships, so that terms like **"myocardial infarction"**, **"coronary artery conditions"**, or **"cariology treatment"** are recognized as related to **"heart disease"**, even if they share no words in common.
- Unlike keyword or ontology-based approaches, **semantic matching can automatically learn relationships** between concepts from large corpora, improving both **recall** (finding more relevant documents) and **precision** (finding documents that are truly relevant). This makes it particularly effective in domains where the same idea can be expressed in many different ways.



- An early form of concept embedding was **Latent Semantic Indexing (LSI)** in 1988, which used Singular Value Decomposition (SVD) to analyze the term-document matrix and reveal hidden links between words and concepts. By examining which words occur together, LSI uncovered underlying semantic structure. This let it connect related ideas, for example matching documents about "**heart disease**" to queries about "**cardiology**" even when keywords did not match exactly. LSI could also find cross-language relationships, so a query in one language could return relevant documents in another. However, LSI's high computational cost from the expensive SVD operation and restrictive patent coverage limited its widespread adoption and further development.
- In the early 2010s, **word2vec** popularized word embeddings based on local context and changed how people approached language modeling. It showed that words appearing in similar contexts can be represented as vectors in a high-dimensional space, where semantic similarity matches geometric closeness. Its simple neural network architecture also scaled well to learn semantic relationships from large document collections.
- Despite these advantages, word2vec's reliance on **local context** causes problems for real-world language understanding. Take the word "bank". Depending on context, it can mean a financial institution or a riverbank. For example: "I was going to the bank on Park Street. I sat there and watched the boats go by". Humans easily infer that the second sentence makes the first refer to the riverbank. Word2vec cannot do this because it models only relationships within a limited window of nearby words, and never cross sentence boundaries.
- The 2017 paper "**Attention Is All You Need**" introduced transformers and changed how models represent meaning. Its attention mechanism let models find relationships among all words in a sequence, greatly improving context. This led to **BERT** in 2019, which used bidirectional context to gain a more nuanced understanding of text. Initially, BERT created sentence representations using a special [CLS] token. This method produced poor sentence-level embeddings, often worse than simply averaging word embeddings. The breakthrough came in 2019 with **Sentence-BERT (SBERT)**, which restructured BERT into siamese and triplet network architectures. SBERT greatly improved the quality and speed of sentence embeddings.
- These advances let systems capture deeper semantic links between words and phrases and better identify which documents most likely match a user's query. In practice, retrieval models often still use keyword matching for efficiency, while **rankers and re-rankers** apply more advanced semantic models to refine result ordering, a topic examined later in this chapter.

## 6.2 Latent Semantic Analysis

- Previously, we viewed documents as sparse high-dimensional vectors, using either binary (set-of-words) or $tf \cdot idf$ values. We leveraged vector sparsity with the inverted index for fast document retrieval, and employed parallelism to enhance performance and concurrency. However, this approach assumed term independence, limiting matching for similar but different terms like “house”, “villa”, and “houses”. To address this, we used stemming techniques to reduce words to a common stem and lemmatization to identify synonyms and hypernyms. This improved matching between query and document terms, such as query expansion for “house” queries with synonyms like “villa”.
- Stemming and lemmatization, while effective, are language-dependent, general across various documents, and demand substantial manual effort for high-quality results. Moreover, they often fail to address semantically similar terms specific to a collection. To illustrate this with an example from this course, we discussed tokens, words, and terms, which, though not identical, can be seen as closely related. For instance, a search for “tokens” might also yield paragraphs mentioning “terms”. To enhance term matching in such situations, we need a method capable of learning semantic relationships specific to each context, ideally without manual intervention.
- **Latent Semantic Indexing (LSI)** is a method to understand the semantic meaning of terms in document collections through dimensionality reduction. The sparse, high-dimensional document vectors are transformed into a compact, lower-dimensional representation unique to that collection. These dimensions no longer align with individual terms but instead represent latent topics that characterize the collection. However, these topics may not precisely align with our conceptual understanding. Both documents and terms can be expressed as combinations of these topics, and it is this connection between terms and topics that partly explains the extracted topics. For instance, consider a retrieved topic explained as `3.45*airplane + 0.34*bird`. We might interpret this as “flying”. However, the model can also generate a mathematically justified description as `3.45*airplane + 0.34*flower`, which does not immediately align with a concept we commonly associate with our thinking.
- LSI was developed at Bell Labs in the 1980s by Susan Dumais and Scott Deerwester to enhance information retrieval systems. The first article was published in 1988, and a patent was granted the same year (the patent has since expired). Although LSI found applications in various scenarios, it faced challenges due to the substantial computational requirements for topic learning and the inability to use inverted files with the compact lower-dimensional vectors. While computational challenges have been addressed, LSI now lacks the fine-grained semantic associations found in embeddings.

- We briefly introduce the mathematical concepts before delving into their application in text retrieval. In linear algebra, **eigenvector decomposition** is a technique used to convert a quadratic $n \times n$-matrix $\mathbf{A}$ into a set of eigenvalues $\lambda_i$ and corresponding eigenvectors $v_i$ of length 1, satisfying the equation for matrix $\mathbf{A}$:

$$\mathbf{A}\boldsymbol{v} = \lambda\boldsymbol{v}$$

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\top$$

  Eigenvalues are determined by solving the equation $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$, equivalent to finding roots of a polynomial of degree $n$. Eigenvalues can be real or complex and may have multiplicity. The associated eigenvectors are orthonormal. The formula on the right illustrates the eigenvalue decomposition of matrix $\mathbf{A}$. Let $r \leq n$ be the rank of $\mathbf{A}$. We can express matrix $\mathbf{A}$ as the product of $\mathbf{U}$ (an $n \times n$ matrix) containing the eigenvectors and $\boldsymbol{\Lambda}$ (an $n \times n$ diagonal matrix) containing the corresponding eigenvalues (with $n - r$ values equal to $0$).

- Eigenvectors describe the directions in which the matrix scales and stretches, valuable for characterizing latent topics in a text corpus. However, the document-term matrix is generally non-square. Therefore, we use the **Singular Value Decomposition (SVD)**, a generalization of the eigenvalue decomposition. Let $\mathbf{A}$ be an $\mathrm{m} \times n$-matrix of rank $r$. There exists an $r \times r$-diagonal matrix $\mathbf{S}$, an orthonormal $\mathrm{m} \times r$-matrix $\mathbf{U}$, and an orthonormal $\mathrm{n} \times r$-matrix $\mathbf{V}$ such that:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$$

  The connection between singular value and eigenvalue decomposition is shown with the following representations. Specifically, the singular values are the square roots of the eigenvalues for the matrices $\mathbf{A}^\top\mathbf{A}$ and $\mathbf{A}\mathbf{A}^\top$:

$$\mathbf{A}^\top\mathbf{A} = (\mathbf{U}\mathbf{S}\mathbf{V}^\top)^\top(\mathbf{U}\mathbf{S}\mathbf{V}^\top) = \mathbf{V}\mathbf{S}\mathbf{U}^\top\mathbf{U}\mathbf{S}\mathbf{V}^\top = \mathbf{V}\mathbf{S}^2\mathbf{V}^\top$$

$$\mathbf{A}\mathbf{A}^\top = (\mathbf{U}\mathbf{S}\mathbf{V}^\top)(\mathbf{U}\mathbf{S}\mathbf{V}^\top)^\top = \mathbf{U}\mathbf{S}\mathbf{V}^\top\mathbf{V}\mathbf{S}\mathbf{U}^\top = \mathbf{U}\mathbf{S}^2\mathbf{U}^\top$$

- We can express $\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$ as a sum of vector products, known as dyadic vector products.

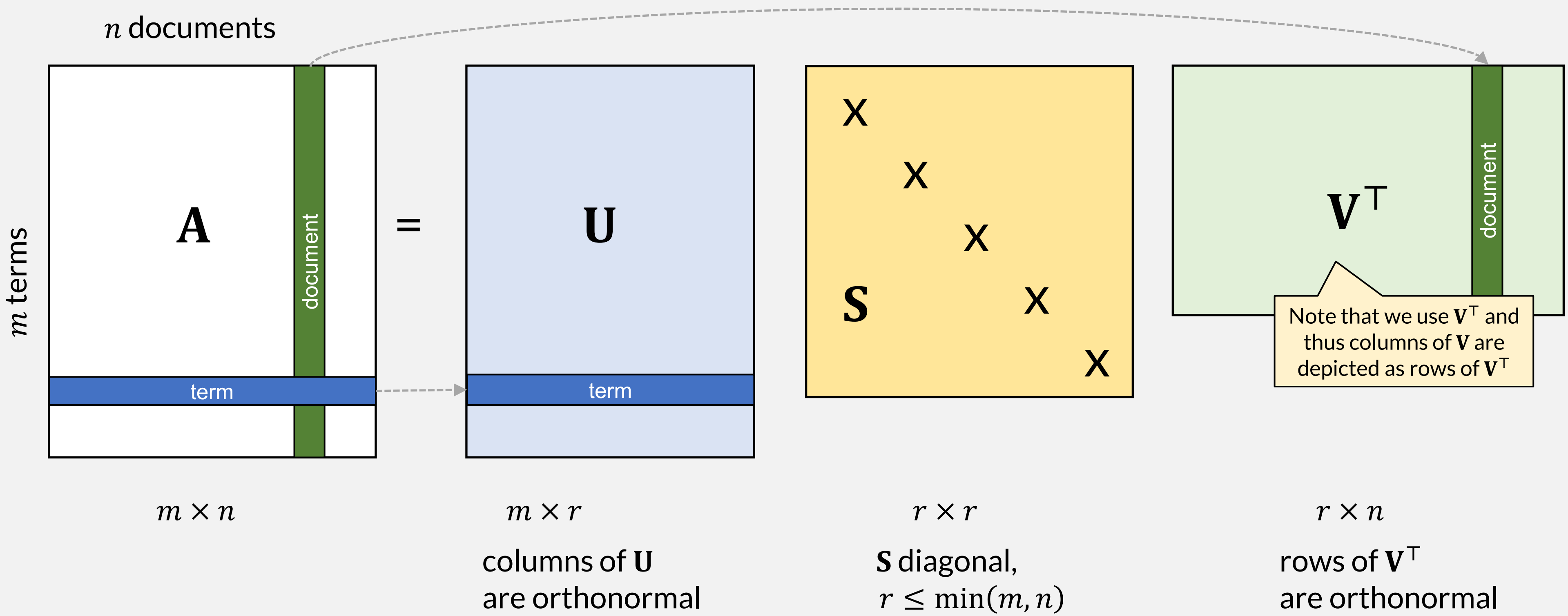
$$\mathbf{A} = s_1\left(\boldsymbol{u_1}\boldsymbol{v_1}^\top\right) + s_2\left(\boldsymbol{u_2}\boldsymbol{v_2}^\top\right) + \cdots + s_r\left(\boldsymbol{u_r}\boldsymbol{v_r}^\top\right)$$

  By omitting one or more of these summands, we obtain an approximation for $\mathbf{A}$. We get the best approximation (Frobenius norm) of rank $k < r$ by keeping the summands of the $k$ largest singular values and their corresponding columns in $\mathbf{U}$ and $\mathbf{V}$. This provides then a mapping from the original $m$-dimensional to a compact $k$-dimensional space.

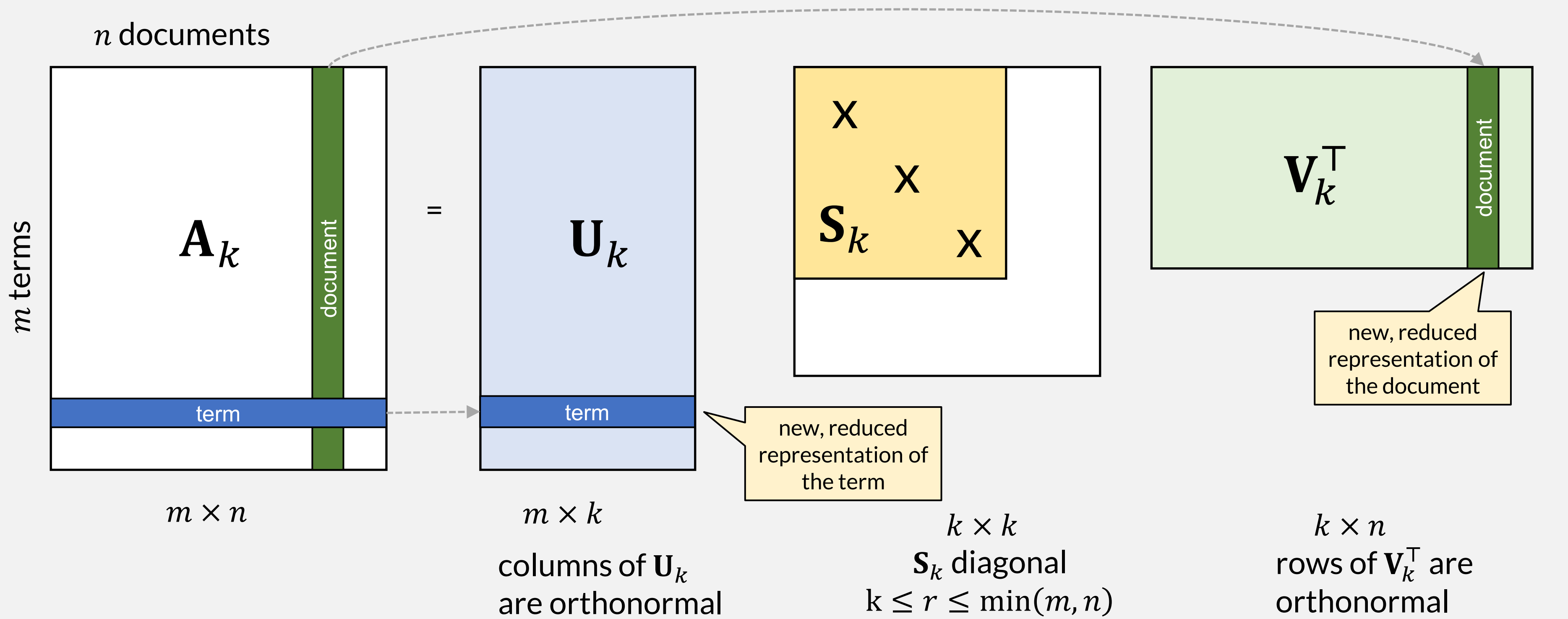
# 6.2.1 Application in Text Retrieval

- In text retrieval, we can apply the **Singular Value Decomposition (SVD)** to the document-term matrix $\mathbf{A}$, typically using $tf \cdot idf$ weighted components. SVD decomposes $\mathbf{A}$ into matrices $\mathbf{U}$, $\mathbf{S}$, and $\mathbf{V}$, reducing them to the intrinsic rank $r \leq \min(m,n)$. Matrix $\mathbf{U}$ represents the $m$ terms of the vocabulary in an $r$-dimensional space, $\mathbf{S}$ contains singular values (usually sorted by decreasing value on the diagonal), and $\mathbf{V}$ holds the $n$ documents in the collection as representations in an $r$-dimensional space.
- As we enumerate singular values in decreasing order, the values quickly diminish in magnitude, allowing us to remove many of them while still be able to accurately reconstruct matrix $\mathbf{A}$. Removing singular values and their corresponding columns in $\mathbf{U}$ and $\mathbf{V}$ reduces the dimensionality of the new term and document representations.

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$$



- **Dimensionality reduction:** When we reduce the number of singular values in the dyadic vector product representation of $\mathbf{A}$, we also eliminate corresponding columns in $\mathbf{U}$ and $\mathbf{V}$, resulting in reduced matrices $\mathbf{U}_k$, $\mathbf{S}_k$, and $\mathbf{V}_k$, as illustrated below. Consequently, the new representations for documents and terms in the original document-term matrix are now more compact, with $k$ dimensions. Columns in $\mathbf{V}_k^\top$ (equivalent to rows in $\mathbf{V}_k$) contain the new representations for documents, each dimension expressing a latent topic in the corpus. Consequently, the $i$-th row in $\mathbf{V}_k^\top$ (or $i$-th column in $\mathbf{V}_k$) describes the $i$-th latent topic in terms of the documents. Similarly, columns in $\mathbf{U}_k$ contain the new representations for terms, again with each dimension expressing a latent topic in the corpus. Consequently, the $i$-th column in $\mathbf{U}_k$ portrays the relationship between the $i$-th topic and the vocabulary terms. This enables us to describe the topics identified by LSI.
- When new documents, possibly with new terms, are added to the collection, we must repeat this process for the updated document-term matrix to adapt to topic changes. To avoid recalculations for each new document, the following pages detail an approximate method that delays the need for renewed SVD computations.

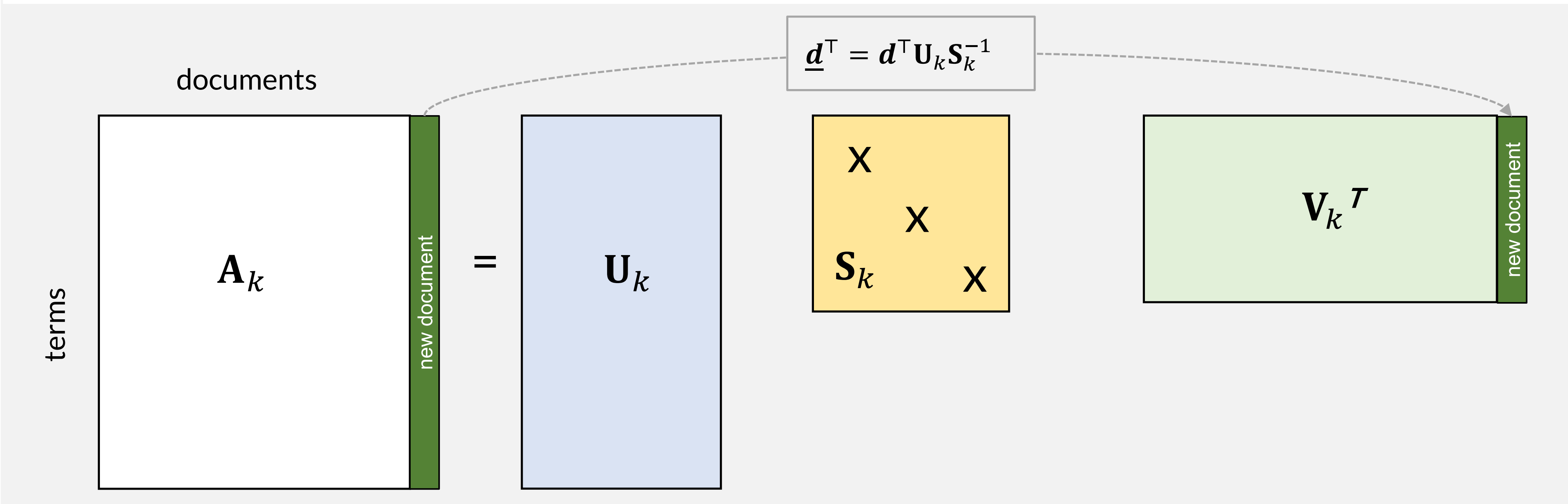


- **Inserting new documents (approximation):** Using the approximate representation $\mathbf{A}_k$ from the reduced singular value decomposition, we can derive a mapping from the original term space to the topic space in three steps as follows: 1) transpose both sides of the equation, 2) multiply first by $\left(\mathbf{U}_k^\top\right)^{-1} = \mathbf{U}_k$ and then by $\mathbf{S}_k^{-1}$, and 3) focus on a single document in $\mathbf{A}_k^\top$ (denoted as $\boldsymbol{d}^\top$) and $\mathbf{V}_k$ (denoted as $\underline{\boldsymbol{d}}^\top$).

$$\mathbf{A}_k = \mathbf{U}_k\mathbf{S}_k\mathbf{V}_k^\top \xrightarrow{1} \mathbf{A}_k^\top = \mathbf{V}_k\mathbf{S}_k\mathbf{U}_k^\top \xrightarrow{2} \mathbf{V}_k = \mathbf{A}_k^\top\mathbf{U}_k\mathbf{S}_k^{-1} \xrightarrow{3} \underline{\boldsymbol{d}}^\top = \boldsymbol{d}^\top\mathbf{U}_k\mathbf{S}_k^{-1}$$

  As long as the new documents do not significantly alter the collection's characteristics, the latent topics remain relatively consistent, allowing us to delay the SVD recalculations. When the collection size has increased by a certain threshold percentage, we can initiate recalculations and operate with updated topics.

- What about new terms? We can map new terms to their reduced space in $\mathbf{U}_k$ with the formula $\underline{\boldsymbol{t}}^\top = \boldsymbol{t}^\top\mathbf{V}_k\mathbf{S}_k^{-1}$. However, because a new term appears only in a new document, the term vector $\boldsymbol{t}$ solely depends on the reduced representation of that new document in $\mathbf{V}_k$ and the values in $\mathbf{S}_k$. If a document has two new terms, they both receive the same approximate representation $\underline{\boldsymbol{t}}$ due to this. A better approach is to disregard terms not in the vocabulary and introduce them through a fresh SVD calculation.



- Like vector space retrieval, LSI treats queries as miniature documents. To compare them with the documents in the collection, we must initially map the query, similar to newly added documents, to the reduced topic space:

$$\underline{\boldsymbol{q}}^\top = \boldsymbol{q}^\top \mathbf{U}_k \mathbf{S}_k^{-1}$$

- We apply the same similarity functions as in Vector Space retrieval, using either the dot-product or the cosine measure, to compare the query with the document collection.

$$sim_{dot}(Q, D_i) = \boldsymbol{q} \cdot \boldsymbol{d}_i = \sum_{j=1}^{M} q_j \cdot d_{i,j}$$

$$sim_{cos}(Q, D_i) = \frac{\boldsymbol{q} \cdot \boldsymbol{d}_i}{\|\boldsymbol{q}\| \cdot \|\boldsymbol{d}_i\|} = \frac{\sum_{j=1}^{M} q_j \cdot d_{i,j}}{\sqrt{\sum_{j=1}^{M} q_j^2} \cdot \sqrt{\sum_{j=1}^{M} d_{i,j}^2}}$$

- The main difference is that we are now comparing two dense vectors. Because the query vector usually has nonzero values in every dimension, we cannot use an inverted index to accelerate the search. Instead we must compute similarity with every document and then sort them by score. Although LSI vectors have fewer dimensions than the original vectors, we still must process much more data and cannot prune documents as efficiently as with inverted indexes. We will cover indexing methods for dense vector search in a later chapter. For now it is important to balance more topics for a richer semantic representation of the corpus's latent topics with fewer dimensions to reduce retrieval costs.
- It is possible to reuse the mapping from document vectors to a compact low-dimensional representation when the same vocabulary appears in different collections. However, we usually have to run LSI separately on each corpus. LSI learns not only the important topics from terms but also how those terms are used in documents. Therefore the mapping for an IT article collection will be quite different from the mapping for news articles. Using a generic mapping would lower topic quality, require more topics to cover a broad range of applications, and could harm retrieval performance.

## 6.2.2 A Simple Example with LSI

- Let's consider a simple example to illustrate, step-by-step, how LSI works:

| | |
|---|---|
| c1 | Human machine interface for Lab ABC computer applications |
| c2 | A survey of user opinion of computer system response time |
| c3 | The EPS user interface management system |
| c4 | System and human system engineering testing of EPS |
| c5 | Relation of user-perceived response time to error measurement |
| m1 | The generation of random, binary, unordered trees |
| m2 | The intersection graph of paths in trees |
| m3 | Graph minors IV: Widths of trees and well-quasi-ordering |
| m4 | Graph minors: A survey |

- We see that the collection has two separate document groups: one about human interfaces and one about graph algorithms. If we search the collection for "**human-computer interaction**", not every relevant document c1 to c5 uses those exact words. Traditional retrieval models must expand the query to add synonyms and broader terms to improve search quality, for example changing "human" to "user" and "computer" to "system". Now we will assess how LSI performs on this example.

- First, we create the document-term matrix. For simplicity, we consider only term frequencies and assume equal importance for all terms (i.e., $idf = 1$ for all terms). We also exclude stop words and terms that appear only once in the collection since they are unlikely to contribute to topics, being isolated to a single document.

**A =**

*(m=12, n=9)*

| | c1 | c2 | c3 | c4 | c5 | m1 | m2 | m3 | m4 |
|---|---|---|---|---|---|---|---|---|---|
| human | 1 | | | 1 | | | | | |
| interface | 1 | | 1 | | | | | | |
| computer | 1 | 1 | | | | | | | |
| user | | 1 | 1 | | 1 | | | | |
| system | | 1 | 1 | 2 | | | | | |
| response | | 1 | | | 1 | | | | |
| time | | 1 | | | 1 | | | | |
| EPS | | | 1 | 1 | | | | | |
| survey | | 1 | | | | | | | 1 |
| trees | | | | | | 1 | 1 | 1 | |
| graph | | | | | | | 1 | 1 | 1 |
| minors | | | | | | | | 1 | 1 |

- Then we apply the singular value decomposition on the matrix $\mathbf{A}$ below. The results are shown below.

$$\mathbf{U} = \begin{bmatrix} 0.2214 & -0.1132 & 0.2890 & -0.4148 & -0.1063 & -0.3410 & 0.5227 & -0.0605 & -0.4067 \\ 0.1976 & -0.0721 & 0.1350 & -0.5522 & 0.2818 & 0.4959 & -0.0704 & -0.0099 & -0.1089 \\ 0.2405 & 0.0432 & -0.1644 & -0.5950 & -0.1068 & -0.2550 & -0.3022 & 0.0623 & 0.4924 \\ 0.4036 & 0.0571 & -0.3378 & 0.0991 & 0.3317 & 0.3848 & 0.0029 & -0.0004 & 0.0123 \\ 0.6445 & -0.1673 & 0.3611 & 0.3335 & -0.1590 & -0.2065 & -0.1658 & 0.0343 & 0.2707 \\ 0.2650 & 0.1072 & -0.4260 & 0.0738 & 0.0803 & -0.1697 & 0.2829 & -0.0161 & -0.0539 \\ 0.2650 & 0.1072 & -0.4260 & 0.0738 & 0.0803 & -0.1697 & 0.2829 & -0.0161 & -0.0539 \\ 0.3008 & -0.1413 & 0.3303 & 0.1881 & 0.1148 & 0.2722 & 0.0330 & -0.0190 & -0.1653 \\ 0.2059 & 0.2736 & -0.1776 & -0.0324 & -0.5372 & 0.0809 & -0.4669 & -0.0363 & -0.5794 \\ 0.0127 & 0.4902 & 0.2311 & 0.0248 & 0.5942 & -0.3921 & -0.2883 & 0.2546 & -0.2254 \\ 0.0361 & 0.6228 & 0.2231 & 0.0007 & -0.0683 & 0.1149 & 0.1596 & -0.6811 & 0.2320 \\ 0.0318 & 0.4505 & 0.1411 & -0.0087 & -0.3005 & 0.2773 & 0.3395 & 0.6784 & 0.1825 \end{bmatrix}$$

$$\mathbf{S} = \begin{bmatrix} 3.3409 & & & & & & & & \\ & 2.5417 & & & & & & & \\ & & 2.3539 & & & & & & \\ & & & 1.6445 & & & & & \\ & & & & 1.5048 & & & & \\ & & & & & 1.3064 & & & \\ & & & & & & 0.8459 & & \\ & & & & & & & 0.5601 & \\ & & & & & & & & 0.3637 \end{bmatrix}$$

$$\mathbf{V}^\top = \begin{bmatrix} 0.1974 & 0.6060 & 0.4629 & 0.5421 & 0.2795 & 0.0038 & 0.0146 & 0.0241 & 0.0820 \\ -0.0559 & 0.1656 & -0.1273 & -0.2318 & 0.1068 & 0.1928 & 0.4379 & 0.6151 & 0.5299 \\ 0.1103 & -0.4973 & 0.2076 & 0.5699 & -0.5054 & 0.0982 & 0.1930 & 0.2529 & 0.0793 \\ -0.9498 & -0.0286 & 0.0416 & 0.2677 & 0.1500 & 0.0151 & 0.0155 & 0.0102 & -0.0246 \\ 0.0457 & -0.2063 & 0.3783 & -0.2056 & 0.3272 & 0.3948 & 0.3495 & 0.1498 & -0.6020 \\ -0.0766 & -0.2565 & 0.7244 & -0.3689 & 0.0348 & -0.3002 & -0.2122 & 0.0001 & 0.3622 \\ 0.1773 & -0.4330 & -0.2369 & 0.2648 & 0.6723 & -0.3408 & -0.1522 & 0.2491 & 0.0380 \\ -0.0144 & 0.0493 & 0.0088 & -0.0195 & -0.0583 & 0.4545 & -0.7615 & 0.4496 & -0.0696 \\ -0.0637 & 0.2428 & 0.0241 & -0.0842 & -0.2624 & -0.6198 & 0.0180 & 0.5199 & -0.4535 \end{bmatrix}$$

- To simplify document presentation later, we choose $k = 2$ and adjust all matrices accordingly. The matrix $\mathbf{V}_k{}^T$ contains the reduced documents as 2-dimensional vectors in its columns, maintaining the same order as in the collection. On the next page, we use these vectors to illustrate the document positions in this 2-topic space.

$\mathbf{U}_k$

| | |
|---|---|
| 0.2214 | -0.1132 |
| 0.1976 | -0.0721 |
| 0.2405 | 0.0432 |
| 0.4036 | 0.0571 |
| 0.6445 | -0.1673 |
| 0.2650 | 0.1072 |
| 0.2650 | 0.1072 |
| 0.3008 | -0.1413 |
| 0.2059 | 0.2736 |
| 0.0127 | 0.4902 |
| 0.0361 | 0.6228 |
| 0.0318 | 0.4505 |

$\mathbf{S}_k$

| | |
|---|---|
| 3.3409 | |
| | 2.5417 |

$\mathbf{V}_k{}^T$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0.1974 | 0.6060 | 0.4629 | 0.5421 | 0.2795 | 0.0038 | 0.0146 | 0.0241 | 0.0820 |
| -0.0559 | 0.1656 | -0.1273 | -0.2318 | 0.1068 | 0.1928 | 0.4379 | 0.6151 | 0.5299 |

reduced representation for c1

- Next, we project the query into the topic space. Since "interaction" is not in the vocabulary, the query vector contains only two 1s. This vector is then mapped to the 2-dimensional topic space. On the left side, we also display the approximate representation of $\mathbf{A}$ with $k = 2$. While it may not closely resemble the original document-term matrix, $\mathbf{A}_k$ shown below is the best rank-2 approximation for $\mathbf{A}$ under the Frobenius norm.

$\mathbf{A}_k =$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0.1621 | 0.4005 | 0.3790 | 0.4676 | 0.1760 | -0.0527 | -0.1151 | -0.1591 | -0.0918 |
| 0.1406 | 0.3698 | 0.3290 | 0.4004 | 0.1650 | -0.0328 | -0.0706 | -0.0968 | -0.0430 |
| 0.1524 | 0.5050 | 0.3579 | 0.4101 | 0.2362 | 0.0242 | 0.0598 | 0.0869 | 0.1240 |
| 0.2580 | 0.8411 | 0.6057 | 0.6974 | 0.3923 | 0.0331 | 0.0832 | 0.1218 | 0.1874 |
| 0.4488 | 1.2344 | 1.0509 | 1.2658 | 0.5563 | -0.0738 | -0.1547 | -0.2096 | -0.0489 |
| 0.1596 | 0.5817 | 0.3752 | 0.4169 | 0.2765 | 0.0559 | 0.1322 | 0.1889 | 0.2169 |
| 0.1596 | 0.5817 | 0.3752 | 0.4169 | 0.2765 | 0.0559 | 0.1322 | 0.1889 | 0.2169 |
| 0.2185 | 0.5496 | 0.5110 | 0.6281 | 0.2425 | -0.0654 | -0.1425 | -0.1966 | -0.1079 |
| 0.0969 | 0.5321 | 0.2299 | 0.2118 | 0.2665 | 0.1368 | 0.3146 | 0.4444 | 0.4250 |
| -0.0613 | 0.2321 | -0.1389 | -0.2656 | 0.1449 | 0.2404 | 0.5461 | 0.7674 | 0.6637 |
| -0.0647 | 0.3353 | -0.1456 | -0.3014 | 0.2028 | 0.3057 | 0.6949 | 0.9766 | 0.8487 |
| -0.0431 | 0.2539 | -0.0967 | -0.2079 | 0.1519 | 0.2212 | 0.5029 | 0.7069 | 0.6155 |

$\boldsymbol{q}$

| |
|---|
| 1 |
| 0 |
| 1 |
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |

$\xrightarrow{\mathbf{U}_k \mathbf{S}_k^{-1}}$

$\underline{\boldsymbol{q}}$

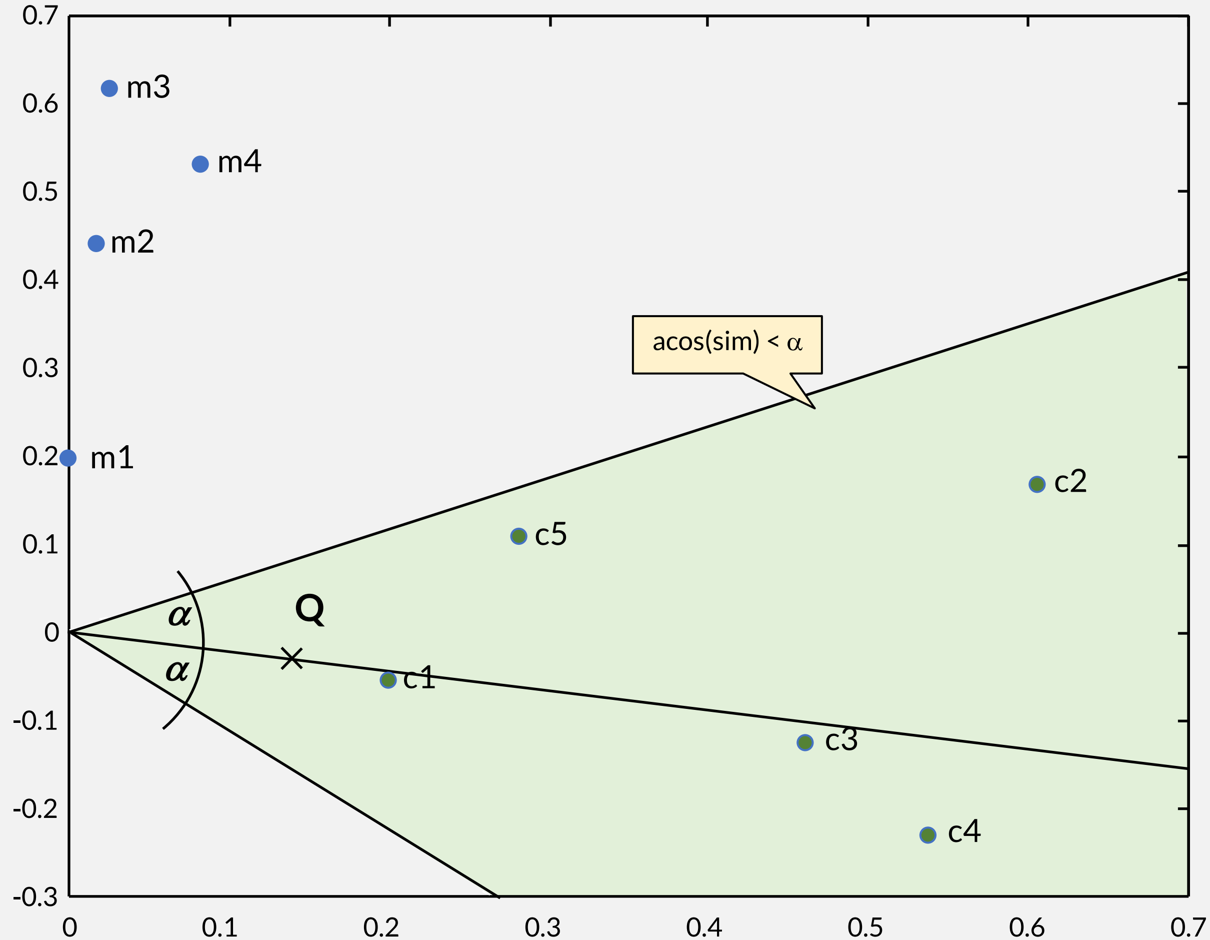
| |
|---|
| 0.1382 |
| -0.0276 |

- The right side visualizes the document collection. We notice that topic 1 (horizontal) aligns more with documents c1…c5, while topic 2 (vertical) aligns with documents m1…m4
- The query is represented at (0.14, -0.03), pointing toward the c-documents. When we apply a cosine similarity measure, we select the green area, which encompasses the subspace with an angle of at most $\alpha$ to the query vector. This area includes all the c-documents, and we can arrange them as follows: c1 < c3 < c4 < c2 < c5.
- Interestingly, c3 ranks as the second-best document despite lacking any of the query terms. Due to the SVD reduction, some of its terms align with topics similar to the query terms, making c3 highly relevant.
- We can extract the meaning of topic 1 from the $\mathbf{U}_k$ matrix (first column).

```
0.64*system   + 0.40*user +
0.30*eps      + 0.27*time +
0.27*response + 0.24*computer
```

and for topic 2:

```
0.62*graph    + 0.49*trees +
0.45*minors   + 0.27*survey
```



# 6.3 Word Embeddings

- Word embeddings, like latent semantic indexing, map words (or sub-words) to a $d$-dimensional space, where $d$ is much smaller than the vocabulary size, usually ranging from 100 to 1,000 dimensions. The key distinction from LSI is as follows:
  - **LSI maps documents to vectors**, while **embeddings map vocabularies to vectors**. Modern transformer-based embeddings, however, can generate embeddings for entire sequences, capturing richer semantics than earlier models that aggregated vectors along the sequence.
  - LSI examines **semantic relationships between terms globally**, without considering the distance or context of term occurrences. Earlier embedding models used **defined context windows** around words to establish these relationships. Transformer-based models address the challenge of sequence-to-sequence transformations by employing **self-attention mechanisms** to acquire contextualized word embeddings.
  - **LSI reduces dimensionality** and minimizes the loss on the original document-term matrix with lower-dimensional representations. However, SVD can be computationally expensive and does not scale efficiently (although it can handle millions of documents). Embeddings employ neural network-based learning techniques to **optimize the loss between predictions and targets**, offering a more efficient approach.
  - **LSI employs a static vocabulary** and necessitates retraining to accommodate new terms (such as names or brands). Earlier embedding models are vocabulary-based as well, but **some operate at a sub-word level**, enabling them to handle unknown or misspelled terms. **Transformer-based models utilize pre-trained sub-words like BPE and word pieces** to generate embeddings for a smaller vocabulary. They can address unseen words by breaking them into sub-parts and providing embeddings for these smaller tokens.
  - LSI mappings can theoretically be applied to other collections, but this often results in suboptimal performance because **latent topics are shaped by both documents and terms specific to a collection**. Therefore, an LSI model tailored for IT might not excel with biology articles. Conversely, **embeddings create mappings based on terms and context specific to the language**, enabling reuse in different but similar collections. However, optimal performance with embeddings requires collection-specific optimization. For instance, embeddings from one language do not transfer well to another language. Transformer-based models may also face issues with suboptimal tokens (e.g., BPE, word pieces) when transferred to different languages, affecting embedding quality.

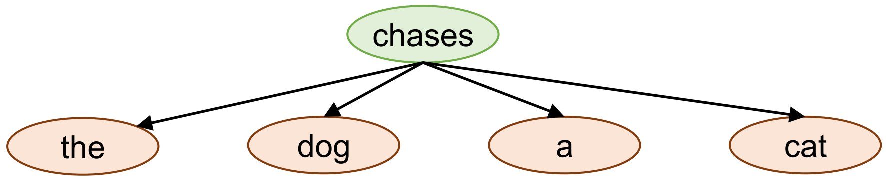
## 6.3.1 Word2vec

- Word2Vec, introduced in 2013 by Mikolov et al., maps words to a $d$-dimensional vector space. The mapping uses context windows around each word and is trained in a self-supervised way, so no labeled data is needed. Word2Vec has two variants:
  - The **Skip-Gram Model** operates within a context window of size $2m + 1$ around a center word. It learns word representations by predicting context words from the center word. For example, if the center word is “apple”, the model predicts which words are likely to appear in the context window, like “juice”, “tree”, “red”, or “eat”. Words like “complex”, “retrieval”, “planet”, and “learn” are less likely to be found near “apple”.
  - The **Continuous Bag of Words (CBOW) Model** employs a similar method with a context window of size $2m + 1$. However, it learns representations to predict the center word from all the context words. For instance, in the sentence “the apple is [blank] and tastes delicious”, CBOW would aim to predict the center word "[blank]" based on the surrounding words “the”, “apple”, “is”, “and”, “tastes”, and “delicious”. A word like “ripe” is a better match than the word “car”.
- Both models yield a mapping from the **vocabulary to a $d$-dimensional vector** which can be used for different tasks:
  - **Semantic Word Analysis:** Vector representations establish semantic relationships between terms, enabling the use of similarity measures (e.g., cosine, Euclidean, dot-product) to identify closely related terms. These relationships are learned and adapted for the collection, eliminating the need for manual dictionary curation.
  - **Token Classification:** Word embeddings can enhance part-of-speech and named entity recognition, replacing the need for manual or rule-based methods.
  - **Machine Translation:** When translating between languages, word embeddings assist in selecting the optimal words and arranging them in the target language by considering the broader context. This helps to resolve ambiguities and to improve translation quality.
  - **Text Classification:** By representing entire documents as sequences of vectors using embeddings, we can enhance machine learning approaches. These semantically rich representations can lead to improved quality even with simple models. Similarly, embeddings help to discover latent topics in collections.
  - **Text Retrieval:** Word embeddings' semantic relationships improve query-document matching in retrieval tasks, which we will explore in greater detail later in this section.

- **The Skip-Gram Model:** Let's take the phrase "**the dog chases a cat**" with its center word "**chases**" and a context window of size $2m+1=5$ ($m=2$ words before and $m=2$ words after the center word). In the skip-gram model, we assess the conditional probability of the center word generating the surrounding words, assuming independence among the surrounding words:

$$P(the\ dog\ the\ cat \mid chases) = P(the|chases) \cdot P(dog|chases) \cdot P(a|chases) \cdot P(cat|chases)$$

We can illustrate this relationship graphically as follows:




  - In the skip-gram model, a word $w_i$ is represented by two $d$-dimensional vectors $\boldsymbol{v}_i \in \mathbb{R}^d$ and $\boldsymbol{u}_i \in \mathbb{R}^d$ when employed as a center word ($\boldsymbol{v}_i$) or as a surrounding word ($\boldsymbol{u}_i$). We can employ a softmax operation to model the conditional probability of generating the surrounding word $w_s$ from the center word $w_c$:

$$P(w_s \mid w_c) = \frac{e^{\boldsymbol{u}_s^T \boldsymbol{v}_c}}{\sum_{i \in \mathbb{T}} e^{\boldsymbol{u}_i^T \boldsymbol{v}_c}}$$

    where $\mathbb{T}$ represents the set of words in a corpus of documents. Assuming the corpus consists of a sequence of $n$ words $w_1 \dots w_n$, the likelihood function for the skip-gram model is expressed as:

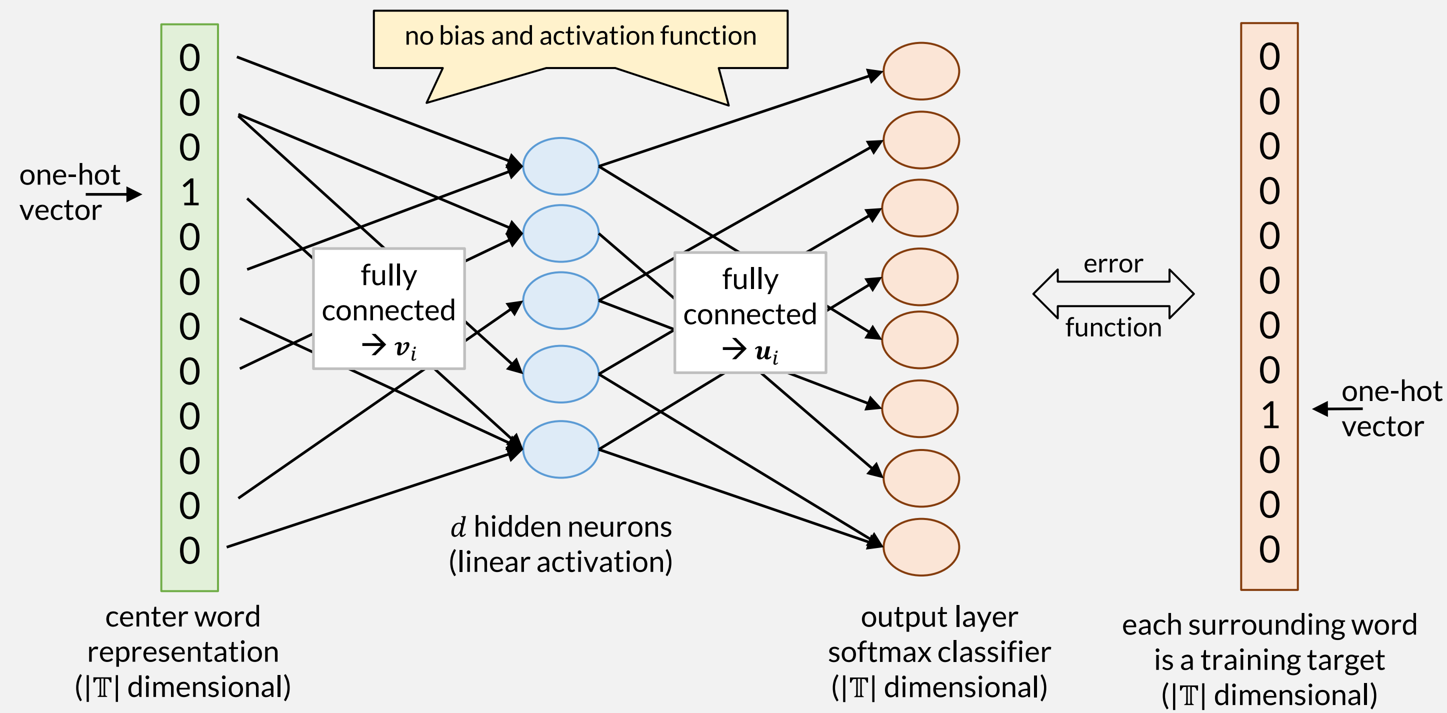
$$\prod_{i=m+1}^{n-m} \prod_{j=i-m, j \neq i}^{i+m} P(w_j|w_i)$$

    with a context window of size $2m+1$. The objective is to find vectors $\boldsymbol{v}_i \in \mathbb{R}^d$ and $\boldsymbol{u}_i \in \mathbb{R}^d$ that maximize the likelihood function, and we then can use the mapping $w_i \rightarrow v_i$ to translate words from the vocabulary to a $d$-dimensional vector.

– To optimize the model's likelihood function, we can minimize the following loss function instead:

$$-\sum_{i=m+1}^{n-m} \sum_{j=i-m, j\neq i}^{i+m} \log P(w_j|w_i)$$

We cannot directly find a solution for the above optimization problem. Instead, we employ a gradient descent method during a training phase to minimize the loss. An alternative approach is the modeling of the optimization problem with a single hidden layer network, as shown at the bottom of this page. The model takes a one-hot vector representing the center word as input. It then passes through a fully connected network without bias and activation function. The columns in the corresponding weight matrix represent vectors $\boldsymbol{v}_i$. The hidden layer comprises $d$ neurons and is followed by another fully connected network, again without bias and activation function. The columns in this matrix correspond to the vectors $\boldsymbol{u}_i$. A softmax classifier on the output layer is compared with the target one-hot vector of surrounding words.



- We can employ a self-supervised approach to train the model, which means we can utilize supervised learning techniques without relying on external or human-provided labels. To train word2vec models using a large text corpus, we create the training, test, and validation datasets from the corpus as follows:
  - We enumerate all windows of size $2m + 1$ in the corpus. To avoid incorrect associations across sentence boundaries, we first split texts into sentences and ensure that windows are confined within sentences.
  - For every window, we generate pairs of center word and surrounding word. In each window, we produce $2m$ data samples. As an example, consider the window "the red **apple** tastes fine". We create four pairs: (apple, the), (apple, red), (apple, tastes), and (apple, fine). Each of these pairs contributes to training the model.
  - Optionally, we can sub-sample or exclude pairs with common terms (stop words). Sub-sampling reduces the numbers of pairs often to as low as 1%. This enhances accuracy and accelerates the training.
  - Optionally, we can lemmatize or tokenize words (e.g., stemming) to decrease vocabulary size, but this may restrict the applicability of vectors for some contexts.
- During training, we utilize pairs of center and surrounding word to compute the loss function and make adjustments to the model weights, that is the vectors $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$. This process is iterated until the loss function reaches a sufficiently low value. The outcome is a vocabulary-to-vector mapping, represented as $\boldsymbol{v}_i$.
- To efficiently handle word pairs without using memory-intensive one-hot vectors, a `PyTorch` implementation can use the `Embedding` layer and the `CrossEntropyLoss` function, directly working with token IDs instead of vectors. The code below defines the model on the left side, and outlines the training process on the right side.

```python
class SkipGramModel(nn.Module):
    def __init__(self, vocab_size: int, embedd_size: int):
        super().__init__()
        self.embeddings = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedd_size)
        self.linear = nn.Linear(
            in_features=embedd_size,
            out_features=vocab_size,
        )

    def forward(self, inputs):
        x = self.embeddings(inputs)
        x = self.linear(x)
        return x
```

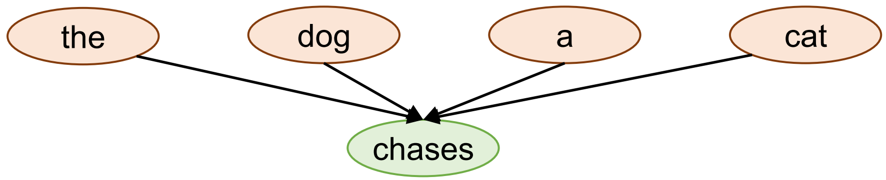
```python
# sketch of training process
model = SkipGramModel(vocab_size=len(vocab), embedd_size=100)
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()

for inputs, labels in batch_loader_skipgram(text_corpus):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, labels)
    loss.backward()
    optimizer.step()
```

- Word2vec also defines an alternative model known as the **Continuous Bag of Words (CBOW)** model. It operates similarly to the skip-gram model but focuses on the likelihood that the surrounding words generate the center word. To illustrate, let's revisit the phrase "**the dog chases a cat**" with the center word "**chases**" and a context window of size $2m + 1 = 5$. We evaluate the conditional probability of the surrounding words generating the center word, with an assumption of independence among the surrounding words:

$$P(chases \mid the\ dog\ the\ cat)$$

We can illustrate this relationship graphically as follows:




  - In the CBOW model, a word $w_i$ is represented by two $d$-dimensional vectors $\boldsymbol{v}_i \in \mathbb{R}^d$ and $\boldsymbol{u}_i \in \mathbb{R}^d$ when employed as a center word ($\boldsymbol{v}_i$) or as a surrounding word ($\boldsymbol{u}_i$). We can employ a softmax operation to model the conditional probability of generating the center word $w_c$ from its surrounding words $w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m}$:

$$P(w_c \mid w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m}) = \frac{e^{\frac{1}{2m}\boldsymbol{u}_c^T(\boldsymbol{v}_{c-m}+\cdots+\boldsymbol{v}_{c-1}+\boldsymbol{v}_{c+1}+\cdots+\boldsymbol{v}_{c+m})}}{\sum_{i\in\mathbb{T}} e^{\boldsymbol{u}_i^T(\boldsymbol{v}_{i-m}+\cdots+\boldsymbol{v}_{i-1}+\boldsymbol{v}_{i+1}+\cdots+\boldsymbol{v}_{i+m})}}$$

where $\mathbb{T}$ represents the set of words in a corpus of documents. Assuming the corpus consists of a sequence of $n$ words $w_1 \dots w_n$, the likelihood function for the skip-gram model is expressed as:

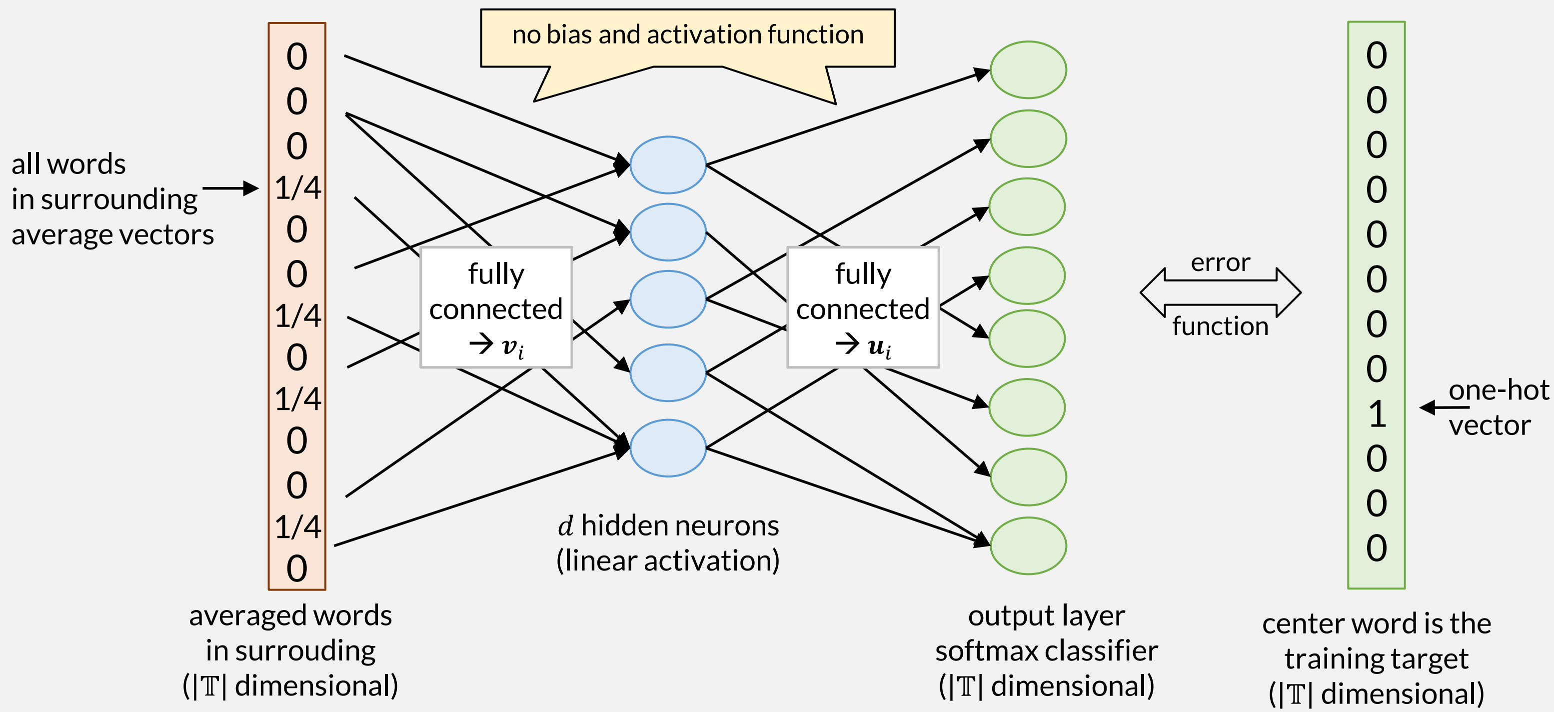
$$\prod_{i=m+1}^{n-m} P(w_c \mid w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m})$$

with a context window of size $2m + 1$. The objective is to find vectors $\boldsymbol{v}_i \in \mathbb{R}^d$ and $\boldsymbol{u}_i \in \mathbb{R}^d$ that maximize the likelihood function, and we then can use the mapping $w_i \to v_i$ to translate words from the vocabulary to a $d$-dimensional vector.

– To optimize the model's likelihood function, we can minimize the following loss function instead:

$$-\sum_{i=m+1}^{n-m} \log P(w_c \mid w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m})$$

We cannot directly find a solution for the above optimization problem. Instead, we employ a gradient descent method during a training phase to minimize the loss. An alternative approach is the modeling of the optimization problem with a single hidden layer network, as shown at the bottom of this page. Contrary to the skip-gram-model, the input is now an $2m$-hot vector with each surrounding word setting a component to $1/2m$. It then follows the same structure as with the skip-gram model, passing through a fully connected network without bias and activation function. The columns in the corresponding weight matrix represent vectors $\boldsymbol{v}_i$. The hidden layer comprises $d$ neurons and is followed by another fully connected network, again without bias and activation function. The columns in this matrix correspond to the vectors $\boldsymbol{u}_i$. A softmax classifier on the output layer is compared with the target one-hot vector of the center word.



- We can employ a similar self-supervised approach to train the model as with the skip-gram model. To train CBOW models using a large text corpus, we create the training, test, and validation datasets as follows:
  - We enumerate all windows of size $2m + 1$ in the corpus as with the skip-gram model.
  - For every window, however, we generate only one pair of surrounding words and center word. As an example, consider the window "the red **apple** tastes fine". The data sample is given now as ([the, red, tastes, fine], apple).
  - Optionally, we can sub-sample or exclude pairs with common terms (stop words), and lemmatize or tokenize words (e.g., stemming) to decrease vocabulary size similar as discussed with the skip-gram model.
- During training, we utilize pairs of surrounding words and center word to compute the loss function and make adjustments to the model weights, that is the vectors $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$. This process is iterated until the loss function reaches a sufficiently low value. The outcome is a vocabulary-to-vector mapping, represented as $\boldsymbol{v}_i$.
- To efficiently handle word pairs without using memory-intensive one-hot vectors, a `PyTorch` implementation can use the `Embedding` layer and the `CrossEntropyLoss` function, directly working with token IDs instead of vectors. The key difference in the model code, shown on the left, is that inputs are now vectors of token IDs (representing the surrounding words), rather than a scalar (representing one surrounding word). The model first maps all surrounding words to their embeddings and then averages them to incorporate the $1/2m$ input encoding from the previous page. The training process, depicted on the right side, is similar to the skip-gram model, except for the process to generate data batches

```python
class CBOWModel(nn.Module):
    def __init__(self, vocab_size: int, embedd_size: int):
        super().__init__()
        self.embeddings = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedd_size
        )
        self.linear = nn.Linear(
            in_features=embedd_size,
            out_features=vocab_size,
        )

    def forward(self, inputs):
        x = self.embeddings(inputs)
        x = x.mean(axis=1)
        x = self.linear(x)
        return x
```

```python
# sketch of training process
model = CBOWModel(vocab_size=len(vocab), embedd_size=100)
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()

for inputs, labels in batch_loader_cbow(text_corpus):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, labels)
    loss.backward()
    optimizer.step()
```

## 6.3.2 GloVe and fastText

- **GloVe (Global Vectors for Word Representation)** is a self-supervised embedding algorithm developed at Stanford in 2014. Like other methods, it considers word co-occurrences within context windows. However, it differs in its loss function, which relies on co-occurrence probability ratios:
  - Let $X_{ij}$ represent the number of occurrences of word $w_j$ within the context of word $w_i$, defined by a window around $w_i$. Then, $X_i = \sum_k X_{ik}$ denotes the total number of co-occurrences for word $w_i$. Define $P_{ij} = P(j|i) = X_{ij}/X_i$ as the probability of word $w_j$ appearing in the context of word $w_i$.
  - The initial idea is to examine the relationship between two words, $w_i$ and $w_j$, by analyzing the ratio $P_{ik}/P_{jk}$ in relation to a third word, $w_k$. If $w_i$ and $w_k$ are related while $w_j$, and $w_k$ are not, the ratio $P_{ik}/P_{jk}$ will be large. Conversely, if $w_j$, and $w_k$ are related but $w_i$ and $w_k$ are not, the ratio $P_{ik}/P_{jk}$ will be small. When both $w_i$ and $w_j$, are either related or unrelated to $w_k$, the ratio should be approximately 1. Here is the example for $w_i$="ice" and $w_j$="steam" from the original paper:

| | $w_k$="solid" | $w_k$="gas" | $w_k$="water" | $w_k$="fashion" |
|---|---|---|---|---|
| P(k\|$w_i$="ice") | $1.9 \times 10^{-4}$ | $6.6 \times 10^{-5}$ | $3.0 \times 10^{-3}$ | $1.7 \times 10^{-5}$ |
| P(k\|$w_j$="steam") | $2.2 \times 10^{-5}$ | $7.8 \times 10^{-4}$ | $2.2 \times 10^{-3}$ | $1.8 \times 10^{-5}$ |
| P(k\|$w_i$="ice") / P(k\|$w_j$="steam") | 8.9 | $8.5 \times 10^{-2}$ | 1.36 | 0.96 |

  - We introduce vectors $u_i$ to represent words in a $d$-dimensional space and vectors $v_k$ to represent context words in a $d$-dimensional space (adopting notation similar to word2vec, deviating from the original paper's notation). Additionally, we require biases $b_i$ for words and $c_j$ for context words. Using the relationships described earlier, we can build a cost function employing weighted least squares:

$$J = \sum_{i,j=1}^{n} f(X_{ij}) \cdot \left(u_i^T \cdot v_j + b_i + c_j - \log X_{ij}\right)^2$$

    The function $f(x) = \min((x/x_{max})\hat{}\alpha, 1)$ assigns weights to co-occurrences, giving higher weights to more frequent ones while avoiding excessive emphasis on very frequent co-occurrences. Similar to word2vec, we create pairs from the corpus and train the parameters to optimize the cost function. The resulting vectors $u_i$ represent word vectors. A detailed training algorithm discussion is omitted here due to its similarity to word2vec.

- **fastText** was developed by Facebook's AI Research (FAIR) team and first published in 2017. It improves word2vec algorithms in two ways:
  - fastText employs sub-word representations for center words and constructs word vectors by summing the vectors of their sub-words. In the context of a phrase like “the dog chases a cat” with “chases” as the center word, the context words (“the”, “dog”, “a”, and “cat”) maintain their full-word vectors $u_j$ and are not split into sub-words. However, the center word “chases” is broken into sub-words, for instance, of length 3, such as “<ch”, “cha”, “has”, “ase”, “ses”, and “es>”. The special characters “<“ and “>” mark the start and the end of the word allowing for the differentiation of prefixes, suffixes, and sub-words in the middle of a word. fastText considers sub-words ranging from length 3 to 6 within the annotated word (“<chases>”). Let's denote the set of sub-words for “chases” as $\mathbb{Z}$, and assign vector representations $z_g$ to each sub-word $g \in \mathbb{Z}$. The vector representation $v_i$ for “chases” is then calculated as the sum of its sub-word representations:

$$\boxed{v_i = \sum_{g \in \mathbb{Z}} z_g} \quad \xrightarrow{\text{skip-gram}} \quad \boxed{P(w_s \mid w_c) = \frac{e^{\boldsymbol{u}_s^T \boldsymbol{v}_c}}{\sum_{i \in \mathbb{T}} e^{\boldsymbol{u}_i^T \boldsymbol{v}_c}}}$$

    We then integrate this representation of the center word into the skip-gram model, as illustrated on the right side, and proceed to optimize the loss function as described in the word2vec part. This process yields vectors $z_g$ for the sub-words, enabling us to build of word embeddings for any words, including those not present in the corpus.
  - fastText expedites training by subsampling frequent co-occurrences and utilizing a negative sampling technique. Instead of comparing the softmax result to the one-hot target, both the center (sub-)word and the surrounding word are mapped to their respective $d$-dimensional representations, and their vectors are compared using dot products. In addition to the positive sample, $\boldsymbol{m}$ **negative samples** (words not found in the center word's surroundings) are included to expedite training iterations and enhance weight matrix updates. Detailed information can be found in the relevant research papers. As result, fastText can reduce the computational complexity and produce embeddings faster than the original word2vec approach
  - The sub-word approach can be customized from fixed sizes (3 to 6) to variable lengths using methods like byte pair encoding (BPE) or word pieces algorithms. This results in more concise vocabularies and enables the encoding of diverse Unicode words, as previously discussed in the tokenization section. For instance, English has approximately 3 x 10^8 possible 6-grams, and it may not be efficient to store representations for all of them. By utilizing BPE and word pieces, we can establish an upper limit on the storage required for embeddings.

# 6.3.3 Practical Applications

The code on the right side illustrates how to use the `gensim` package for learning and utilizing embeddings:

1) Gensim can train models with word2vec or fastText, as the code shows. For better embeddings, use large collections of sentences. The movie reviews example may not produce reliable embeddings.
2) The `gensim` package includes several pre-trained embeddings, such as GloVe vectors, which provide general-purpose embeddings mainly for English. Because these models were trained on diverse topics, they may not give the best results for a specific collection.
3) Embeddings let us study word relationships. We create vector representations for words, for example "cat." We then compare these vectors to find the most similar words. Similarity comes from shared contexts, so "dog" is often very similar to "cat." We can also do arithmetic with vectors, for example "Paris + Germany - France = Berlin." Another useful task is finding the odd word out in a list, the word that does not belong with the others.

`spaCy`'s models have vector embeddings (`token.vector`) in their pipelines. In the base models, `spaCy` employs floret, a variation of fastText that generates more space-efficient vector tables. Models like "en_core_web_trf" utilizes transformers (BERT based) for embeddings.

```
1) Train your own model with gensim
from gensim.models import Word2Vec, FastText
from nltk.corpus import movie_reviews
sentences = movie_reviews.sents()

# train a word2vec CBOW model with window 5, 25 dimensions
model = Word2Vec(sentences, vector_size=25, window=5)

# train a fastText model with window 5, 25 dimensions
model = FastText(vector_size=25, window=5, min_count=1)
model.build_vocab(corpus_iterable=sentences)
model.train(corpus_iterable=sentences,
            total_examples=len(sentences), epochs=10)

# access learned vectors
word_vectors = model.wv


2) Use a pre-trained model with gensim
import gensim.downloader as api
word_vectors = api.load('glove-wiki-gigaword-100')


3) Analyze token relationships (two variants)
word_vectors['cat']
word_vectors.get_vector('cat')
↳ [0.23, 0.28, 0.63, -0.59, -0.59, 0.63, 0.24, -0.14, …]

word_vectors.most_similar('cat')
↳ [('dog', 0.88), ('rabbit', 0.74), ('cats', 0.73), …]

word_vectors.most_similar(positive=['paris','germany'],
negative=['france'])
↳ [('berlin', 0.88), ('frankfurt', 0.80), ('vienna', 0.77), ...]

paris = word_vectors.get_vector('paris')
germany = word_vectors.get_vector('germany')
france = word_vectors.get_vector('france')
word_vectors.similar_by_vector(paris+germany-france)
↳ [('berlin', 0.88), ('frankfurt', 0.80), ('vienna', 0.77), ...]

word_vectors.doesnt_match("bird dog cat town".split())
↳ town
```

- An effective demonstration of embeddings is provided in the figure below, along with the code for generating it. We employ a pre-trained GloVe set with 100 dimensions, trained on Wikipedia text. Utilizing 25 words, we map them to the GloVe vector space. To enhance visualization, we project these vectors into a 2-dimensional space using PCA:
  - In the previous example, we employed the `doesnt_match` function on the words “bird dog cat town”. In the visualization, it is now evident that the word “town” is distinctly separated from the animal-related words. This demonstrates how we can identify words that don't belong to a specific group.
  - We can also identify clusters of words, such as those related to animals, city names, or words associated with people. Instead of manually constructing relationships from dictionaries, we can now learn these associations between words automatically. It is important to note that these associations do not necessarily represent semantic relationships but indicate whether two words tend to appear in similar contexts. For example, in our previous example, “dog” was the most similar word to “cat”. This does not mean that cats are similar to dogs, but rather that they are frequently discussed in similar contexts, such as when people talk about their pets.
- A larger online visualization for embedding is here: [https://projector.tensorflow.org](https://projector.tensorflow.org)

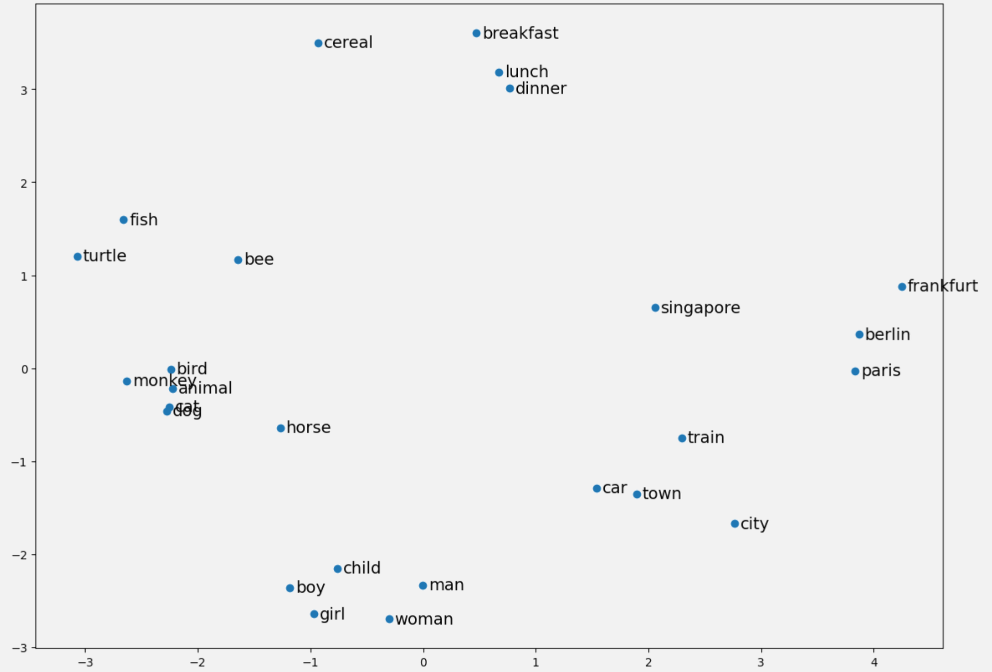
```python
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import gensim.downloader

# get the glove-wiki-gigaword-100 word vectors
word_vectors = gensim.downloader.load('glove-wiki-gigaword-100')

# get vectors for terms
words = "animal bird dog cat horse fish bee … dinner lunch".split()
vectors = word_vectors[words]

# apply a PCA to map to 2 dimensions
pca = PCA(n_components=2)
result = pca.fit_transform(vectors)

# create a scatter plot of the projection
plt.figure(figsize=(14,10))
plt.scatter(result[:, 0], result[:, 1])
for i, word in enumerate(words):
    plt.annotate(word, xy=(result[i, 0]+0.05, result[i, 1]-0.05),
                 fontsize=14)
plt.show()
```



# 6.4 From Word to Sentence Embeddings

- Word embeddings give rich semantic representations of words, but they are not directly suitable for text retrieval because a text is represented by hundreds or even thousands of separate vectors. To make retrieval possible, we must summarize these word-level embeddings into a single compact vector, similar to how TFIDF summarizes term importance in a document. This summarization, often called **pooling**, produces a fixed-length vector that captures the overall meaning of a sentence, paragraph, or document.
- Two common pooling methods are **average pooling** and **max pooling:**
  - **Average pooling** (also called mean pooling) computes the element-wise average across all token embeddings in the sequence. Given $m$ token embeddings $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_m\}$ where each $\boldsymbol{x}_i \in \mathbb{R}^d$ represents a token in a $d$-dimensional space, average pooling produces:

    $$pool_{avg}(\boldsymbol{X}) = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}_i$$

    This method provides a balanced, holistic representation by ensuring every token contributes equally to the final embedding. It captures the overall semantic tone and general context of the text, making it particularly effective for tasks like semantic similarity and document retrieval where comprehensive understanding matters.
  - **Max pooling** selects the maximum value across each dimension of the token embeddings:

    $$pool_{max}(\boldsymbol{X}) = \begin{bmatrix} \max\limits_{1 \le i \le m} x_{i,1} \\ \max\limits_{1 \le i \le m} x_{i,2} \\ \vdots \\ \max\limits_{1 \le i \le m} x_{i,d} \end{bmatrix}$$

    Where the maximum is taken for each of the $d$ dimensions separately. This highlights the most salient features in the text, capturing dominant signals that often correspond to keywords or other critical information. Max pooling is useful when particular tokens, such as "amazing" in sentiment analysis or domain-specific technical terms, carry disproportionate semantic weight.

- After documents and queries are converted into single vectors, we need a function to measure semantic fit. **Cosine similarity is the common choice**. It measures the angle between vectors, so it ignores vector length and captures only directional alignment. This works well for comparing word embeddings because those vectors are often not normalized; a raw dot product can be dominated by one unusually large value in a single dimension.
- In practice, we normalize pooled embedding vectors when indexing documents and queries to convert cosine similarity into a dot product and greatly reduce computation during search. Let $\widehat{\boldsymbol{d}}$ denote the normalized pooled document embedding and $\widehat{\boldsymbol{q}}$ the normalized pooled query embedding. Then:

$$\widehat{\boldsymbol{d}} = \frac{\boldsymbol{d}}{\|\boldsymbol{d}\|} \qquad \widehat{\boldsymbol{q}} = \frac{\boldsymbol{q}}{\|\boldsymbol{q}\|} \qquad sim_{cos}(Q, D) = \frac{\boldsymbol{q} \cdot \boldsymbol{d}}{\|\boldsymbol{q}\| \cdot \|\boldsymbol{d}\|} = \widehat{\boldsymbol{q}} \cdot \widehat{\boldsymbol{d}}$$

- This framework enables efficient text retrieval: documents are embedded, pooled and normalized once during indexing, queries are embedded, pooled, and normalized at search time, and the dot-product ranks documents by semantic relevance (implementing a cosine measure on the original pooled embeddings).
- To understand pooling mechanisms, consider how they work in the **latent space of embeddings**.
  - **Average pooling** computes a centroid in the embedding space. Each token embedding is a point in $d$-dimensional space, and averaging gives the geometric center of those points. When comparing a query to a document, we measure the angle between the query centroid and the document centroid using cosine similarity. Let $\widehat{\boldsymbol{q}}$ and $\widehat{\boldsymbol{d}}$ be the normalized centroids of the word embeddings for query $Q$ and document $D$. They are obtained as follows:

$$\widehat{\boldsymbol{q}} = \frac{\overline{\boldsymbol{q}}}{\|\overline{\boldsymbol{q}}\|} = \frac{\frac{\boldsymbol{1}}{\boldsymbol{N}} \sum_{i=1}^{N} \boldsymbol{q}_i}{\|\overline{\boldsymbol{q}}\|} \qquad \widehat{\boldsymbol{d}} = \frac{\overline{\boldsymbol{d}}}{\|\overline{\boldsymbol{d}}\|} = \frac{\frac{\boldsymbol{1}}{\boldsymbol{M}} \sum_{j=1}^{M} \boldsymbol{d}_j}{\|\overline{\boldsymbol{d}}\|}$$

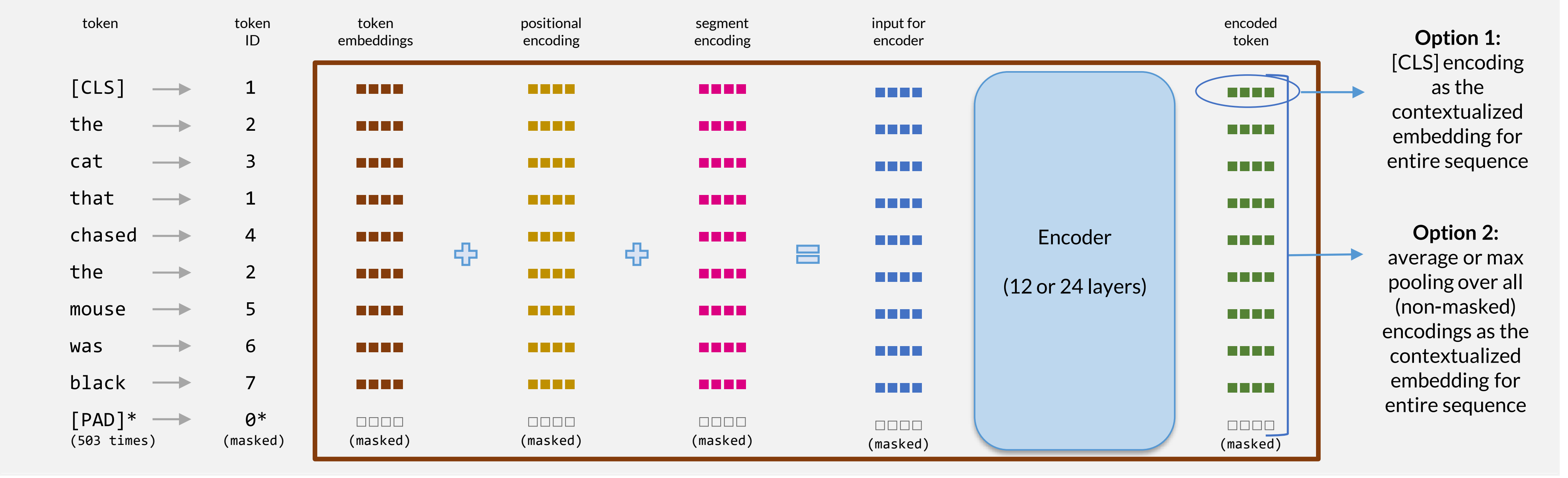
    - Using cosine similarity, we obtain the score between $Q$ and $D$ as follows (regrouping the sums):

$$sim(Q, D) = \widehat{\boldsymbol{q}} \cdot \widehat{\boldsymbol{d}} = \frac{1}{\|\overline{\boldsymbol{q}}\| \cdot \|\overline{\boldsymbol{d}}\|} \cdot \frac{1}{N \cdot M} \sum_{i=1}^{N} \sum_{j=1}^{M} \boldsymbol{q}_i \cdot \boldsymbol{d}_j$$

- **Average pooling (continued)**
  - From the last equation, average pooling compares every token in the query with every token in the document using the dot product, then normalizes and averages the results. Pairs of semantically similar tokens increase the sum, while dissimilar pairs decrease it. Summing over hundreds or thousands of tokens can dilute important information. For example, averaging 500 token embeddings gives each token only 0.2% of the final score, which can wash out distinctive features that distinguish documents. This dilution grows with document length, so the aggregated embedding tends to reflect generic, averaged meaning rather than nuanced content.
  - What about stop words? With average pooling, stop words can dominate the pooled vector because they appear often and pull the average toward directions in the embedding space that carry little meaning. **Stop words typically account for 30% to 50% of words** in natural language text. This large share reflects their role as grammatical building blocks, even though they add little semantic content. For example, in **typical search queries about 12.9% of tokens are stop words**, while in **question queries about 39% are stop words**. If a document has 40% stop words and a query has 30%, then content tokens make up 60% of the document and 70% of the query. Only $0.6 * 0.7 = 0.42$, so just 42 percent of token pairs are between content tokens; the rest include at least one stop word. These stop words dilute the result and make documents harder to distinguish. For that reason, the standard approach is to **remove stop words before applying the pooling function**.
- With **max pooling**, each embedding dimension takes the maximum value across all token embeddings. After the final normalization step, dominant dimensions show large values while less important dimensions are near zero. Max pooling has different challenges than average pooling. Although it preserves strong features, it discards contextual relationships between tokens by keeping only the maximum value for each dimension. For a document with 1,000 tokens, max pooling may retain information from only a small fraction of tokens (for example, 10 to 50), and it can miss important semantic nuances that come from token interactions, phrase structures, or syntactic dependencies. This makes it less sensitive to the broader context and more focused on highly salient individual token features.
  - Common words that occur in many contexts, like the stop words "the," "of," and "and," tend to have embeddings with small magnitudes near zero. Because they appear in diverse settings, their vectors are pulled in many directions during training and end up shorter. With max pooling, stop words rarely produce strong activations in any latent dimension, so their impact is small. They still add computational cost, so it is generally more efficient to ignore stop words when using max pooling, particularly in long documents, without significantly affecting the quality of the resulting embeddings.

- **Limitations of Statistical Pooling**
  - Average and max pooling are simple, effective ways to compress token-level embeddings into a single vector. They are statistical rather than semantic methods. Average pooling treats every token the same, which can wash out distinctions between important and less relevant words. Max pooling preserves dominant features but discards contextual relationships between tokens, for example the link between adjectives and nouns like "red car" versus "white car". Both methods fail to encode word order or syntactic relationships and cannot capture long-range dependencies across sentences.
  - For example, in the sentence "The cat that chased the mouse was black," average pooling treats each token equally and ignores the connection between "cat" and "chased the mouse", while max pooling focuses on a few strong dimensions and can miss important interactions.
- These limitations motivated the development of **contextual embeddings**, where each token's vector is influenced by surrounding words. This allows embeddings to represent not just the word itself, but its role and meaning in the specific context of the sentence or document.
  - The paper **Attention Is All You Need** (2017) introduced the Transformer, a new neural network that relies entirely on self-attention instead of recurrence or convolution for sequence modeling. The Transformer uses multi-head attention to capture relationships between words and positional encoding to preserve word order. This design allows parallel processing, which greatly improves training speed and scalability. The model achieved state-of-the-art results in machine translation and became the foundation for many later models, including **BERT** and **GPT**, transforming natural language processing.
  - Both GPT and BERT use the Transformer architecture but differ in design and training goals. **GPT (Generative Pre-trained Transformer)** uses only the decoder and is trained with a unidirectional left-to-right language modeling objective, making it well suited for text generation. **BERT (Bidirectional Encoder Representations from Transformers)** uses only the encoder and is trained with masked language modeling and next sentence prediction, which lets it capture context from both directions. **GPT focuses on natural language generation, while BERT focuses on natural language understanding.**
  - The initial approach using BERT and the encoded [CLS] token did not yield the expected results. However, the introduction of SBERT (Sentence-BERT), with its bi-encoder and cross-encoder architectures, marked a significant breakthrough, producing far better results than pooled word embeddings. Bi-encoders generate contextualized embeddings for sequences, enabling effective retrieval of semantically similar passages. Cross-encoders, on the other hand, compare pairs of sequences directly and yield a score, making them ideal for reranking results.

- To create more semantically rich embeddings, the first idea was to use BERT to generate contextualized embeddings:
  - BERT uses a WordPiece tokenizer to split words into subword units, for example "**playing"** becomes "**play**" + "**##ing**", so the model can handle rare or unknown words. Each token is then converted into a **768-dimensional embedding** for BERT-Base (1024 dimensions for BERT-Large). **Positional encoding** record each token's position in the sequence to preserve order, while **segment encoding** mark whether a token belongs to sentence A or sentence B (using [SEP] as a seperator). They are added to the token embeddings to form the input vectors for the encoder.
  - BERT-Base has 12 Transformer encoder blocks with 12 attention heads per block. BERT-Large has 24 blocks with 16 heads per block. Inside each block, **bidirectional self-attention lets each token attend to all other tokens at once**, so the model captures context from both left and right. The encoder produces a **768-dimensional contextual vector for each token** (1024 dimensions for BERT-Large), used for tasks like classification, question answering, and named entity recognition.
  - BERT accepts up to **512 tokens per input**, including the special tokens **[CLS] (classification)** and **[SEP] (separator)**.Truncating, padding and masking enable support for any sequence length. The [CLS] token is always at the start of the sequence. Its final hidden state, the first output vector, is usually used as a contextual embedding for the entire sequence. Alternatively, we can pool over all encoded outputs, for example with average pooling or max pooling.



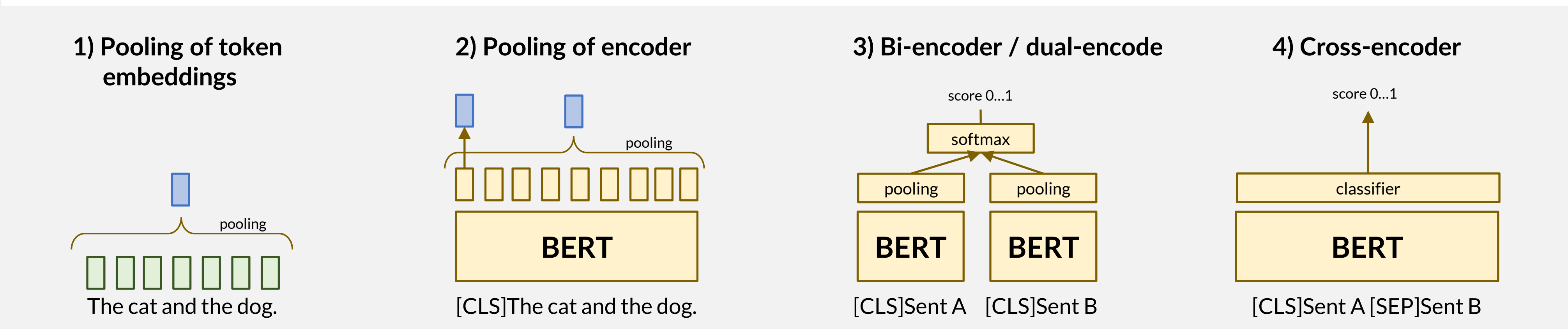
- **Sentence-BERT (SBERT):** Siamese Architecture for Effective Sentence Embeddings
  - Sentence-BERT (SBERT) improves the creation of meaningful sentence embeddings. It uses a so-called **bi-encoder architecture**: two sentences are passed separately through the same BERT-based encoder to produce fixed-size embeddings. The encoder output can either be the [CLS] token encoding or the mean of all token encodings. SBERT is fine-tuned on datasets, where it learns to label sentence pairs as entailing, contradicting, or neutral. Training uses similarity-based losses, typically contrastive, triplet, or multiple negatives ranking loss, with cosine similarity so that semantically similar sentences have closer embeddings and dissimilar ones are farther apart.
  - The advantage of the bi-encoder architecture is that we can preprocess every passage into an embedding vector that captures the deep contextual meaning learned during training. A query is encoded the same way. Sentence Transformers usually output normalized embeddings, so cosine similarity can be replaced by the simpler dot product. This also prevents implementations from choosing a different similarity measure that would conflict with the training setup. In geometric terms, normalization maps all vectors onto a hypersphere so that the angle between vectors becomes the only factor determining their similarity, effectively removing the influence of vector magnitude and ensuring that comparisons are based purely on semantic alignment rather than scale differences.
- **Qwen Embedding Models:** State-of-the-Art Multilingual Text Representation
  - Qwen3 embedding models represents the latest advancement in text embedding technology, specifically designed for comprehensive text embedding and ranking tasks. Built upon the dense foundational models of the Qwen3 series, these models provide various sizes (0.6B, 4B, and 8B parameters) with exceptional multilingual capabilities.
  - Qwen3 embedding models use a **dual encoder design** built on the Qwen3 foundation model. The base model is a **decoder-based transformer** that processes each text segment and produces dense semantic embeddings. These embeddings are taken from the **hidden state of the final [EOS] token**, which acts as a summary of the input text. This in contrast to the encoder based BERT embedding models that start with the [CLS] token.
  - Qwen3 embeddings use **LoRA (Low-Rank Adaptation)** to efficiently fine-tune the model for embedding tasks. LoRA adds small, trainable low-rank matrices to the model's weight matrices, letting it adapt to new tasks with very few extra parameters. This preserves the base model's pretrained knowledge while improving its ability to capture fine-grained semantic relationships in text. The dual encoder is trained with similarity-based objectives similar to SEBRT (contrastive loss, cosine similarity).
  - The 8B Qwen3 embedding model achieved No.1 ranking on the MTEB multilingual leaderboard (June 2025). The models support over 100 languages and provide flexible vector definitions across all dimensions. It supports a context window of up to 32k tokens which typically includes instructions for downstream tasks.

- A **cross-encoder** is a neural model used in text retrieval and ranking systems. It scores how relevant a candidate text, such as a passage or document, is to a query. Instead of relying on precomputed representations, a cross-encoder processes the **query and the document together in the same transformer network**. This lets the model learn and use fine-grained relationships between query and text at the token level, producing very accurate relevance judgments.
  - To see why cross-encoders matter, compare them with the more efficient bi-encoder (or dual-encoder). In a bi-encoder, the query and each document are turned **separately into dense vectors**. Relevance is then measured with a simple similarity function such as cosine similarity or dot product. Because document embeddings can be precomputed and indexed, bi-encoders are very fast and scalable, making them **ideal for searching huge text collections**. However, encoding the query and document independently limits the model's ability to capture subtle, context dependent relationships between specific words or phrases.
  - The cross-encoder avoids that limitation by **joining the query and document into one input sequence**, often separated by a special token like [SEP], and sending the combined text through a transformer model such as BERT. The model attends to all tokens from both the query and the document at once, allowing it to capture precise semantic interactions. The final representation, usually taken from the [CLS] token, goes through a classifier or regression layer to produce a **relevance score**. Because each query-document pair is processed together, cross-encoders require **much more computation**, especially when ranking thousands or millions of documents.
  - Therefore, **cross-encoders are used for re-ranking** rather than direct retrieval. A common workflow is to first use a bi-encoder or BM25 to quickly retrieve a small set of candidate documents, and then apply a cross-encoder to re-evaluate and reorder those candidates with greater accuracy. This hybrid method balances speed and precision.
- The **Qwen3-Reranker** models from Alibaba are cross-encoders built for precise text re-ranking. Built on the Qwen3 foundation model, they score relevance by taking a query and candidate text together. Unlike Qwen3-Embedding models, which use a dual-encoder design for fast large-scale retrieval, the re-rankers output a direct relevance score. They support task-optimized instructions and very long context windows up to 32,000 tokens. Batch inference lets them score multiple documents at once. A typical input sequence looks as follows:

```
      [instruction] {query} [candidate document]
```

- They are built on the Qwen3 transformer architecture and support long contexts and more than 100 languages, so they can handle multilingual and long-document ranking. The models are trained on supervised pairwise relevance data and are optimized for precise reranking rather than general embedding.

- Summary: Text retrieval and semantic similarity models can use different architectures, each balancing efficiency and accuracy in its own way. The figure at the bottom high-lights each architecture (here with BERT):
  1) **Pooling of word embeddings** represents a sentence by averaging or max-pooling its word embeddings, for example from word2vec or GloVe. This creates a single fixed-length vector that roughly captures the sentence meaning. It is fast and easy to implement, but it discards word order and detailed context, so it performs poorly on complex language tasks.
  2) **Pooling of [CLS] token from encoder** uses the final hidden state of the special [CLS] token with a transformer based encoder such as BERT. Alternatively, we can pool the final hidden states over all tokens. Both approaches capture sentence-level meaning but still encode each text independently, producing contextualized vectors for fast retrieval. Because this architecture has not been fine-tuned for semantic similarity tasks, it is more often used for later classification tasks than for semantic search.
  3) **A bi-encoder /dual-encoder** encodes the query and the document separately and learns from labeled examples how to judge relevance between them. This produces contextualized embeddings that can be used for semantic search with dot-product. Retrieval is fast because document embeddings can be precomputed and stored in a vector index. However, since the query and the document do not attend to each other during encoding, some semantic nuances are lost.
  4) **Cross-encoder** combines the **query and document into one input sequence**, allowing the model to jointly attend to all tokens. This enables precise, token-level interaction modeling and leads to superior ranking accuracy. The trade-off is efficiency: it must process every query–document pair, making it much slower and typically reserved for **reranking** a small set of top candidates retrieved by a bi-encoder.

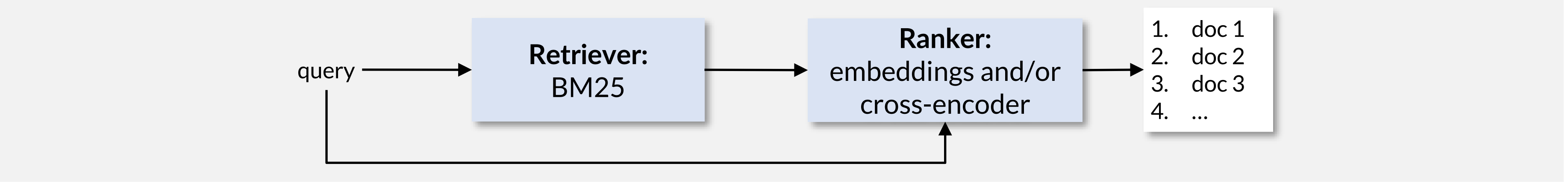


| Feature / Approach | Word Embeddings + Pooling | BERT (Encoder) | SBERT (Bi-encoder) | Qwen3 (Decoder) |
|---|---|---|---|---|
| **Architecture** | Pre-trained word vectors; pooled via mean/max/sum | Transformer encoder; uses [CLS] token at the beginning for sentence representation; 512 token window | Transformer encoder; bi-encoder for learning; [CLS] token at the beginning for sentence representation; 512 token window | Transformer decoder; dual-encoder for learning; uses [EOS] token at the end for sentence representation; 32k token window |
| **Contextualization** | ❌ Non-contextual; each word independent | ✅ Contextualized within sentence | ✅ Contextualized + fine-tuned for semantic similarity | ✅ Strong contextualization; excels with long texts |
| **Computational Efficiency** | ✅ Very fast and lightweight | ⚠️ Medium-heavy; slower on large corpora | ⚠️ Medium-heavy; slower on large corpora | ❌ High resource requirements; slow for large datasets |
| **Sentence Representation Quality** | ⚠️ Basic, depends on pooling | ✅ Good, but [CLS] token may not fully capture meaning | ✅ High-quality semantic embeddings | ✅ Very strong, especially for reasoning and long-text embeddings |
| **Fine-tuning Needs** | ❌ Typically none | ⚠️ Often needed for optimal performance | ⚠️ Usually fine-tuned for domain or task | ✅ Optional, but fine-tuning can improve domain adaptation |
| **Ideal Use Cases** | Clustering, simple retrieval, word relationships | Text classification, sentiment analysis | Semantic search, duplicate detection | RAG, QA, information retrieval, complex reasoning |
| **Multilingual Support** | ⚠️ Depends on embeddings used | ⚠️ Requires multilingual BERT variants | ✅ Available in multilingual SBERT models | ✅ Strong multilingual capability |
| **Strengths** | Fast, simple, lightweight | Rich contextualization | Optimized for semantic similarity, efficient for retrieval | Excels in long-text understanding, reasoning, and RAG |
| **Limitations** | No context, loses syntactic nuances | Computationally heavy, [CLS] may be insufficient | May degrade without domain-specific fine-tuning | Computationally expensive, overkill for simple tasks |

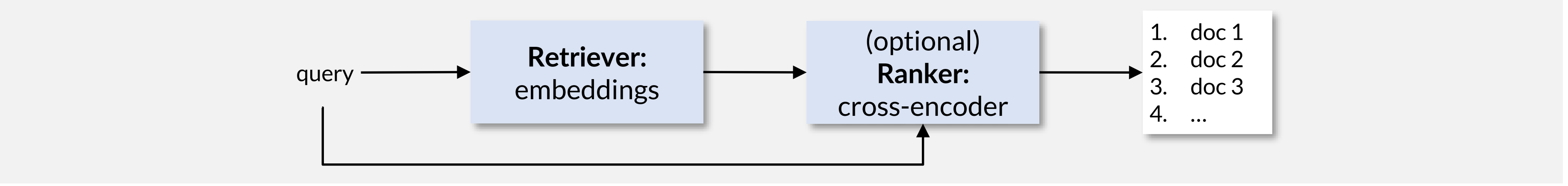
## 6.5 Semantic Search for Documents

- Modern information retrieval systems increasingly use semantic search to find documents by **shared meaning rather than by shared words**. This uses embeddings, which convert text into dense, continuous vectors so that similar meanings are close together. However, semantic methods can be computationally expensive, and purely lexical methods like BM25 often miss subtle relationships between words. The most effective pipelines combine the speed of classical search, the contextual insight of embeddings, and the precision of re-rankers into a single system.
  - The process starts with **document preprocessing and chunking**. Since embedding models work best with limited input lengths, long documents are split into overlapping segments, typically about 200 to 300 tokens for BERT-based models such as SBERT, and 500 to 1,000 tokens for larger decoder models like Qwen3. Each chunk is then embedded as a high-dimensional vector by a sentence or document encoder, creating a searchable semantic index.
  - When a user submits a query, the system first uses **BM25 for lexical retrieval**. This traditional ranking function quickly finds a small set of possibly relevant documents by matching words and counting how often they appear. BM25 does not understand meaning, but it is fast and keeps processing focused on the most promising candidates.
  - Next comes **semantic retrieval**. The query is embedded in the same vector space as the document chunks, and similarity, usually measured by cosine distance, is computed between the query embedding and the stored chunk embeddings. This returns the top-k chunks that are most semantically similar, finding passages that mean the same thing as the query even when they use different words.
  - To refine the results, a **re-ranker**, often a cross-encoder model, is applied to the top candidates. Unlike embedding models that encode query and document separately, a cross-encoder processes them together for improved semantic analysis. The re-ranker gives more accurate relevance scores and improves the final ranking of results.
  - Finally, the system can return the **most relevant chunks** for direct answer extraction, or **combine chunk scores to rank whole documents**. By combining these complementary techniques, the pipeline achieves both scalability and semantic depth. BM25 narrows the candidates, embeddings capture meaning, and re-rankers add precision. Together, they form the backbone of modern semantic search.
- For small document collections, systems often skip BM25 lexical retrieval and start with semantic retrieval over embeddings to increase recall. A re-ranker is usually needed to raise precision and to sort the retrieved documents or chunks by how well they answer the query. Another option is to use embeddings instead of an expensive re-ranker to quickly re-rank BM25 results.

- **Retriever-Ranker Pipeline Options**
  - The retriever-ranker has two main parts: the retriever finds a set of candidate documents in the index that match the query (focused on recall). The ranker assigns a relevance score to each candidate and returns the final ordered results (focused on precision).
  - **Option 1:** We can use a BM25 retriever to get an initial set of candidate documents or chunks. To broaden the search, we can expand the query with additional keywords from dictionaries or from embeddings. Then use embeddings or a cross-encoder to compute semantic similarity scores between the query and each candidate. The result is an updated top-k list of documents for the user.

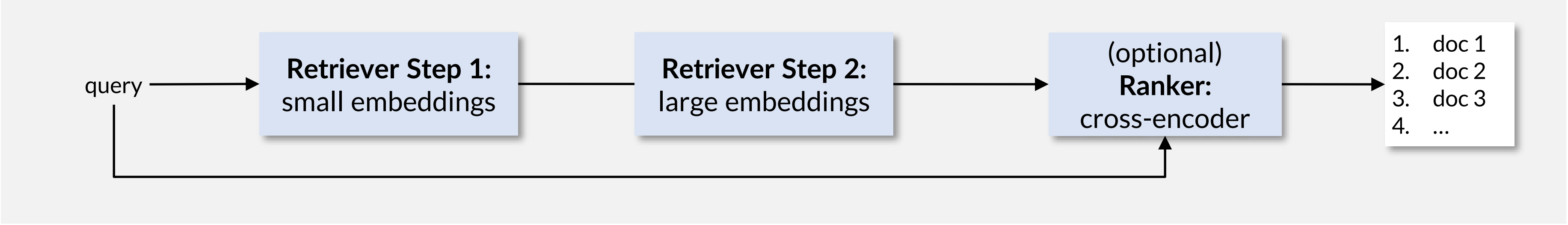



  - **Option 2:** Alternatively, we can replace BM25 with semantic retrieval using embeddings. Compute embeddings for documents and queries and store them in a high-dimensional vector index. At query time, select chunks whose embeddings are close to the query embedding, for example by cosine similarity. Optionally, re-rank the top candidates with a cross-encoder ranker for better precision.




- We will discuss more semantic search pipelines in Chapter 8. The next pages provides a summary overview over classical and semantic retrieval methods.

- **Matryoshka / MRL Embeddings**
  - Recent research introduced **Matryoshka Representation Learning (MRL)**, also called Matryoshka embeddings, to make semantic retrieval more flexible and efficient. Traditional embedding models produce a single vector of a fixed size for each text piece, often 384, 768, or 1,024 dimensions. These vectors are powerful but costly in computation and storage, especially for large vector databases that must store and compare billions of embeddings. MRL builds nested representations that **keep their semantic meaning even when cut down to lower dimensions**, like Russian Matryoshka dolls where each smaller layer fits inside a larger one.
  - The key idea is to train the model so partial embeddings remain useful. Instead of optimizing only the full vector, MRL **adds extra loss terms at several truncation levels**, for example the first 128, 256, and 512 dimensions. This forces the model to spread semantic information across the whole vector in a structured way. As a result, the leading dimensions capture the most important semantic features, while the later dimensions refine the representation for finer distinctions. This is similar to Latent Semantic Analysis. Low-dimensional projections capture the main concepts or topics, while extra dimensions capture finer variations and context. In both methods, the representation space is arranged hierarchically: the first components describe broad semantic structure and later components add detail. This hierarchy makes MRL embeddings behave like compressed versions of the same meaning space, preserving coherence when they are made smaller.
  - This property lets you **balance accuracy and efficiency**. For high-recall searches over large collections, smaller sub-vectors can run fast approximate queries (retriever step 1), cutting memory use and computation. After a small candidate set is found, full-dimension embeddings can re-rank results more precisely (retriever step 2), matching large-model performance at a fraction of the cost. MRL embeddings therefore enable multi-stage retrieval in a single model and remove the need for separate encoders of different sizes.




  - In the next chapter, we study vector search in more detail and examine ways to speed up searches through large sets of vectors.

| Approach | Core Idea | Typical Method or Model | Semantic Awareness | Context Sensitivity | Efficiency | Primary Use Cases |
|---|---|---|---|---|---|---|
| **1. Bag-of-Words (BoW)** | Represent text as a frequency vector of unique words (vocabulary-based). | TF-IDF | Low – counts only co-occurrence | None – ignores order and meaning | High | Text classification, document retrieval (early IR) |
| **2. Linguistic Normalization** | Reduce word form variations to a common root. | Stemming, Lemmatization | Low – focuses on morphology, not semantics | None | High | Preprocessing for BoW and traditional models |
| **3. Latent Semantic Indexing (LSI)** | Project BoW into a lower-dimensional latent space capturing co-occurrence structure. | Singular Value Decomposition (SVD) | Moderate – captures word co-usage patterns | Weak – global context only | Moderate | Information retrieval, topic grouping |
| **4. Neural Word Embeddings** | Learn distributed vector representations for words via prediction tasks. | word2vec, GloVe, fastText | High – captures local semantic relationships | Limited – same word vector regardless of context | High | Similarity, clustering, analogy tasks |
| **5. Sentence Embeddings (Pooling-based)** | Aggregate token embeddings to represent an entire sentence. | Mean / Max Pooling on BERT or word2vec outputs | High – encodes overall semantics | Partial – depends on pooled tokens | High | Semantic similarity, clustering |
| **6. Bi-Encoder (Sentence Transformers)** | Encode each sentence independently using a shared encoder; compare via cosine similarity. | SBERT, Sentence-Transformers, Qwen3-Embedding | Very High – learned semantic alignment | High – contextual, sentence-level | High | Semantic search, retrieval, clustering, RAG retrieval |
| **7. Cross-Encoder** | Jointly encode sentence pairs for direct comparison. | BERT Cross-Encoder, Qwen3-Reranker | Very High | Very High (full self-attention between sentences) | Low | Reranking, QA, paraphrase detection |
| **8. Hybrid Architectures** | Combine dense bi-encoder retrieval with cross-encoder reranking. | Bi-Encoder + Cross-Encoder Pipeline | Very High | Very High | Moderate | Semantic search, question answering |
| **9. Matryoshka / MRL Embeddings** | Train embeddings that can be truncated to variable dimensions while preserving meaning. | Qwen3-Embedding, Matryoshka Representation Learning | Very High | High | Very High | Adaptive retrieval, scalable deployment |

# 6.6 Literature and Links


**Papers**

- Susan T. Dumais (2005). **Latent Semantic Analysis.** Annual Review of Information Science and Technology. https://doi.org/10.1002%2Faris.1440380105
- S. Deerwester, et. al, **Improving Information Retrieval with Latent Semantic Indexing**, Proceedings of the 51st Annual Meeting of the American Society for Information Science 25, 1988.
- T. Mikolov, et. al. **Distributed Representations of Words and Phrases and their Compositionality.** 2013. https://doi.org/10.48550/arXiv.1310.4546
- T. Mikolov, et. al. **Efficient Estimation of Word Representations in Vector Space**. 2013. https://arxiv.org/pdf/1301.3781.pdf
- P. Bojanowski, et. al. **Enriching Word Vectors with Subword Information**. 2017. https://aclanthology.org/Q17-1010.pdf
- J. Pennington, R. Socher, C. Manning. **GloVe: Global Vectors for Word Representation.** Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014. https://aclanthology.org/D14-1162
- A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin: **Attention Is All You Need**. CoRR abs/1706.03762 (2017). https://arxiv.org/pdf/1706.03762
- T. Wolf et. al. **Transformers: State-of-the-Art Natural Language Processing,** Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, 2020. https://www.aclweb.org/anthology/2020.emnlp-demos.6
- N. Reimers, I. Gurevych. **Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks.** Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), 2019. https://aclanthology.org/D19-1410
- A. Kusupati, V. Ramanujan, T. Xie, A. T. Suresh, T. Simons, S. Kumar. **Matryoshka Representation Learning.** In Advances in Neural Information Processing Systems (NeurIPS 2022). https://arxiv.org/abs/2205.13147


**Implementations**

- Models: **Huggingface, the AI community building the future.** https://huggingface.co
- Transformers: **State-of-the-art Machine Learning.** https://github.com/huggingface/transformers, https://huggingface.co/docs/transformers/index
- GloVe: **Global Vectors for Word Representation.** https://github.com/stanfordnlp/GloVe
- Qwen3: https://github.com/QwenLM/Qwen3 (Technical reports: https://arxiv.org/pdf/2506.05176, https://arxiv.org/pdf/2505.09388)