

# Multimedia Retrieval

## Chapter 4: Advanced Text Retrieval

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

[4.1 Introduction](#)

[4.2 Chunking Text](#)

[4.3 Tokenization Revisted](#)

[4.4 Lemmatization and Linguistic Transformation](#)

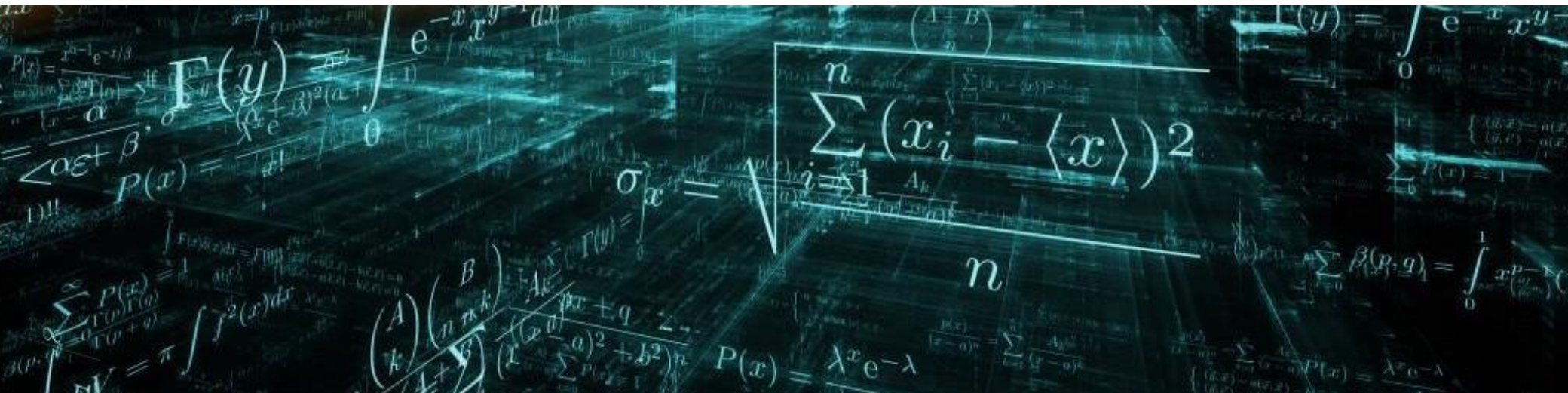
[4.5 Part of Speech](#)

[4.6 Latent Semantic Analysis](#)

[4.7 Embeddings](#)

[4.8 Text Classification](#)

[4.9 Literature and Links](#)



## 4.1 Introduction

- In this chapter, we enhance classical models by incorporating advanced techniques like tokenization, embeddings, natural language processing (NLP), and applying machine learning and generative AI methods.
  - We begin by revisiting document splitting techniques, exploring common methods, and examining their functionality within retrieval models.
  - In the previous chapter, we generated tokens from words and applied the Porter stemmer for English. In this chapter, we explore alternative tokenization methods, including sub-word tokens and phrases (n-grams).
  - We also delve into fundamental NLP techniques, such as part-of-speech tagging and chunking, which construct a tree structure based on language grammar.
  - Tokenization transforms documents into high-dimensional vectors, resulting in sparse document-term matrices. Latent semantic indexing use dimensionality reduction techniques to obtain a more concise document vector.
  - Modern AI approaches employ embeddings, notably through neural networks found and more recently with encoders (part of the transformer model) from large language models.
  - Text classification categorizes documents into predefined groups, such as language, potential author, or sentiment analysis.
  - While we start this chapter with the same retriever-ranker architecture as Lucene, we expand upon it by incorporating generative AI methods to enhance the user experience in the final section.
- Many of these techniques have emerged within the last decade, with transformer models, in particular, gaining recent widespread attention. As of 2023, OpenAI's ChatGPT is built upon the generative pre-trained transformer 3.5 (GPT-3.5). These models continue to evolve, and competitors are launching improved language models that surpass GPT's capabilities. Notable examples include LLaMA, BARD, Falcon, Cohere, PaLM, Claude, and Titan.
- As these models evolve, they have already introduced novel information retrieval methods. For example, retrieval augmented generation (RAG) integrates classical text search with generative AI to enhance user question answering. Another emerging trend involves training a language model on a specific knowledge base and using it directly to respond to user queries through a chat interface.

- In this chapter, we will use various Python packages. Equivalent versions are available for Java and JavaScript. The JavaScript versions are particularly useful for performing browser-based tasks, leveraging attached GPUs.
  - **tokenizers** is an Apache 2.0 open-source library led by Hugging Face. It offers an implementation of widely-used tokenizers with an emphasis on performance and versatility. It is also utilized in transformers.  
Python: `pip install tokenizers` ([PyPi page](#), [documentation](#), [tutorial](#))
  - **transformers** is an Apache 2.0 open-source library led by Hugging Face. Transformers provides thousands of pretrained models to perform tasks on different modalities such as text, vision, and audio.  
Python: `pip install transformers` ([PyPi page](#), [documentation](#), [tutorial](#))  
JavaScript: `npm install @xenova/transformers` ([npm page](#), [documentation](#), [tutorial](#))
  - **langchain** started in 2022 as an open source project (MIT License) and quickly gained popularity with improvements from hundreds of contributors for the most common AI use cases and with integration with systems from Amazon, Google, and Microsoft.  
Python: `pip install langchain` ([PyPi page](#), [documentation](#), [tutorial](#))  
JavaScript: `npm install langchain` ([npm page](#), [documentation](#), [tutorial](#))
  - **nltk** is a popular library for natural language processing with many integrations for text processing, classification, tokenization, stemming, tagging, parsing, and semantic reasoning.  
Python: `pip install nltk` ([PyPi page](#), [documentation](#), [tutorial](#))
  - **spaCy** is an open source library for natural language processing under MIT license. While NLTK is widely used for teaching and research, spaCy focuses on production ready use cases. It is backed by deep learning models, and supports a number of languages (you need to manually download these models).  
Python: `pip install spacy[cuda113]` ([PyPi page](#), [documentation](#), [tutorial](#))
  - The Apache **OpenNLP** library is a machine learning based toolkit for the processing of natural language text.  
Java: `org.apache.opennlp` ([mvn repository](#), [documentation](#), [tutorial](#))
- Online demos for many functions that we consider in the following sections: <https://textanalysisonline.com/>

## 4.2 Chunking Text

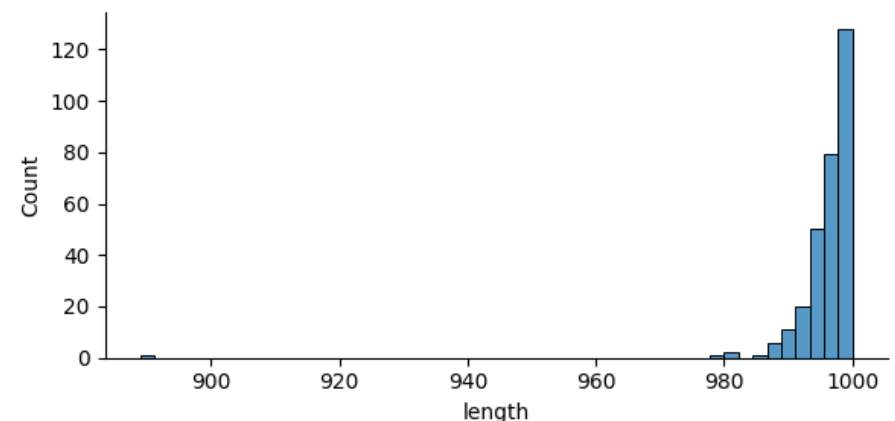
- In the previous chapter, we briefly mentioned splitting larger texts but didn't explore how to perform various splitting strategies and their implications. Let's begin by examining a few use cases and then delve into text splitting implementation strategies:
  - Large documents, like story collections, scientific journals, law books, or novels, often encompass various topics. In **traditional retrieval**, we create a single document vector, merging these diverse aspects. However, this can make it challenging to guide users to specific locations in the document if it appears in search results. By splitting the document into smaller pieces, we generate multiple independent document vectors, enabling precise guidance to relevant parts based on the query.
  - **Text summarization** is more effective when done at the paragraph level, iteratively generating increasingly concise summaries. Large language models often have input token limits, requiring text segmentation before model application.
  - In **Sentiment Analysis**, sentiments in a document can fluctuate. Instead of deriving a single value for the entire document, a more precise analysis continuously assesses sentiments throughout the text, allowing for the calculation of additional document-level statistics.
  - For **large language models**, training foundational models involves smaller text chunks. Large text documents are divided into smaller, coherent parts suitable for the learning task.
  - **Retrieval Augmented Generation**: To produce relevant answers from text searches using generative AI methods, we need to provide sufficient context with the question. However, the token limits of prompts restrict the amount of context that can be provided. To overcome this, we can find small, relevant paragraphs in the text collection and use them to enrich the prompts with context
  - **Machine translation** tasks work on larger chunks of text rather than individual words or entire documents. Splitting the text can improve the semantic correctness of the translation, especially when the translation of a word depends on the broader context. For example, this can help to produce the correct inflected form of a verb or to choose the right word from a list of possible translations.
  - **Natural Language Processing**: Natural language processing (NLP) tasks use larger text chunks to improve semantical interpretation, especially when word meanings depend on context.

- **Method 1: Splitting the text into fixed-sized chunks**

- A simple way to chunk text is to split it into words, and then merge words until the resulting chunks reach a certain size. To remediate the impact of breaking sentences or paragraphs in the middle, we can overlap subsequent chunks by a defined number of words. This way, even if a paragraph is split, it is likely to be retained with the next chunk.
- The `langchain` library has a convenient function for such splits

```
from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(
    separator = " ",
    chunk_size = 1000,
    chunk_overlap = 200,
    add_start_index = True
)
text_splitter.split_text('...a very long text...')
```

- The `CharacterTextSplitter` class creates a splitter using the given `separator` (here a space) that creates chunk up to 1000 characters with a maximum overlap of 200 characters. The `add_start_index` adds information about the start of the chunks when creating sub-documents (see `split_documents`).
- The plot on the right displays the distribution of chunk lengths obtained by applying the above splitter to the sample document ("A Study in Scarlet").
  - Most chunks are nearly 1000 characters long, with variations resulting from word lengths causing splits.
  - The left outlier represents the final chunk of the document, which is generally smaller.
  - Because all chunks have similar lengths, there is no need to be concerned about varying document lengths or input padding for neural networks.
  - A pragmatic approach, but chunks may not align with the changes of topics in the document

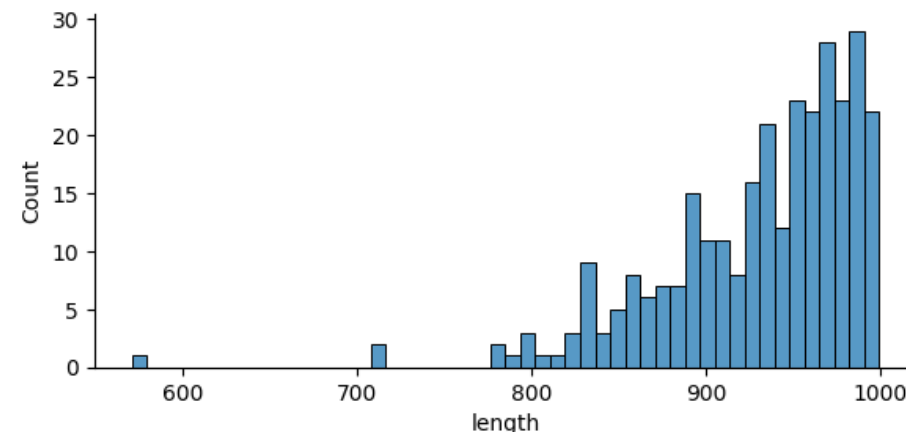


- **Method 2: Splitting at sentence boundaries**

- Similar to the previous approach, we use sentences as the smallest units of text for chunking. This avoids abrupt sentence breaks but may introduce slight variations in chunk sizes, which are usually negligible.
- Sentence segmentation may seem straightforward, but it is slightly more complex and varies by language. For English, a reliable rule set includes:
  - a '?' and '!' terminate the sentence
  - a '.' terminates a sentence unless it is part of a number, initials, or a known abbreviation (e.g., Dr., Mr., U.S.A.); we also require the next character after a sentence boundary to be of upper case
  - punctuations within quotes (spoken words) do not terminate the sentence (e.g., "Hey!", he said)
- The NLTK library includes the Punkt sentence tokenizer, which employs unsupervised algorithms to construct a model for abbreviations, collocations, and sentence-starting words. For example, it identifies (for this very sentence) that the token 'For', a word typically in lowercase, is a likely start of a new sentence. Punkt is trained on a text corpus in the target language to learn probabilities for sentence boundaries. NLTK offers a pre-trained English version of Punkt. There is also a trainer class to learn parameters for other languages. The implementation is speedy and generally very accurate, with occasional difficulties in handling dialogue structures in narrated texts with punctuations inside and outside of quotes.
- The spaCy library employs deep learning models for tasks like information extraction and sentence boundary detection. It offers greater flexibility than Punkt but demands considerably more CPU/GPU resources for sentence tokenization. Thanks to its pre-trained models in multiple languages and its robust architecture, spaCy is well-suited for production use cases.
- The `langchain` library has a convenient functions for sentence based tokenization for both NLTK and spaCy:

```
from langchain.text_splitter import SpacyTextSplitter, NLTKTextSplitter
text_splitter = SpacyTextSplitter(    # or: NLTKTextSplitter
    separator = " ",
    chunk_size = 1000,
    chunk_overlap = 200,
    add_start_index = True
)
text_splitter.split_text('...a very long text...')
```

- The plot on the right displays the distribution of chunk lengths obtained by applying the NLTK splitter to the sample document ("A Study in Scarlet"). The results for spaCy are very similar:
  - o Most chunks are between 900 and 1000 characters long, with variations resulting from sentence lengths causing splits. Long sentences can result in much higher variance than the simple word splitting approach.
  - o The left outlier represents the final chunk of the document, which is generally smaller.
  - o The approach is a bit better than word based splitting, however, we still have a misalignment between chunk size and semantic changes in the document.



### • Method 3: splitting on structure

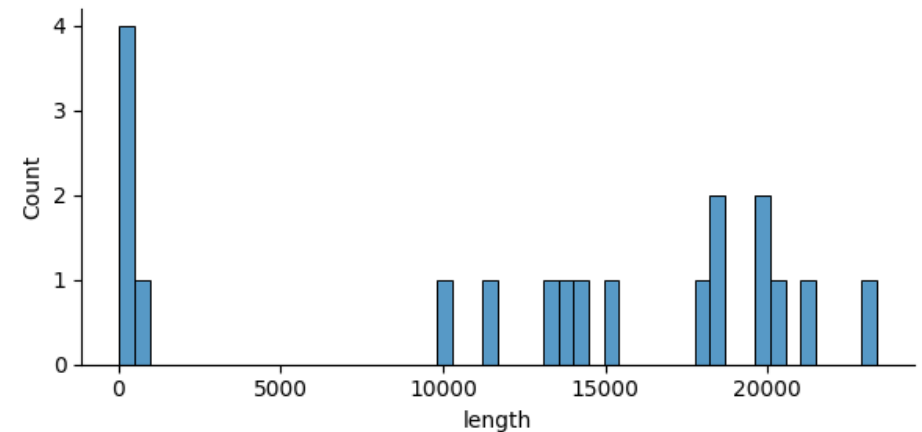
- The concept is to divide text based on its structural elements. For instance, in books, we can split at pages, parts, chapters, sections, and paragraphs. Authors commonly use these structural elements to separate content, making them strong indicators of topic or aspect changes.
- Detecting these structural elements relies on the document's format. In our discussion, we utilized text from Project Gutenberg, where paragraphs, chapters, and parts are structured with increasing numbers of newlines before their start. For instance, 4 consecutive newlines indicate chapters, and 2 consecutive newlines indicate paragraph breaks. In HTML or Markdown formats, we can split by identifying headers (<h1>, <h2>, ...), while treating the text within <p> and <div> tags as paragraphs.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
    separators = ["\n\n"],      # use 4x\n for chapters, and 2x\n for paragraphs
    chunk_size = 100,
    chunk_overlap = 20,
    add_start_index = True
)
text_splitter.split_text('...a very long text...')
```

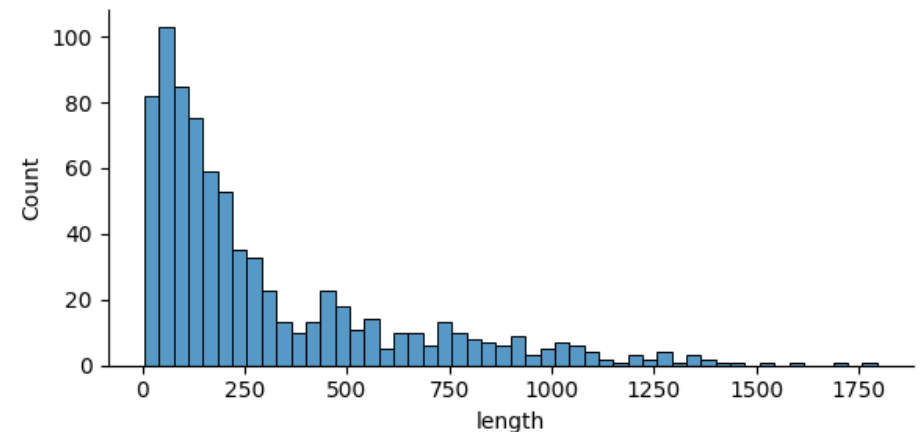


- In this example, we maintain the chunk size and overlap so minimal that **langchain** cannot merge consecutive chunks. Consequently, we obtain a list of distinct parts, chapters, and paragraphs.
- The right-hand figures illustrate chunk size distributions for chapters and paragraphs, revealing a significant contrast with word and sentence-based approaches. Chunks now align with chapter or paragraph boundaries, offering more precise descriptions than the fixed-size splits used previously.
- In contrast to the previous splitting strategies, we now have chunks of varying sizes. In the case of chapters, smaller chunks may result from formatting at the text's start or around parts that can be disregarded. While these coarse-grained splits offer semantically coherent text chunks, their sizes can be challenging to control and may be overly large for certain scenarios.
- When focusing on paragraphs (lower figure on the right), we achieve better control of chunk sizes, typically staying below 2000 characters, although this may vary by the author. However, we end up with a substantial number of very small chunks. This is primarily due to the book's dialog format, where double new lines separate different speakers. These smaller chunks are less suitable for tasks like paragraph indexing as they provide minimal context.

## Chapters



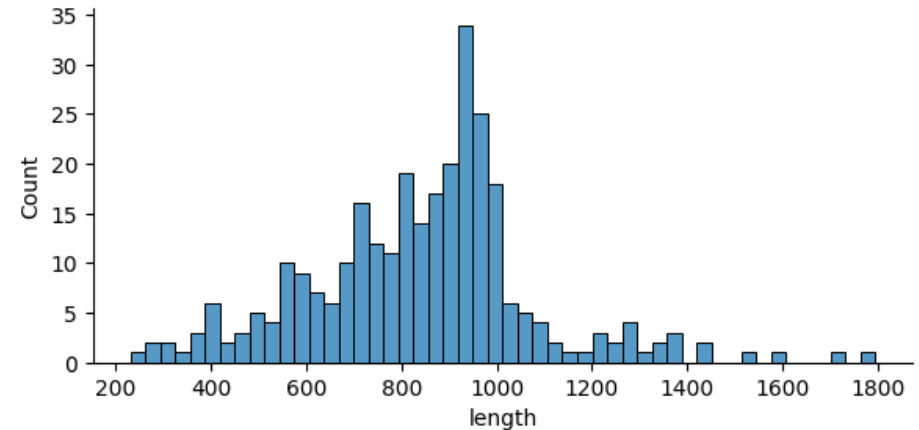
## Paragraphs



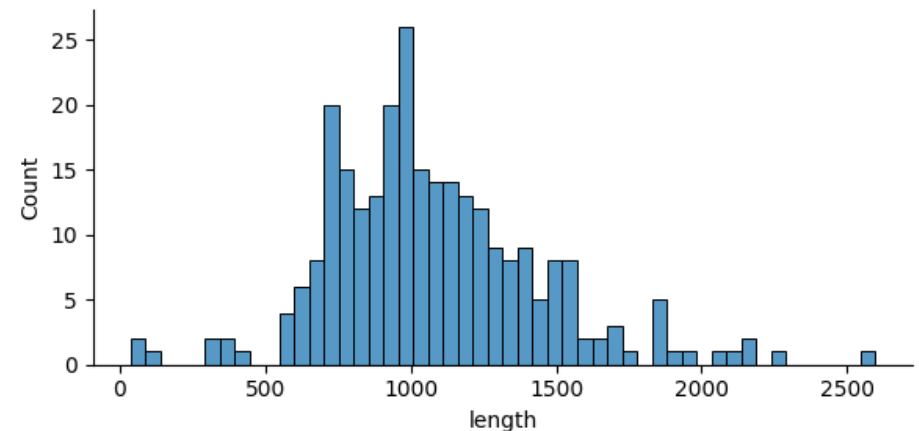


- For enhanced paragraph-based chunking, we can enlarge the chunk sizes and combine smaller paragraphs into larger ones, as they are likely semantically related. The result on the right demonstrates this approach. Small chunk sizes have been eliminated, and most chunks are between 800 and 1000 characters, with some being longer due to the author's preference for lengthy paragraphs. To address excessively long paragraphs, we can introduce additional splits by using the [RecursiveCharacterTextSplitter](#), which accepts a list of separators to further divide oversized chunks hierarchically into smaller ones.
- Controlling the overlap of paragraph-based chunks by specifying a fixed number of characters is challenging, as entire paragraphs can exceed the overlap parameter, resulting in no overlap. A more effective method involves generating paragraph chunks as previously described, merging smaller ones. Then, for each paragraph, append the last sentence from the preceding chunk and the first sentence from the following chunk. These sentences often convey the semantics of neighboring paragraphs, providing improved paragraph descriptions and context.
- If the document includes pagination (e.g., PDFs), a hierarchical chunking method first extracts individual pages and then further divides them into smaller sections if the page size exceeds predefined limits.

### Paragraphs (merged)



### Paragraphs (merged with overlap)



- **Method 4: semantic splitting**

- The structural approach is often the simplest way to obtain semantically coherent chunks. However, when reliable structural context extraction is challenging (e.g., in web pages with varying header formats or scanned documents), we can extend the sentence-based splitting method:

- 1) Define a similarity measure between sentences.
- 2) Set minimum and maximum chunk sizes.
- 3) Split the text into sentences using NLTK or spaCy (merge very short sentences to meet the minimum size).
- 4) Merge neighboring chunks if they are similar (while ensuring they don't exceed the maximum size).

A comprehensive algorithm description is omitted here, but we'll touch on a few key details.

- Similarity between sentences:

- An initial approach uses a vector space model to measure sentence similarities. However, a challenge arises when two sentences lack overlaps in distinctive terms and only share stop words. For example, consider the first two sentences of this paragraph. Despite being part of the same paragraph and discussing the same topic, they only share 'sentence(s)' as a common word among several stop words. If two sentences do not share terms other than stop words, similarity measures such as cosine or dot-product will return 0 values.
- A more effective approach uses embeddings (introduced later in this chapter). Embeddings represent sentences in a lower-dimensional vector space where dimensions correspond to concepts or topics rather than individual terms. Specialized sentence transformers, tailored for embedding generation, yield excellent results. In this chapter, we will also look at alternative techniques, including Latent Semantic Analysis, word2vec, and GloVe. Unlike term vectors, embeddings can map semantically related yet distinct terms to similar positions in the vector space. As a result, similarity measures like cosine or dot product provide more meaningful assessments, even when two sentences lack shared terms.

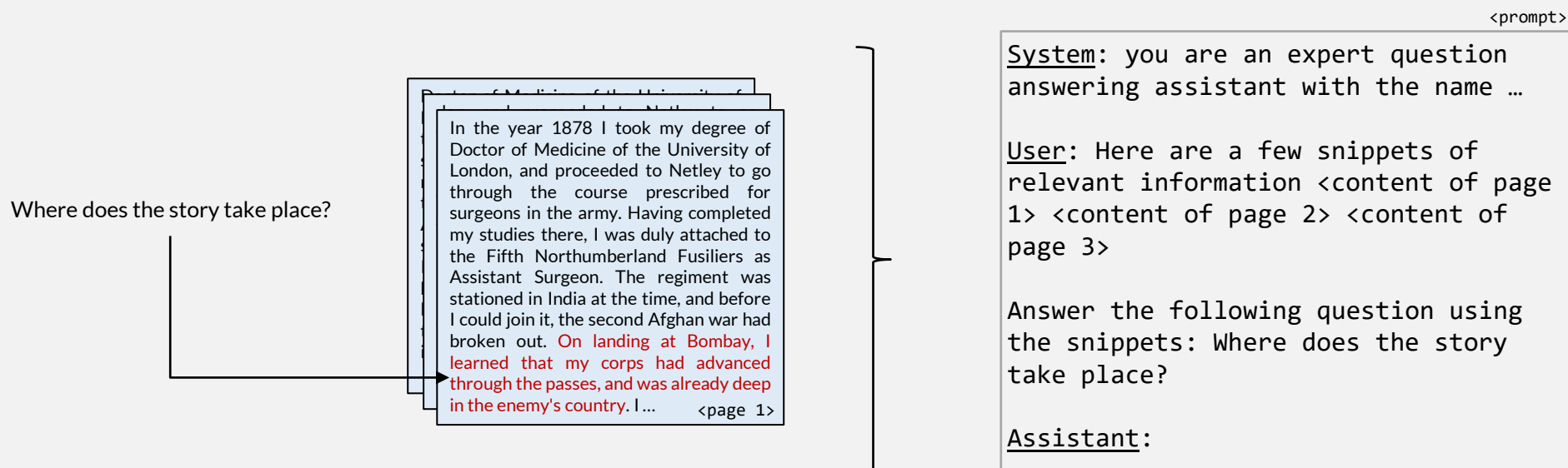
- Merging chunks based on similarities

- A simple approach is to iterate through all chunks, calculate their similarity to the next chunk, and merge them if a specific threshold is surpassed. However, this approach has two main issues:
  1. Determining an appropriate threshold is challenging and requires extensive training before application.
  2. It is difficult to control chunk sizes. Some chunks may remain small because merging is prevented by low similarities, while others may become excessively long when similar but already lengthy chunks are merged.

- Merging chunks based on similarities (continued)
  - A superior approach is to initially compute similarities between all adjacent chunks and then merge the top-k most similar chunks. To avoid excessively long chunks, we prevent the merging of neighboring chunks if their combined length surpasses the predefined maximum size. We can choose a relatively small value for k, such as a percentage of the current number of chunks, and iterate this process multiple times to enhance the clustering of longer sequences of similar sentences in several steps.
- **Discussion:** Segmenting text into smaller parts is specific to the use case domain and, therefore, necessitates optimization tailored to the given scenario, similar to hyperparameter optimization in machine learning. Fortunately, for most use cases, a well-defined overlap between parts reduces the sensitivity to the specific splitting method. Default text splitting may suffice for many scenarios.
  - Chunk size varies based on use case limitations. For instance, in summarization tasks, the maximum chunk size is constrained by the token limit of the chosen language model. In text retrieval scenarios, especially for passage search, an optimal chunk size typically falls between 2000 and 4000 characters, to provide useful results and context for the user.
  - In text retrieval scenarios, breaking a text into numerous smaller parts transforms a document into a collection of sub-documents. For instance, when using Lucene to search novels from Project Gutenberg, this splitting results in the creation of hundreds of sub-documents for each novel. These sub-documents are individually added to the Lucene index and share common metadata for identifying the original document. They also include a position marker in the original text for quick navigation and reference to relevant passages within the full text. This approach expands the number of entries in the Lucene index by a factor of several hundreds, but the overall index size does not increase to the same extent (depending on overlap and duplicated metadata).
  - A useful guideline is to begin with a straightforward splitting method and refine it, if it does not yield the desired results due to suboptimal splitting.

- Method 5: hierarchical chunking

- For RAG use cases, we aim to locate relevant chunks in our library and incorporate them into our prompt alongside the user's query. Embeddings (which we will discuss later) are most effective with smaller chunks, while text generation works better with more context. Earlier models had much smaller context windows, but modern language models have 100k or even 1m token context, allowing for much more information to be included.
- The concept of hierarchical chunking involves using a larger chunk for context and a smaller chunk within that for retrieval. When a smaller chunk is found, it fills in the larger context. We can also add more context by including preceding and succeeding chunks, similar to how humans scan a book for more context.
- To divide into larger parts, we can use the methods mentioned earlier to split the text into overlapping chunks. If the source is a PDF, we can also use page boundaries. For the smaller chunks, it's best to use sentences or a sequence of sentences up to a certain size to match queries against the content with semantic search.
- If we find smaller chunks within bigger chunks, we only include the bigger chunks once in the prompt. Including more nearby (bigger) chunks can give the language model more (hopefully useful) information and create more thorough answers than just one (bigger) chunk. The size and choice of nearby chunks depend on the use case and document types and need to be adjusted during an evaluation phase (hyper parameters).



# 4.3 Tokenization Revisted

- In the previous chapter, we divided the text into parts and employed a straightforward tokenization method. A token is separated by non-word characters. Here is a straightforward Python implementation:

```
def word_tokenize(text: str) -> list[str]:
    text = re.sub(r'^\w\-[ ]+', ' ', text)
    return [token for token in text.split(' ') if token]
```

In this basic scenario, any string of characters that isn't a Unicode letter, number, underscore, or hyphen gets substituted with a single space. Afterward, we break the text at spaces to create a token list. This method usually functions effectively but has certain limitations. Take a look at the sentence below and the resulting split on the right:

I buy my parents' 10% of U.K. startup for \$1.4 billion. Dr. Watson's cat called Mrs. Hersley and it was w.r.o.n.g., more to come ...

While it is an artificial and nonsensical sentence, it highlights some of the weaknesses:

- The possessive "'s" (also: parents' "") is omitted, resulting in a single "s" token. This outcome can be advantageous or disadvantageous, depending on the task. It's beneficial for retrieval since it enables the merging of "Watson" and "Watson's" allowing users to find the name without testing alternative written forms. However, it becomes problematic for sentence analysis as it breaks the link between "Watson" and "cat". The typical retrieval approach involves removing possessive forms and single-letter as well as non-alphabetic terms.
- Numbers function well when they are positive integers, but tokenization struggles with percentages, currencies, and floating-point numbers, among other cases not covered here. In the context of retrieval and NLP, numbers are often disregarded or entirely removed. However, in generative AI, the language model may need to generate an answer with the accurate dollar amount from this sentence.
- Abbreviations like "U.K.," "Dr.," "Mrs.," and the artificial "w.r.o.n.g." are not accurately identified. Abbreviations with multiple dots are treated as separate terms, with all tokens lacking the final dot that signifies an abbreviation. Consequently, searching for "U.K." is not feasible.
- Interpunctuation is absent. This is beneficial for retrieval but restricts sentence analysis for context and word relationships.

word_tokenize
I
buy
my
parents
10
of
U
K
startup
for
1
4
billion
Dr
Watson
s
cat
called
Mrs
Hersley
and
it
was
w
r
o
n
g
more
to
come

- Modern word-based tokenizers are available in the [nltk](#) and [spaCy](#) libraries. The outcome for the same sentence is displayed on the right. They closely match each other, with the exception of the artificial abbreviation "w.r.o.n.g." which [spaCy](#)'s neural model finds challenging. A short comparison to the previous page's basic method:
  - “s” and “” possessives are treated as terms. In retrieval tasks, they can be filtered out, while in NLP tasks, they aid sentence structure analysis.
  - Floating-point numbers are now recognized correctly, including negative numbers (not shown). Percentages and currency symbols are split into individual tokens, preserving this information compared to the previous method.
  - Abbreviations are accurately identified and represented as single tokens. Both [nltk](#) and [spaCy](#) employ machine learning to detect common abbreviations.
  - Interpunctuation is fully retained. In retrieval tasks, it can be filtered out, whereas in NLP tasks, it helps to analyze sentence structure.

Both packages offer support for language-specific peculiarities, such as French abbreviations. Refer to their documentation for details on enabling multi-lingual tokenization.

- **Tokenization for Retrieval:** The tokens displayed on the right are well-suited for NLP and will use them later for part-of-speech tagging. However, for retrieval tasks, many of these tokens are unnecessary as they do not provide additional information. To create a token list for retrieval scenarios, we can undertake the following cleanup actions:
  - Remove short tokens, like single-letter ones, as they lack specific content description.
  - Exclude non-word tokens, such as numbers and special characters, except for words with hyphens and abbreviations with dots. This also removes tokens from possessive forms.
  - Optionally, convert Unicode characters (e.g., accents) to their closest ASCII equivalents, e.g., “Zürich” to “Zurich”. This can reduce vocabulary size (from 16/32 bits to 8 bits) and simplify matching between queries and documents, especially when users lack easy access to specific Unicode letters (e.g., "słychać" with characters not found on the keyboard).
  - Optionally, convert tokens to lowercase or apply case conversion to their standardized form. This is useful for scenarios like sentence beginnings with capitalized words, title case usage, or dealing with misspellings.

nltk-word	spacy-word
I	I
buy	buy
my	my
parents	parents
,	,
10	10
%	%
of	of
U.K.	U.K.
startup	startup
for	for
\$	\$
1.4	1.4
billion	billion
.	.
Dr.	Dr.
Watson	Watson
's	's
cat	cat
called	called
Mrs.	Mrs.
Hersley	Hersley
and	and
it	it
was	was
w.r.o.n.g.	w.r.o.n.g.
,	.
more	,
to	more
come	to
...	come
	...

- There are scenarios where it is not obvious where a word starts and ends:
  - **Scriptio continua** is a writing style without spaces or word separators, often lacking punctuation and sentence boundaries. Prominent examples include Chinese, Japanese, Thai, as well as classical Greek and Latin. Here is an example in Chinese:

莎拉波娃现在居住在美国东南部的佛罗里达。  
 莎拉波娃 现在 居住 在 美国 东南部 的 佛罗里达  
 Sharapova now lives in US southeastern Florida

- A modern variation is found in programming, where literals cannot contain spaces. Depending on coding style, developers employ different methods to create meaningful names, like `QueryParser`, `assertEquals`, `word_tokenize`, `preserve_line`, and more. Coding assistants can break these literals into meaningful tokens to grasp the developer's intent.
- Transcribing spoken language into written form initially creates a phoneme stream and then determines word boundaries. However, in speech, words are not separated; instead, they are joined into a continuous stream of phonemes. For the transcriber who identifies phonemes in the first step, it appears as follows:

ðɪskɔːs'tiːtʃɪz,mʌltɪ'mɪːdiəɹɪ'triːvəl.  
 ðɪs kɔːs 'tiːtʃɪz ,mʌltɪ'mɪːdiə ri'triːvəl.  
 This course teaches multimedia retrieval.

- There are two different approaches to break continuous streams into tokens:
  1. We can combine a dictionary-based approach with a hidden Markov model (or a neural network). The dictionary helps determine if a character sequence can form a word the dictionary and provides all options for the current text position. For instance, consider the English character sequence “h e s i t a t e”. We could extract the single token “hesitate” or the series of tokens “he”, “sit”, “ate”; or “he’s”, “it”, “ate”. Vocabulary lookup can introduce ambiguity which we can resolve using a trained hidden Markov model (or a neural network). This model evaluates alternatives and selects the most likely sequence such as “hesitate” for our example. Language-specific models can utilize rules like maximum matching (finding the longest sequence in the dictionary) and language-specific character usage to identify word boundaries. A general challenges, for instance also in transcription of spoken text, are names of people or brands as well as loan words from other language (e.g., English computer terms in German or Thai).



2. Another approach involves sub-words that are used directly for retrieval. Using the previous example for spoken text retrieval, the phoneme stream is divided into overlapping sequences of three phonemes:

ðɪskɔ:s'ti:tʃɪz,mʌltɪ'mi:diəɪ'tri:vəl. → ðɪs ɪsk skɔ kɔ:s ɔ:s't ... ,mʌl ʌlt ltɪ tɪ'm ɪ'mi 'mi:d i:di diə iər ...

Stress symbols in the phoneme stream are combined with the following phoneme, enlarging the symbol vocabulary. To match this with a query, let's consider the user is searching for “multimedia”. This query is initially translated into a phoneme stream and then segmented into sequences of three phonemes:

multimedia → ,mʌltɪ'mi:diə → ,mʌl ʌlt ltɪ tɪ'm ɪ'mi 'mi:d i:di diə

This creates an 8-token query, and we can employ a standard retrieval method that may consider token proximity. An intriguing outcome of this method is that we do not need to match all sub-sequences to locate relevant spoken text passages. For instance, if a non-native speaker mispronounces words or someone has unclear articulation, the phoneme stream from the spoken text may differ from the one generated by the query. However, as long as there are sufficient overlaps between the sequences, we can still locate the passage.

We can apply this method also in situations where word boundaries are identifiable:

This course teaches multimedia retrieval.

thi his cou our urs rse tea eac ach che hes mul ult ltɪ tim ime med edi dia ret etr tri ...

In this scenario, we create sub-sequences only within words, avoiding sub-word tokens spanning across two words unlike the phoneme example above. An extension of this approach is to differentiate between sub-sequences at the beginning of a word and those within. This distinction can be made by treating them as separate tokens and prefixing sub-sequences at the start of words with “#” (or any unused symbol).

This course teaches multimedia retrieval.

#Thi his #cou our urs rse #tea eac ach che hes #mul ult ltɪ tim ime med edi dia ...

We can transform queries the same way. Let's use an example to illustrate some of the advantages:

Q = “teach multtimedia” → #tea eac ach #mul ult ltt tti tim ime med edi dia

Despite having a different flexed form for “teach” and a misspelling (double “tt”), 10 out of 12 sub-sequences match those from the sentence above. A retrieval model with partial matching and optional token proximity consideration can locate the relevant passage without requiring stemming or spelling corrections. Modern large language models use a similar approach as we will discuss later in this section.

- **N-grams:** Rather than making tokens smaller, we can create larger tokens by combining multiple words into a single token known as n-grams. This approach is particularly valuable in languages where words form phrases with distinct or more specific meanings. Examples include:

- mother tongue, red handed, butterfly effect, black box, cold shoulder, silver bullet, piece of cake
- thai food, prime minister, middle management, crystal clear, chief of staff, speed dial, multimedia retrieval
- New York City, Salt Lake City, Albert Einstein, Amazon Web Services, Ford Mustang, University of Basel

In all these examples, it makes more sense to use phrases rather than the individual terms. In order to enrich a vocabulary with phrases, we can create them manually or form them automatically from a text corpus.

- A naïve approach first constructs all possible bi-grams in a corpus and then counts their occurrences. The top-n most frequent bi-grams are added to the vocabulary. However, a limitation of this method is evident in the upper table on the right side:

bi-gram	$tf(t_1)$	$tf(t_2)$	$tf(t_1, t_2)$
of the	1198	2323	297
in the	669	2323	186
to the	1071	2323	135
to be	1071	246	96
and the	1313	2323	89
upon the	195	2323	88
I have	936	276	81
that I	614	936	76

- "of the" is the most frequent bi-gram simply because it comprises two frequently used stop words in the language.
- A first enhancement excludes stop words when generating bi-grams and considers only consecutive pairs of non-stop words. Ensure that you do not merely remove stop words from the stream but eliminate pairs containing a stop word. Otherwise, you create pairs that originally had a stop word in between. The lower table on the right illustrates the outcomes:
  - The result appears more favorable than previously, with names from the novel forming new terms in the vocabulary. This streamlines the search for names since we only need to search for the bi-gram, eliminating the need to search for individual parts and apply a proximity constraint.
  - Nonetheless, a few issues remain. Phrases like "said Holmes", "could see", and "young man" are common pairs, but they do not contribute significantly to describing the context they appear in. Notably, in the case of "said Holmes", we observe that these two terms less frequently occur together but are more often associated with other words (e.g., "said" is not exclusively used with "Holmes").

bi-gram	$tf(t_1)$	$tf(t_2)$	$tf(t_1, t_2)$
Sherlock Holmes	48	94	48
Jefferson Hope	37	42	34
John Ferrier	31	58	26
Brixton Road	15	13	13
said Holmes	207	94	12
Lucy Ferrier	29	58	10
Enoch Drebber	9	62	9
Salt Lake	9	9	9
could see	96	56	9
young man	40	154	9

- The **Pointwise Mutual Information (PMI)** measures word associations by comparing their actual co-occurrence frequency to what would be expected if they were independent. In our previous example, we noted the bi-gram “said Holmes” occurred 12 times together. However, “said” appeared 207 times, and “Holmes” 94 times individually. In essence, “said” and “Holmes” rarely co-occur (12 out of a maximum of 94 times), and “said” pairs with many other words. Although they occur together more frequently than other bi-grams, this observation suggests they are not a distinctive enough bi-gram for our vocabulary. We are more interested in word pairs like the names which predominantly appear as bi-grams (even though first and last names can also occur independently).
- To formalize this measure, let  $t_1$  represent the first term in the bi-gram and  $t_2$  the second term. We count the occurrences of the individual terms as  $tf(t_1)$  and  $tf(t_2)$ , and of the bi-gram as  $tf(t_1, t_2)$ . PMI compares the likelihood of terms occurring together to the expected probability if they were independent of each other:

$$pmi(t_1, t_2) = \log_2 \frac{p(t_1, t_2)}{p(t_1) \cdot p(t_2)} = \log_2 p(t_1, t_2) - \log_2 p(t_1) - \log_2 p(t_2)$$

- If the corpus comprises  $N$  terms, the probabilities are determined by the ratio of the term frequency to  $N$ :

$$pmi(t_1, t_2) = \log_2 \frac{p(t_1, t_2)}{p(t_1) \cdot p(t_2)} = \log_2 \frac{\frac{tf(t_1, t_2)}{N}}{\frac{tf(t_1)}{N} \cdot \frac{tf(t_2)}{N}} = \log_2 \frac{N \cdot tf(t_1, t_2)}{tf(t_1) \cdot tf(t_2)} \sim \log_2 \frac{tf(t_1, t_2)}{tf(t_1) \cdot tf(t_2)}$$

- In the last part of the formula above, we eliminated the constant multiplier  $\log_2(N)$  that applies to all bi-grams. This adjustment determines the significance of bi-grams in the PMI measure. While it is possible to remove the  $\log_2()$  as well, keeping it in place helps maintain values within more manageable ranges for humans.
- In the context of the last formula, the PMI value is maximized when  $tf(t_1) = tf(t_2) = tf(t_1, t_2)$ , meaning that all occurrences of the two terms exist exclusively within the bi-gram. If a term appears outside the bi-gram, the denominator becomes larger, resulting in a smaller PMI value as a consequence.
- As a result of the previous statement, stop words that appear in bi-grams are naturally given lower weights because they are highly frequent outside of the bi-gram context. Consequently, there is no longer a necessity to employ a stop word filter (although it can still be used for efficiency when computing PMI).

- Now, let's use the PMI measure to find the most significant bi-grams in the same example text. The upper table on the right side displays the outcomes. Notably, all stop words have been excluded, but at the top, we see bi-grams consisting of rare terms. For instance, "Army Medical" appears only once, and the terms within the bi-gram also occur only once within that bi-gram. This is why it receives the highest score.
- We already established that the PMI score is the highest if  $tf(t_1) = tf(t_2) = tf(t_1, t_2)$ . Let's say such a bi-gram occurs  $n$  times. The PMI score is then given by:

$$pmi(t_1, t_2) = \log_2 \frac{N \cdot tf(t_1, t_2)}{tf(t_1) \cdot tf(t_2)} = \log_2 \frac{N \cdot n}{n \cdot n} \\ = \log_2(N) - \log_2(n)$$

- In other words, for bi-grams where the terms only occur together in that bi-gram, the PMI is high when the count  $n$ , denoting the number of bi-gram occurrences, is low. The optimal value is achieved when  $n = 1$ , as demonstrated in the result table ( $\log_2(N)$  is 15.39 for this example).
- To improve the quality of returned bigrams, we can apply a minimum frequency filter as applied for the lower table on the right side. This now eliminates all bi-grams with stop-words and all bi-grams with infrequent terms.
- The bi-gram result is improved, revealing numerous names from the novel and capturing meaningful pairs like "never mind", "old farmer", or "two detectives".

bi-gram	$tf(t_1)$	$tf(t_2)$	$tf(t_1, t_2)$	PMI
Army Medical	1	1	1	15.39
Assistant Surgeon	1	1	1	15.39
Avenging Angels	1	1	1	15.39
Beautiful beautiful	1	1	1	15.39
Boarding Establishment	1	1	1	15.39
CITY Died	1	1	1	15.39
Conan Doyle	1	1	1	15.39
Continental Governments	1	1	1	15.39
Copernican Theory	1	1	1	15.39
Cremona fiddles	1	1	1	15.39
Criterion Bar	1	1	1	15.39
Danite Band	1	1	1	15.39

bi-gram	$tf(t_1)$	$tf(t_2)$	$tf(t_1, t_2)$	PMI
Private Hotel	5	5	5	13.07
Scotland Yard	8	6	6	12.39
Salt Lake	9	9	9	12.22
Baker Street	6	11	6	11.93
Lake City	9	13	8	11.52
Brixton Road	15	13	13	11.48
Never mind	5	37	5	10.18
Jefferson Hope	37	42	34	9.87
Enoch Drebber	9	62	9	9.43
old farmer	38	9	5	9.29
John Ferrier	31	58	26	9.28
Joseph Stangerson	13	43	7	9.07
young hunter	40	14	6	8.84
Sherlock Holmes	48	94	48	8.83
two detectives	75	9	5	8.31
Lucy Ferrier	29	58	10	7.99

- **Likelihood Ratios (LHR)** are another form of hypothesis testing, similar to the chi-squared test but more robust when dealing with sparse data. Moreover, the resulting number is easier to interpret, indicating how much more likely one hypothesis is compared to another. In the context of bi-grams, the initial hypothesis assumes independence between terms  $t_1$  and  $t_2$  in the bi-gram. This hypothesis can be represented as:

$$H_1: P(t_2|t_1) = P(t_2|\neg t_1) = p$$

The first probability represents the conditional likelihood of  $t_2$  following  $t_1$ , while the second one is the conditional probability of  $t_2$  not following  $t_1$ . Let  $tf_1 = tf(t_1)$  represent the occurrences of  $t_1$ ,  $tf_2 = tf(t_2)$  for  $t_2$ , and  $tf_{12} = tf(t_1, t_2)$  for the bi-gram. For hypothesis  $H_1$ , we can use the maximum likelihood estimate for  $p = tf_2/N$ , where  $p$  is the probability of  $t_2$  following any term, whether it is  $t_1$  or not (independence). Assuming a binomial distribution, we can calculate the likelihood of observing these counts as follows:

$$L(H_1) = b(tf_{12}; tf_1, p) \cdot b(tf_2 - tf_{12}; N - tf_1, p) \quad \text{with} \quad b(k; n, x) = \binom{n}{k} x^k (1 - x)^{n-k}$$

The first binomial distribution calculates the likelihood of observing  $tf_{12}$  instances of  $t_2$  following  $t_1$  out of  $tf_1$  occurrences, considering the probability  $p$  that the term  $t_2$  appears at any position. The second hypothesis  $H_2$  assumes that  $t_2$  depends on  $t_1$  and hence the conditional probabilities differ:

$$H_2: p_1 = P(t_2|t_1) \quad p_2 = P(t_2|\neg t_1) \quad p_1 \neq p_2$$

As previously, we can employ maximum likelihood estimates for  $p_1 = tf_{12}/tf_1$  and  $p_2 = (tf_2 - tf_{12})/(N - tf_1)$  using the observed counts. Assuming a binomial distribution, we can then compute the likelihood of the second hypothesis, which is similar to the first, considering both scenarios:  $t_2$  following  $t_1$  and  $t_2$  not following  $t_1$ .

$$L(H_2) = b(tf_{12}; tf_1, p_1) \cdot b(tf_2 - tf_{12}; N - tf_1, p_2) \quad \text{with} \quad b(k; n, x) = \binom{n}{k} x^k (1 - x)^{n-k}$$

Lastly, the likelihood ratio log lambda is given as

$$\log \lambda = \log \frac{L(H_1)}{L(H_2)} = \log \frac{L(tf_{12}; tf_1, p) \cdot L(tf_2 - tf_{12}; N - tf_1, p)}{L(tf_{12}; tf_1, p_1) \cdot L(tf_2 - tf_{12}; N - tf_1, p_2)} \quad \text{with} \quad L(k; n, x) = x^k (1 - x)^{n-k}$$

- Now, let's apply the LHR measure to identify the most significant bi-grams in the same example text. For that purpose, we sort the bi-grams by  $-2 \cdot \log \lambda$ . The upper table on the right shows the results, and interestingly, the stop-words have reappeared. Unlike the naive approach where stop words appeared due to their high frequency, we now have a different scenario as LHR compares the hypothesis of independence versus dependence. Take, for instance, the bi-gram “I am” which is not a significant bi-gram for the vocabulary. Nevertheless, it is evident that “am” is strongly dependent on “I” and follows the term “I” in 39 out of 41 instances.
- We can enhance the quality of bi-grams obtained with LHR by excluding those containing a stop word (refrain from filtering stop words before forming bi-grams). The final outcome is displayed in the lower table on the right. Unlike the PMI ranking, the more frequent names now occupy the top positions. Notably, “Sherlock Holmes” appears 48 times as a bi-gram and attains the highest LHR value, while it held only the 14th place in the PMI ranking, due to the preference of the PMI for lower numbers of occurrences.
- With all the bi-gram scoring methods that we discussed so far, we need to establish a threshold. All bi-grams with scores exceeding this threshold are included in the vocabulary, while the rest are excluded. There is no need to be accurate in setting the threshold, rather, we should take additional search and storage overhead into account. If we missed a bi-gram, we can still find it with proximity measures.
- When creating bi-grams, we can also choose to index both individual terms and the bi-gram. This allows us, for example, to search for “Holmes” which would otherwise not match with occurrences of the bi-gram “Sherlock Holmes”.

bi-gram	$tf(t_1)$	$tf(t_2)$	$tf(t_1, t_2)$	LHR
Sherlock Holmes	48	94	48	618.26
of the	1198	2323	297	517.86
Jefferson Hope	37	42	34	491.94
don t	32	88	32	409.61
to be	1071	246	96	395.47
had been	470	147	63	378.63
in the	669	2323	186	359.01
he said	630	207	72	352.64
John Ferrier	31	58	26	330.18
I have	936	276	81	300.87
I am	936	41	39	284.24
upon the	195	2323	88	259.81

bi-gram	$tf(t_1)$	$tf(t_2)$	$tf(t_1, t_2)$	LHR
Sherlock Holmes	48	94	48	618.26
Jefferson Hope	37	42	34	491.94
John Ferrier	31	58	26	330.18
Brixton Road	15	13	13	224.92
Salt Lake	9	9	9	170.49
Lake City	9	13	8	129.82
Enoch Drebber	9	62	9	119.12
Scotland Yard	8	6	6	109.52
Baker Street	6	11	6	103.36
Private Hotel	5	5	5	100.59
Lucy Ferrier	29	58	10	96.68
Lauriston Gardens	4	4	4	82.26
Joseph Stangerson	13	43	7	79.97
Never mind	5	37	5	71.28
little girl	80	27	8	68.66
young hunter	40	14	6	65.60

- We can expand this concept to tri-grams or even quad-grams. The tables on the right display the top n-grams in the corpus, and we can expand our vocabulary accordingly (typically selecting several hundreds to thousands in a large corpus).
- Finally, the code below demonstrates how to compute the n-gram tables discussed in this section. `nltk` offers a variety of convenient functions for handling collocations. For more details, refer to the documentation.

```
from nltk.collocations import (
    BigramCollocationFinder, TrigramCollocationFinder, QuadgramCollocationFinder,
    BigramAssocMeasures, TrigramAssocMeasures, QuadgramAssocMeasures
)
from nltk.corpus import stopwords

# choose bi-grams, tri-grams, quad-grams
finder = QuadgramCollocationFinder.from_words(tokens)
finder = TrigramCollocationFinder.from_words(tokens)
finder = BigramCollocationFinder.from_words(tokens)

# choose a measure (must match with the finder, here for bi-grams)
measure = BigramAssocMeasures.raw_freq
measure = BigramAssocMeasures.pmi
measure = BigramAssocMeasures.likelihood_ratio

# apply frequency filter
finder.apply_freq_filter(3)

# apply stop word filter
ignored_words = stopwords.words('english')
stopword_filter = lambda w: len(w) < 3 or w.lower() in ignored_words
finder.apply_word_filter(stopword_filter)

# obtain results (top-k)
k = 20
scores = finder.score_ngrams(measure)[:k]

# output term 1, term 2, freq of term 1, freq of term 2, freq of bigram, score
for ((t1,t2),score) in scores:
    print(f'{t1} {t2} {finder.word_fd[t1]} {finder.word_fd[t2]} {finder.ngram_fd[(t1,t2)]} {score}')
```

bi-gram	tf	LHR
Sherlock Holmes	48	618.26
Jefferson Hope	34	491.94
John Ferrier	26	330.18
Brixton Road	13	224.92
Salt Lake	9	170.49
Lake City	8	129.82
Enoch Drebber	9	119.12
Scotland Yard	6	109.52
Baker Street	6	103.36
Private Hotel	5	100.59
Lucy Ferrier	10	96.68
Lauriston Gardens	4	82.26
Joseph Stangerson	7	79.97
Never mind	5	71.28
little girl	8	68.66
young hunter	6	65.60

tri-gram	tf	LHR
said Sherlock Holmes	7	654.14
Sherlock Holmes sprang	3	651.73
asked Sherlock Holmes	3	635.19
said Jefferson Hope	3	504.11
Salt Lake City	8	300.31

quad-gram	tf	LHR
in the Brixton Road	5	644.39
Halliday s Private Hotel	5	255.54
as he spoke and	5	238.97
sprang to his feet	6	221.47

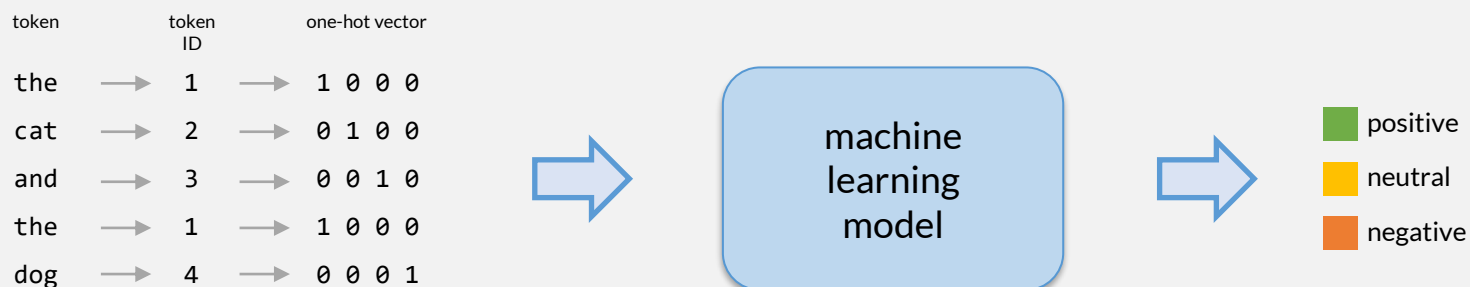


- Tokenization has regained significance alongside the rise of large language models. Later in this chapter, we will explore these models and their applications in retrieval scenarios. When dealing with text in machine learning, a central challenge is how to input text into the model:
  - Many machine learning models typically process continuous input values, with exceptions like decision trees and naive Bayes. In neural networks, a fully connected layer multiplies input values by weights, adds a bias, and applies an activation function. This process results in an outcome represented by a function  $f$  applied to the input values. Therefore, we must find a way to map the generated tokens (assuming words for now) to meaningful input values.
  - An initial idea is to assign a unique ID to each token as we add them to the vocabulary. To illustrate this, consider a simple example: after tokenizing the sentence “the cat and the dog”, we have four tokens:

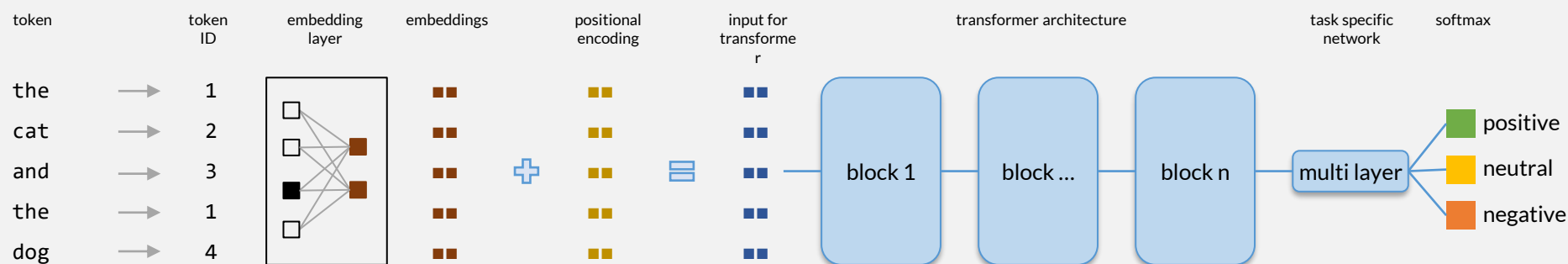
the → 1      cat → 2      and → 3      dog → 4

These IDs enable us to represent the original sentence as a sequence of numbers, such as [1, 2, 3, 1, 4]. However, we cannot directly input these numbers into a machine learning model: the tokens in the example sentence have weak semantic relationships among themselves. Yet, assigning them numerical values like 1, 2, 3, and 4 implies a strong relationship between them. For instance, if “cat” is mapped to 2 and “dog” is mapped to 4, does this imply that 2 cats make up a dog? Additionally, the mapping of “and” to 3, positioned between “cat” and “dog” may suggest that “and” is also an animal because of its proximity to “cat” and “dog”.

- To prevent such misinterpretations, the standard practice in data science and machine learning is to convert discrete values (such as categories or token IDs) into **one-hot vectors**. These vectors have a dimensionality equal to the number of tokens (or categories), and each token is represented by a vector containing all zeros except for one component, determined by the token ID, which is set to 1. The following illustrates this approach in an example of sentiment analysis, where the model predicts whether a sentence is positive, negative, or neutral:



- One-hot vectors work well with a few hundred categories but becomes challenging with millions of entries in vocabularies. Handling massive input layers for machine learning models becomes necessary. Additionally, once a model is trained, expanding the vocabulary is problematic. For instance, adding new terms like names or brands requires retraining the model each time. To address this, models like BERT (Google, 2018) with a 30,000-token vocabulary, GPT-3 (OpenAI, 2020) with 50,000 tokens, and Google's recent Bard model (2023) with 137,000 tokens maintain reasonably sized vocabularies with approaches like Word Piece by Google and Byte-Pair Encoding by OpenAI.
- Before we go deeper into these tokenization methods, note that none of the previously mentioned language models utilize one-hot input vectors due to performance and storage reasons. Instead, they incorporate an extra layer for embedding and positional encoding before the data proceeds to the transformer blocks of the model. While we will explore embeddings later in this chapter, it is opportune to discuss here shortly the complete process from text to transformer layers, as depicted at the bottom of this page:
  - o Embedding layers transform one-hot vectors into lower-dimensional, dense vectors. The concept is to map tokens with semantic similarity closer together in this lower-dimensional space. For example, “cat” and “cats” are semantically related, but one-hot vectors treat them as distinct representations. Embedding layers, however, assign “cat” and “cats” similar vectors, enabling the model to grasp their relationship more effectively.
  - o The embedding layer is a basic fully connected network that maps the one-hot vector into a lower-dimensional representation. This transformation is learned alongside the rest of the model and does not include bias or an activation function. Given that the input is a one-hot vector, the embedding layer essentially looks up the corresponding column in the weight matrix, avoiding the need for expensive matrix-vector multiplications.
  - o The positional encoding is an sinusoidal signal function for the model to understand the order of tokens in the input sequence as transformers lack the inherent sense of order found in recurrent neural networks (RNN).



- **Byte Pair Encoding (BPE) tokenization:** initially proposed for text compression, it was employed by OpenAI to reduce the vocabulary sizes. There are many variants but they all share the same idea:
  - o We begin by normalizing the input sequence. Older models converted Unicode characters to ASCII and to lowercase. More recent models operate at the byte level, treating Unicode characters as sequences of bytes. This enables newer models to better handle languages with special characters. While punctuation is often minimized, it still holds significance in ensuring the model can generate coherent sentences. Let's illustrate BPE tokenization with a simple example and the normalization step:

This course is about this topic. → `this course is about this topic`

- o The initial vocabulary is established with all the characters of all words in the corpus. With our simple example, the initial vocabulary is:

`a, b, c, e, h, i, o, p, r, s, t, u`

- o Next, we expand the vocabulary by including the most common bi-gram found within all words of the corpus. We start by creating a bag-of-words representation and break down each word into character sequences. In our simple example, most words appear only once, but in practice, they would have varying frequencies:

<code>this</code>	<code>2</code>	<code>t, h, i, s</code>
<code>course</code>	<code>1</code>	<code>c, o, u, r, s, e</code>
<code>is</code>	<code>1</code>	<code>i, s</code>
<code>about</code>	<code>1</code>	<code>a, b, o, u, t</code>
<code>topic</code>	<code>1</code>	<code>t, o, p, i, c</code>

- o For each word, we generate all possible bi-grams. For instance, with the word `this` we form the bi-grams `th`, `hi`, and `is`. Subsequently, we count the occurrences of these bi-grams, factoring in the frequency of the words:

`is (3), th (2), hi (2), ou (2), cu (1), ur (1), rs (1), se (1), ab (1), bo (1), ...`

- o `is` is the most frequent bi-gram. So we create a new vocabulary item for it:

`a, b, c, e, h, i, o, p, r, s, t, u, is`

- Using this updated vocabulary, we can now modify the word representations by substituting consecutive **i** and **s** with the new vocabulary item **is**.

this	2	t, h, is
course	1	c, o, u, r, s, e
is	1	is
about	1	a, b, o, u, t
topic	1	t, o, p, i, c

- Now, we can repeat this process by generating again possible bi-grams (where **is** counts as one item, not two):

th (2), his (2), ou (2), cu (1), ur (1), rs (1), se (1), ab (1), bo (1), ut (1), ...

- This time, we encounter a tie, so we can randomly select one of the best pairs. Let's choose **th** and incorporate it into the vocabulary. As previously, we update our word representation:

this	2	th, is
course	1	c, o, u, r, s, e
is	1	is
about	1	a, b, o, u, t
topic	1	t, o, p, i, c

- The process continues until we have arrived at a specified vocabulary size. For the example, we stop at 20:

vocabulary: a b c cou cour cours course e h i is o ou p r s t th this u

- Using this vocabulary, we can now encode our original sentence as follows:

this course is a b ou t this t o p i c

- Even if some words are no longer in the vocabulary, we can still represent them as a sequence of smaller tokens. In a large corpus, this allows the model to handle all words, including misspelled ones, and accept previously unseen words. Newer models use byte-level encoding, avoiding the need to reserve the entire Unicode alphabet in the vocabulary. Instead, they start with 256 initial entries and the BPE algorithm will automatically compose 2 bytes to represent common Unicode characters in the corpus.

- The transformers library offers efficient tools for training custom tokenizers. The code on the right demonstrates how to utilize the library to train custom BPE tokenizers (steps 1-4) and how to reuse an existing BPE tokenizer (step 5 with GPT-2).
  1. Language model training necessitates special tokens that convey unique conditions to the model. The "unknown" token, for example, is employed for any input sequence that cannot be matched to a token in the vocabulary.
  2. The trainer drives the BPE algorithm, allowing us to configure the vocabulary size, specify a minimum frequency threshold for pairs to enter the vocabulary, and to define unique prefixes and suffixes for pairs occurring within or at the end of words. This ensures that the same pair, like "is", is treated as distinct tokens when it appears at the start, middle, or end of a word.
  3. Specifies how input text is segmented into words, handles punctuation, and applies normalization before tokenization, such as converting text to lowercase. Note that normalization is useful for tasks like classification but should be avoided when working with language models that generate text. Lowercasing, for example, would prevent the model to produce proper English sentences like this one (capital cases).
  4. Trains the BPE tokenizer using a set of input files, and then apply it to encode some text.
  5. Create a GPT-2 pre-trained tokenizer and use it to encode some text.

```
from tokenizers import (
    models, Tokenizer,
    normalizers, pre_tokenizers,
    trainers,
)

# 1) define special tokens and create tokenizer object
unknown_token = "[UNK]"
special_tokens = [unknown_token, "[SEP]", "[MASK]", "[CLS]"]
tokenizer = Tokenizer(models.BPE(unk_token = unknown_token))

# 2) setup the trainer for BPE tokenization
trainer = trainers.BpeTrainer(vocab_size=10000,
    min_frequency=3,
    special_tokens = special_tokens,
    continuing_subword_prefix='#',
    end_of_word_suffix='>')

# 3) define how to split the text and normalize words
tokenizer.pre_tokenizer = pre_tokenizers.WhitespaceSplit()
tokenizer.normalizer = normalizers.Sequence(
    [normalizers.NFD(), normalizers.Lowercase(),
     normalizers.StripAccents()])

# 4) train the tokens on a set of files
tokenizer.train(files, trainer=trainer)
tokenizer.encode(text)

# 5) using a pre-built BPE tokenizer
from transformers import GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer.encode(text)
```

– **WordPiece Tokenization** follows BPE's general approach but extends it in two key ways:

1. It distinguishes between characters at the word's beginning and those in the middle. The original BPE version, initially stemming from a compression algorithm, did not account for character positions. However, later extensions, including the transforms library demonstrated on the previous page, incorporated this concept. In our example sentence, "this course is about this topic", the starting vocabulary is altered: '##' serves as a special annotation for word pieces that start within the word:

```
vocabulary:  ##b, ##c, ##e, ##h, ##i, ##o, ##p, ##r, ##s, ##t, ##u, a, c, i, t
this        2    t, ##h, ##i, ##s
course      1    c, ##o, ##u, ##r, ##s, ##e
is          1    i, ##s
about       1    a, ##b, ##o, ##u, ##t
topic       1    t, ##o, ##p, ##i, ##c
```

While this approach doubles the base vocabulary, it enhances our ability to capture prefixes, which frequently convey shared semantics across words. The same principle applies to suffixes, which often group inflected forms based on gender, numbers, tense, and case.

2. It constructs pairs in the same manner as BPE but prioritizes pairs with their components occurring more frequently together than with other pieces. This results in the same scoring criteria that we introduced for selecting bi-grams with PMI. If  $(a, b)$  represents a potential candidate pair, we denote the frequencies of the individual elements of the pair as  $tf(a)$  and  $tf(b)$ , and the frequency of the pair itself as  $tf(a, b)$ . The best pair is determined as follows:

$$(a^*, b^*) = \operatorname{argmax}_{(a,b) \in \text{pairs}} \frac{tf(a, b)}{tf(a) \cdot tf(b)}$$

As discussed with PMI, we need to apply a minimum frequency filter as the formula above prefers infrequent pairs such as the ones with  $tf(a, b) = tf(a) = tf(b) = 1$ .

- The BPW and WordPiece algorithms share the same process: they begin with an initial vocabulary, create pairs, count frequencies, expand the vocabulary with the best pair, merge pairs for all words, and continue this cycle of pair creation, vocabulary expansion, and merging until a specified vocabulary size is achieved.

- We can employ efficient WordPiece trainers from the transformers library. The code on the right demonstrates how to utilize the library to train custom WordPiece tokenizers (steps 1-4) and how to reuse an existing WordPiece tokenizer (step 5 with BERT uncased).
1. As before, we define the special tokens depending on the task we want to address with the model. The "unknown" token, for example, is employed for any input sequence that cannot be matched to a token in the vocabulary.
  2. The trainer drives the WordPiece algorithm, allowing us to configure the vocabulary size, specify a minimum frequency threshold for pairs to enter the vocabulary, and to define unique prefixes and suffixes for pairs occurring within or at the end of words.
  3. Specifies how input text is segmented into words, handles punctuation, and applies normalization before tokenization, such as converting text to lowercase. Again, select normalization that is applicable for the scenario and model usage. Lowercase normalization, for instance, prevents the model from producing grammatically correct output.
  4. Trains the WordPiece tokenizer using a set of input files, and then apply it to encode some text.
  5. Create a BERT (uncased) pre-trained tokenizer and use it to encode some text.

```

from tokenizers import (
    models, Tokenizer,
    normalizers, pre_tokenizers,
    trainers,
)

# 1) define special tokens and create tokenizer object
unknown_token = "[UNK]"
special_tokens = [unknown_token, "[SEP]", "[MASK]", "[CLS]"]
model = models.WordPiece(unk_token = unknown_token)
tokenizer = Tokenizer(model)

# 2) setup the trainer for WordPiece tokenization
trainer = trainers.WordPieceTrainer(vocab_size=10000, \
    min_frequency=3, \
    special_tokens = special_tokens, \
    continuing_subword_prefix='_', \
    end_of_word_suffix='|')

# 3) define how to split the text and normalize words
tokenizer.pre_tokenizer = pre_tokenizers.BertPreTokenizer()
tokenizer.normalizer = normalizers.Sequence(
    [normalizers.NFD(), normalizers.Lowercase(),
     normalizers.StripAccents()]
)

# 4) train the tokens on a set of files
tokenizer.train(files, trainer=trainer)
tokenizer.encode(text)

# 5) using a pre-built WordPiece tokenizer
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
tokenizer.encode(text)

```



## 4.4 Lemmatization and Linguistic Transformation

- Stemming reduces words in the input to their root, ensuring variants match during search. For example, 'houses' in a document can match a query with 'house'. Stemming is language-dependent, but typically, removing prefixes and suffixes is effective for most languages. Words that can undergo significant inflections, like 'go' and 'went,' present more challenges. We distinguish different types of stemming algorithms:
  - **Rule-based stemmers** use rules to transform words to their stems, which may not always be linguistically correct but are designed to match variants of the same root. In text retrieval, displaying these stems to users is not necessary; they are only used for quick lookup with inverted files.
  - **Dictionary-based stemmers** use a small set of rules for regular inflections and rely on a dictionary and irregular inflection list to find the correct linguistic stem. In text retrieval, this improves the success for matching word variants, especially in cases with strong inflections like 'go' and 'went'.
- In the last chapter, we learned about the **Porter** algorithm, a basic English stemmer that creates pseudo-stems to unify word variations. The **Lancaster** stemmer is another rule-based stemmer for English. It aggressively cuts off word endings (suffixes), which can lead to very short stems. It is faster than other algorithms and suitable for general English text processing.
- In various retrieval situations, handling diverse languages is common. Applying Porter or Lancaster stemmers to non-English text does not work. As a solution, Martin Porter introduced the **Snowball** framework to create rule-based stemmers for multiple languages similar to Porter and Lancaster. This framework features its own rule definition language, and can generate code for different programming languages. The result is still a pseudo-stem, and in languages with strong inflections, the stem can vary due to gender, tense, or case changes. These algorithms are highly efficient and can operate in any environment without requiring a large dictionary.
- An improvement over the rule based stemmers are dictionary based stemmers such as provided by WordNet and spaCy. They consist of three parts:
  - a simple rule-based stemmer for regular inflections (e.g., '-ing', '-ed')
  - an exception list for irregular inflections
  - a dictionary of all possible stems of the language

- The dictionary based algorithms works as follows:
  1. Retrieve a part-of-speech (POS) tag for the current word. This is typically done during tokenization and considers the broader context to determine the correct tag (e.g., noun, verb, adjective, punctuation). For example, whether 'run' is a noun or a verb depends on its context.
  2. Search for the word in the dictionary; if it is found, the the word is not inflected, we can return it as its own stem.
  3. Search for the word in the exception list for its POS tag (see tables below for examples); if it is found, we can return the stem as given in the list.
  4. Apply rules based on the POS tag to shorten regularly inflected forms using their suffixes. The table on the right shows some English examples. Use each applicable rule and check the dictionary; if the word is found, return the form from the dictionary.
  5. If no dictionary entry is found, return the word as its own stem. This can occur with names, misspelled words, or loanwords (words from another language, like English words in German).

Type	Suffix	Ending
NOUN	s	
NOUN	ses	s
NOUN	xes	x
NOUN	zes	z
NOUN	ches	ch
NOUN	shes	sh
NOUN	men	man
NOUN	ies	y
VERB	s	
VERB	ies	y
VERB	es	e
VERB	es	
VERB	ed	e
VERB	ed	
VERB	ing	e
VERB	ing	
ADJ	er	
ADJ	est	
ADJ	er	e
ADJ	est	e

#### adj.exc (1500):

...  
 stagi<sup>est</sup>      stagi  
 stalk<sup>ier</sup>      stalky  
 stalk<sup>iest</sup>      stalky  
 stap<sup>ler</sup>      stapler  
 starch<sup>ier</sup>      starchy  
 starch<sup>iest</sup>      starchy  
 star<sup>er</sup>      starer  
 star<sup>est</sup>      starer  
 star<sup>rier</sup>      starry  
 star<sup>riest</sup>      starry  
 statel<sup>ier</sup>      stately  
 statel<sup>iest</sup>      stately  
 ...

#### verb.exc (2400):

...  
 ate      eat  
 atroph<sup>ied</sup>      atrophy  
 aver<sup>ed</sup>      aver  
 aver<sup>ring</sup>      aver  
 awak<sup>ed</sup>      awake  
 awak<sup>en</sup>      awake  
 babi<sup>ed</sup>      baby  
 baby-sat      baby-sit  
 baby-sit<sup>ting</sup>      baby-sit  
 back-pedall<sup>ed</sup>      back-pedal  
 back-pedall<sup>ing</sup>      back-pedal  
 backbit      backbite  
 ...

#### noun.exc (2000):

...  
 neuromata      neuroma  
 neuroptera      neuropter<sup>on</sup>  
 neuroses      neurosis  
 nevi      nevus  
 nibelungen      nibelung  
 nidi      nidus  
 nielli      niello  
 nilgai      nilgai  
 nimbi      nimbus  
 nimbostrati      nimbostratus  
 noctilucae      noctiluca  
 ...

- **nlTK** and **spaCy** provide multiple stemmers for text processing in different languages. The code on the right demonstrates how to begin using these stemmers.
  - **English Example:** The table below displays stemming outcomes for Porter, Lancaster, Snowball, WordNet, and **spaCy** when applied to English text. The table excludes terms that yield the same stem across all stemmers. It also include the part-of-speech tag (pos) that was used for WordNet and **spaCy**.

term	pos	porter	lancaster	snowball	wordnet	spaCy
blue	ADJ	blue	blu	blue	blue	blue
bottles	NOUN	bottl	bottl	bottl	bottle	bottle
bristled	VERB	bristl	bristl	bristl	bristle	bristle
companion	NOUN	companion	comp	companion	companion	companion
cry	NOUN	cri	cry	cri	cry	cry
discovered	VERB	discov	discov	discov	discover	discover
distant	ADJ	distant	dist	distant	distant	distant
feet	NOUN	feet	feet	feet	foot	foot
flickering	NOUN	flicker	flick	flicker	flickering	flicker
found	ADP	found	found	found	found	find
had	VERB	had	had	had	have	have
have	VERB	have	hav	have	have	have
his	PRON	hi	his	his	his	his
is	VERB	is	is	is	be	be
lofty	ADJ	lofti	lofty	lofti	lofty	lofty
nothing	NOUN	noth	noth	noth	nothing	nothing
one	NUM	one	on	one	one	one
only	ADV	onli	on	onli	only	only
over	ADP	over	ov	over	over	over

```
import nltk
from nltk.corpus import wordnet
import spacy

# building a stemmer
porter = nltk.PorterStemmer()
lancaster = nltk.LancasterStemmer()
snowball = nltk.SnowballStemmer("english")
wordnet = nltk.WordNetLemmatizer()
spacy = spacy.load('en_core_web_sm')

# applying it
porter.stem('discovered')
lancaster.stem('discovered')
snowball.stem('discovered')
wordnet.lemmatize('discovered', 'v')

# spacy processes full text sequence,
# not just one word
for token in spacy('I have discovered it'):
    print(token.text, token.lemma_)
```

- **English Example (cont'd):** The Snowball and Porter algorithm yield very similar results as they mostly rely on the same rules, with Snowball being a slightly revised version of Porter. In contrast, Lancaster is more aggressive in removing suffixes, often resulting in overly short stems that may collide with unrelated words, especially when they are short themselves (e.g., 'one' and 'only' both reduced to 'on'). WordNet and [spaCy](#) produce similar results, but their stems differ from those of the other algorithms. Notably, all WordNet and [spaCy](#) stems are linguistically correct ('bottle' vs. 'bottle'). In text retrieval, stem correctness matters less than ensuring variants map to the same stem and thus the same token ID in the index. This is evident in examples like 'had' and 'have' which the rule-based algorithms map to different stems, while the dictionary-based algorithms map them to the same base form 'have'. This enhances the search engine's ability to match query variants with those found in documents.
- **German Example:** Snowball and [spaCy](#) are the only options for German stemming, allowing us to compare Snowball's rule-based approach with [spaCy](#)'s dictionary-based approach. The results are displayed on the right. We observe similar differences between rule-based (Snowball) and dictionary-based ([spaCy](#)) stemming. Additionally, Snowball maps special characters to a base character set and converts text to lowercase. It also handles cases like 'ae' → 'a' if the text doesn't use 'ä' correctly. [spaCy](#) corrects casing only if a word starts a sentence and would normally be in lowercase. Snowball's results are acceptable for text retrieval, but [spaCy](#) performs significantly better in identifying the true linguistic stem and matching strongly inflected variants (consider 'beschloß' and 'beschließen').

term	pos	snowball	spaCy
abgefunden	VERB	abgefund	abfinden
abraten	VERB	abrat	abraten
alleiniger	ADJ	allein	alleinig
alles	PRON	all	alle
am	ADP	am	an
Andererseits	ADV	andererseit	andererseits
Anteil	NOUN	anteil	Anteil
aufgerichtet	VERB	aufgerichtet	aufrichten
Augen	NOUN	aug	Auge
bedeutet	VERB	bedeutet	bedeuten
beschloß	VERB	beschloss	beschließen
blickte	VERB	blickt	blicken
damals	ADV	damal	damals
das	PRON	das	der
dem	PRON	dem	der
den	DET	den	der
des	DET	des	der
Die	DET	die	der
Einerseits	ADV	einerseit	einerseits
endgültig	ADV	endgult	endgültig
energischen	ADJ	energ	energisch
Erbe	NOUN	erb	Erbe
Firma	NOUN	firma	Firma
Flamme	NOUN	flamm	Flamme

- **French Example:** For French stemming, Snowball and [spaCy](#) are the sole options, and the table on the right compares their performance with a French text. Unlike in German, French Snowball retains accented characters but still converts words to lowercase, while [spaCy](#) preserves casing for names. We observe similar differences between the rule-based (Snowball) and dictionary-based ([spaCy](#)) approaches as seen in the German example. In this French example, the ability to map various inflected forms to the same stem is even more noticeable, as Snowball often assigns different stems to different inflected forms of the same root (e.g., ‘aperçu’ and ‘aperçut’, ‘avait’ and ‘avaient’).

term	pos	snowball	spaCy
Aiguillon	PROPN	aiguillon	Aiguillon
Aramis	X	aram	Aramis
Artagnan	PROPN	artagnan	Artagnan
Tous	ADJ	tous	tout
Tréville	PROPN	trévill	Tréville
agréable	ADJ	agréabl	agréable
aperçu	ADJ	aperçu	apercevoir
aperçut	VERB	aperçut	apercevoir
approcha	PROPN	approch	approcher
arrivé	VERB	arriv	arriver
avaient	AUX	avaient	avoir
avait	AUX	avait	avoir
causant	VERB	caus	causer
cet	DET	cet	ce
comme	CONJ	comm	comme
conciliation	NOUN	concili	conciliation
contraire	NOUN	contrair	contraire
conversation	NOUN	convers	conversation
courtoisie	NOUN	courtois	courtoisie
côté	NOUN	côt	côté
devant	ADP	dev	devant
du	ADP	du	de
emporté	ADJ	emport	emporté
entier	ADJ	enti	entier

- In linguistics, compounds are words created by combining two or more base words, occasionally using binding syllables (e.g., 'Liebeslied') or characters (e.g., 'must-have'). While most languages support basic compound formation to create new words (e.g., 'smalltalk'), languages such as German and Finnish permit the formation of arbitrary long compounds. Let's examine a few examples:

- Finnish:

- |   |   |
|---|---|
| – <i>kolmivaihekilowattituntimittari</i>                                    | <b>en:</b> electricity meter  |
| – <i>atomiydinenergiareaktorigeneraattorilauhduttajaturbiiniratasvaihde</i> | <b>en:</b> atomic nuclear energy reactor generator condenser turbine cogwheel stage |
| – <i>rautatieasema</i>  | <b>en:</b> railway station  |

- German:

- |   |   |
|---|---|
| – <i>Wolkenkratzer</i>  | <b>en:</b> skyscraper   |
| – <i>Rinderkennzeichnungs- und Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz</i> (German (law in Mecklenburg-Vorpommern, 1999-2013) | <b>en:</b> cattle marking and beef labeling supervision duties delegation law |
| – <i>Stacheldraht</i>   | <b>en:</b> barbed wire  |

- Dutch

- |  |  |
|--|--|
| – <i>arbeidsongeschiktheidsverzekering</i> | <b>en:</b> disability insurance          |
| – <i>rioolwaterzuiveringsinstallatie</i>   | <b>en:</b> sewage treatment plant        |
| – <i>doorgroeimogelijkheden</i>            | <b>en:</b> possibilities for advancement |

- We can classify compounds as either endocentric or exocentric.

- **Endocentric compounds** derive their meaning from their constituent parts. They have a 'head' that imparts both semantic and syntactic attributes to the compound, while the other elements modify and refine its meaning. For instance, in 'sunglasses', 'glasses' serves as the head, and 'sun' acts as the modifier.
- **Exocentric compounds** do not derive their meaning from their constituent parts and may even ignore the lexical class of their individual elements (e.g., 'must-have' is a noun, not a verb). In the word 'skyscraper', neither 'sky' nor 'scraper' acts as the head, and the term names an entirely different object (in this case, a type of building).

- Consider a compound word like ‘Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz’: the first challenge is spelling it correctly, making it difficult to locate the term in document titles if users do spelling mistakes in the query (perhaps why the law was abandoned). Another issue is that we must list all constituent parts to find this document. However, it would be more user-friendly to search with ‘Rindfleisch Etikettierung Gesetz’ as a partial query match. Unfortunately, the retrieval models we have discussed so far do not support partial term queries against the vocabulary or the use of such matches for actual searches.
- The recommended approach for handling compounds is to break them into their constituent parts and include both the parts and the compound as tokens in the document. For instance, the German word ‘Abfalleimer’ is treated as three tokens: ‘Abfall’, ‘Eimer’, and ‘Abfalleimer’. This allows users to match this word with a broader range of query possibilities. While this method works well for endocentric compounds like ‘Abfalleimer’ where the parts convey a similar meaning to the compound, it is less effective for exocentric compounds. Take ‘skyscraper’ as an example (similarly in German: ‘Wolkenkratzer’). If we split the compound into ‘sky’ and ‘scraper’ (German: ‘Wolke’, ‘Kratzer’), we introduce incorrect semantics into the document descriptors. The effectiveness of balancing the additional value gained from splitting endocentric compounds with the introduction of incorrect semantics for exocentric compounds depends on the specific retrieval scenario.
- We discuss two approaches to split compounds. Both approaches use a rule-based or morphological analysis to identify potential splits of a term. The specific details vary from language to language. Let’s look at examples:
  - In English, we can split compounds using hyphens and syllables in accordance with English hyphenation rules. For example: ‘must-have’ becomes ‘must’ and ‘have’; and ‘skyscraper’ becomes ‘sky’, ‘scrap’, and ‘er’.
  - In German, we split on syllables following German hyphenation rules. For example, ‘Wolkenkratzer’ becomes ‘wol’, ‘ken’, ‘krat’, ‘zer’; and ‘Schiffahrtskapitän’ becomes ‘Schiff’, ‘fahrt’, ‘fahrtss’, ‘ka’, ‘pi’, ‘tän’. Note that in that last example ‘s’ is a binding letter for compound generation and we have to test with both pieces ‘fahrt’ and ‘fahrts’.

As a next step, we produce all possible combinations of such splits:

skyscraper	→ (sky, scrap, er), (skyscrap, er), (sky, scraper)
wolkenkratzer	→ (wol, ken, krat, zer), (wolken, krat, zer), (wol, kenkrat, zer), (wol, ken, kratzer), (wolken, kratzer), (wolkenkrat, zer), (wol, kenkratzer)
Schiffahrt <u>s</u> kapitän	→ (Schiff, fahrt, ka, pi, tän), (Schiff, fahrts, ka, pi, tän), (Schiffahrt, ka, pi, tän), (Schiffharts, ka, pi, tän), (Schiff, fahrtska, pi, tän), (Schiff, fahrtska, pi, tän), ... (Schiffhart, kapitän), (Schiffharts, kapitän)



- To find valid splits, we start by discarding any splits that contain components not found in our vocabulary or dictionary. When multiple options remain, we determine the best split based on the frequency of the components. Let  $\mathbb{S}$  represent the set of all possible splits, and let  $S = \{p_i\}$  represent all the individual components of split option  $S \in \mathbb{S}$ . We calculate  $tf(p_i)$  as the number of times piece  $p_i$  appears in the corpus (or is provided by the dictionary), and  $N$  represents the total number of tokens in the corpus (or as given by the dictionary):

$$S^* = \operatorname{argmax}_{S \in \mathbb{S}} \left( \prod_{p_i \in S} \frac{tf(p_i)}{N} \right)^{\frac{1}{|S|}} = \operatorname{argmax}_{S \in \mathbb{S}} \frac{1}{|S|} \cdot \sum_{p_i \in S} \log \left( \frac{tf(p_i)}{N} \right) = \operatorname{argmax}_{S \in \mathbb{S}} \frac{1}{|S|} \cdot \sum_{p_i \in S} \log(tf(p_i))$$

In simpler terms, we choose the split with the highest average log-frequency values for its components. This indicates the most probable way to combine the parts into a compound.

- When analyzing text, we come across **homonyms** and **synonyms**. A homonym is a word that looks the same as another word but has a different meaning and sometimes different pronunciation (e.g., ‘lead’ for guiding or as a metal). In contrast, a synonym has a similar or nearly identical meaning but is expressed with a different word, often to prevent repetition in writing (e.g., ‘big’ and ‘large’). Let's explore how they affect text retrieval and methods to address them effectively:
  - **Synonyms** are commonly used to add variety to written text. However, this can affect the retrieval engine's ability to match query terms with those in the document. For instance, if the document contains ‘purchase’, a query with ‘buy’ or ‘acquire’ can not match it due to the different token forms. There are two main alternatives to address this. First, synonym expansion involves tokenizing the document (and/or the query) and expanding tokens using predefined synonym lists. Second, as discussed later in this chapter, word embeddings can be used to map terms into a high-dimensional, sparse space, considering relationships between words.
  - Handling **homonyms** involves analyzing the context to clarify the intended meaning. In straightforward cases, part-of-speech tags can distinguish between verb and noun forms (e.g., ‘lead’ as a guide or as a metal). More advanced solutions use machine learning models to determine the context accurately or analyze grammatical structures for context. When a query contains a homonym, we can either select the most common meaning or present the user with individual results for each potential interpretations. For example, the word ‘bank’ has two meanings (sloping land by water, financial institution). We can seek user feedback for the correct interpretation or offer two result options with synonym expansion for both possible meanings.

- Another common word relationship is between **hypernyms** and **hyponyms**. A hypernym has a broader, more general meaning and is often seen as the higher-level category among words. In contrast, a hyponym has a narrower, more specific meaning and is typically viewed as the lower-level category. For instance, ‘animal’ is a hypernym (a more general class) related to the hyponyms ‘cat’ and ‘dog’ which represent more specific types of animals. Words can be hypernyms and hyponyms at the same time. A mammal is hypernym for cat, but a hyponym for animal.
  - **Faceted search** enables users to explore search results by expanding or narrowing categories using hypernym/hyponym relationships. For example, in an image search for ‘animals’, users can drill down to more specific types like ‘cats’ and ‘dogs’. Conversely, if a query is too specific and yields few results, users can quickly broaden the search using presented hypernym hierarchies.
  - Automatically **expand queries** with hypernyms and hyponyms to broaden the search. We can assign weights to the original term, its hypernyms (with less weight), and its hyponyms (with the same or less weight) to incorporate term relationships into the search process.
  - **Relevance ranking** considers hypernym/hyponym relationships to evaluate document relevance, even when the query term is absent. This is similar to query expansion, but the distinction lies in where the expansion occurs. In query expansion, we submit a longer query with weighted hypernyms and hyponyms. In relevance ranking, we retain the user's original query and adjust scoring functions to account for hypernyms and hyponyms.
- The WordNet website offers an online demo for in-depth exploration of synonyms, homonyms, hypernyms, and hyponyms in English. You can access WordNet data through [nltk.corpus.wordnet.synsets\(word\)](http://nltk.corpus.wordnet.synsets(word)), which returns synsets providing functions to access synonyms, homonyms, hypernyms, meronyms, and various other relationships. Visit <http://wordnetweb.princeton.edu/perl/webwn> for the WordNet online demo.
- The last discussion in this section on tokenization considers **spelling mistakes** and how to treat them. Typically, we employ a spellchecker to replace words not found in the dictionary. During document indexing, we retain the original misspelled version, and add the auto-corrected version(s) to the index. In queries, we can expand the query with auto-corrected version(s) or suggest alternative queries if the misspelled query yields insufficient results (“did you mean?”). Spelling mistakes, especially in names, are common but can be challenging to differentiate from intentional variations. For instance, the name “Britney” has various alternative forms such as “Britni”, “Brittney”, “Britnee”, “Britneigh”, “Britnie” and many more. If we would only use auto-corrected versions, we may not find these alternative forms.

## 4.5 Part of Speech

- Sentences consist of words belonging to various grammatical classes, known as part-of-speech (POS) categories, which share similar grammatical characteristics. In English, common parts of speech include noun, verb, adjective, adverb, pronoun, conjunction, interjection, numeral, article, and determiner. POS tagging is a method for assigning the appropriate class to a word based on its position and function within a sentence. Note that the mapping from a word to its POS tag is not always deterministic; for example, the word “run” can function as both a noun and a verb.
- In information retrieval, POS tags can substitute words for stop word filtering. For example, we keep “IT” if it is a noun but remove it if it is a pronoun. In query processing, we analyze a query's structure, especially in questions. This analysis helps to extract details, and to query directly against structured metadata instead of relying solely on keyword-based search. For instance, the question “Who is Albert Einstein?” can be split into a query word “who”, a verb “is”, and a name “Albert Einstein”. Using POS tagging, we can infer that the user seeks a person named “Albert Einstein”. Rather than a full-text search, we query a ‘people’ database for structured information.
- A treebank is a text corpus where sentences are parsed and annotated to depict their syntactic structure. These trees also convey information about word-to-word grammatical relationships and hierarchical sentence composition. Treebanks offer labeled data to aid algorithms in learning grammatical structures and associating POS tags with words in sentences.
- A closely related issue is Named Entity Recognition (NER), as seen with “Albert Einstein” in the previous example. Typical categories for NER classification include names, locations, organizations, and currencies. Typically, NER terms are not found in dictionaries, and their frequencies and occurrences vary over time. Identifying a term (or n-gram) as an NER helps us to infer the user's intent. In some cases, NER searches are common, such as in online shops, where proper indexing is crucial. Instead of learning valuable bi-grams to add to the dictionary, we can generate them using simple rules based on NER tags. For instance, if we encounter two names (two consecutive NER-people tags), we can index them both individually and as a bi-gram (or tri-gram if three consecutive names appear). POS tags provide additional insights into the roles of names, as seen in sentences like “How to drive from Basel to Luzern”, “I am in Basel and want to drive to Luzern”, or “How to drive to Luzern from Basel”.
- Let’s look at a few common implementation for POS and NER tagging, and the algorithms behind them.

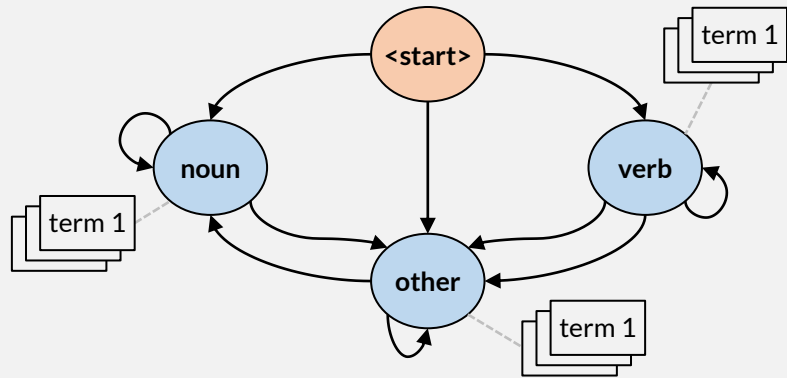
- **Rule-based POS tagging:** These algorithms use predefined rules based on context to assign POS tags to words. In machine learning models, simplified versions of these rules are often integrated as fallback or pre-filtering mechanisms to enhance accuracy for common patterns. For instance, in English:

If a word ends with the suffix “ing” and the stem (without “ing”) is a recognized verb, assign the tag VERB.

This rule effectively handles many regular gerund forms like “hearing” and “walking”, correctly ignores “thing” and “king”, but can not handle “running” or “swimming” correctly (though it is straightforward to add a secondary rule for such cases). While simple and fast in processing, each language requires a fresh set of rules.

- **Stochastic POS Tagging:** A hidden Markov model (HMM), is a probabilistic technique applied in multiple domains. It is a graphical representation comprising states, their probabilistic transitions, and emitted symbols at each state. It consists of four essential components:
  - **Hidden States:** These represent unobservable internal states that capture the system's behavior. They form a Markov chain, where transitions between states depend solely on the current state and are independent of previous states.
  - **Observations:** These are symbols or events linked to hidden states, emitted during transitions between one hidden state and another. However, uncertainty often exists regarding which symbol is emitted because each hidden state can produce multiple outcomes (according to a probability distribution), and an observable symbol or event can be emitted by multiple hidden states.
  - **Transition Probabilities:** Usually represented as a transition matrix, these probabilities indicate the likelihood of moving from one hidden state to another. They are learned from training data, often using maximum likelihood estimation. In the transition matrix, rows correspond to the current state, and columns correspond to the next state.
  - **Emission Probabilities:** These are probabilities associated with each hidden state, indicating the likelihood of generating specific observations or emissions when in that state. These probabilities help define how likely it is for a hidden state to produce particular observable outcomes. Emission probabilities are often represented as an emission matrix or emission probability distribution, with rows corresponding to the state, and columns representing the observations.

- For simplicity, let's consider three POS tags: nouns, verbs, and others. The structure of a Hidden Markov Model (HMM) for POS tagging is as follows:



Transition	noun	verb	other
<start>	0.4	0.2	0.4
noun	0.2	0.5	0.3
verb	0.3	0.2	0.5
other	0.2	0.2	0.6

Emission	term 1	...	term n
noun	0	...	0.004
verb	0	...	0.001
other	0.01	...	0

Each POS tag is a hidden state, including a start state that marks the beginning of sentence processing. The transition matrix specifies the probability of transitioning between hidden states. These probabilities can be learned by using maximum likelihood estimation based on state transitions in the training data. For a transition probability from state  $s_i$  to  $s_{i+1}$ , denoted as  $P(s_{i+1}|s_i)$ , and the count of such state transitions in the training data as  $C(s_i, s_{i+1})$ , the maximum likelihood estimate for  $P(s_{i+1}|s_i)$ ,  $\forall 1 \leq i < m$  is given by:

$$P(s_{i+1}|s_i) = \frac{C(s_i, s_{i+1})}{\sum_{j=1}^m C(s_i, s_j)} \quad \text{with smoothing (small } \epsilon): \quad P(s_{i+1}|s_i) = \frac{C(s_i, s_{i+1}) + \epsilon}{\sum_{j=1}^m C(s_i, s_j) + m \cdot \epsilon}$$

Where  $m$  represents the total number of states. The smoothing variant serves to avoid zero-values in the transition matrix, which can cause numerical problems when performing calculations with logarithms in the Viterbi algorithm. Additionally, it enables the model to handle transitions that were not observed in the training data but are encountered when processing new sentences. Similarly, we compute the maximum likelihood estimates for the probability  $P(t_k|s_i)$  of term  $t_k$  being emitted at state  $s_i$  based on counts denoted as  $C(s_i, t_k)$ :

$$P(t_k|s_i) = \frac{C(s_i, t_k)}{\sum_{k=1}^n C(s_i, t_k)} \quad \text{with smoothing (small } \epsilon): \quad P(t_k|s_i) = \frac{C(s_i, t_k) + \epsilon}{\sum_{k=1}^n C(s_i, t_k) + m \cdot \epsilon}$$

- Using the trained Hidden Markov Model, we can apply the Viterbi Algorithm to determine the most probable state transitions for a given sentence's observed term sequence. These states can then be linked to the terms in the sequence to assign them the corresponding POS tags.
- **Transformation-based POS tagging** builds upon rule-based POS tagging by iteratively correcting errors. It starts with an initial, simple, and hand-crafted rule-based tagging, which is compared to training data to find errors. Transformation rules, either learned from training data or manually created based on observed patterns, are then used to correct these errors. This error identification and rule application process is repeated until a maximum number of iterations is reached or no more errors are found. During sentence analysis, the initial tagging and all transformation rules are consistently applied to generate the final POS tagging for the sequence.

After applying the “ing” rule for verb detection, we notice that in some cases, the gerund form of a verb can also function as a noun. For example, in the sentence “The swimming was nice”, “swimming” is a noun but was initially tagged as a verb. To address this, we introduce a straightforward transformation rule: verbs that follow articles or adjectives should be reclassified as nouns.
- **Deep learning POS tagging**, on the other hand, employs neural networks to automatically learn and predict POS tags for words in a text. Instead of relying on hand-crafted rules or transformations, deep learning models leverage large datasets to capture complex patterns and relationships between words and their corresponding POS tags. These models use layers of neurons and sophisticated architectures to process sequential data, making them particularly effective for tasks like POS tagging, where the order of words in a sentence is crucial. Deep learning approaches have achieved remarkable accuracy in various natural language processing tasks, including POS tagging, and continue to be a cornerstone of modern NLP research and applications.

Modern taggers utilize a transformer-based architecture. In this architecture, the input sequence represents the sentence and is transformed into embeddings and including positional encoding. The transformer architecture then maps this sequence to a POS (Part-of-Speech) sequence. Training these models with POS-tagged sentences enables the neural network to learn its parameters. These specialized models are optimized for POS-tagging and are not suitable for other tasks.
- All these approaches have a common limitation: they usually support only one language. While multi-language POS-tagging is possible, it is more advisable to use a language-optimized model when the language of the sentence is known. Most of these approaches achieve high accuracy, often exceeding 99%, across a wide range of sentences.

- NLTK employs a transformation-based POS tagger. The process starts with sentence tokenization, followed by a dedicated classifier that predicts the POS tags for the tokens. It is crucial to input the entire sentence into the POS tagger because words that can assume various grammatical roles might be misclassified otherwise (e.g., "running" as a gerund form and "running" as a noun). By considering the complete sentence context, NLTK can provide more accurate POS tags. Here is the Python code to obtain and list the tokens:

```
tokens = nltk.word_tokenize(text_en)
# tagset = None for standard, or tagset = 'universal'
tagged_tokens = nltk.pos_tag(tokens, tagset=tagset)
ner_chunks = [chunk for chunk in nltk.ne_chunk(tagged_tokens) if hasattr(chunk, 'label')]
```

`nltk` supports different tag sets. The standard tag set is more detailed and depicted on the left side. The universal tag set focuses on a few main categories as shown on the right side. The standard set is often used for deep NLP tasks to construct parse trees which allows the extraction of context and the transformation of sentences.

POS	Description (standard)
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun

Proper nouns are specific people, places, things.

POS	Description (standard)
PRP\$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VCN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP\$	Possessive wh-pronoun
WRB	Wh-adverb

with NLTK, use `nltk.help.upenn_tagset()`

WH-words are: where, what, which, when, ...

POS	Description (universal)
ADJ	adjective
ADP	adposition
ADV	adverb
CONJ	conjunction
DET	determiner, article
NOUN	noun
NUM	numeral
PRT	particle
PRON	pronoun
VERB	verb
.	punctuation marks
X	other

- Named Entity Recognition (NER) is a transformation based on POS tagging. It involves collapsing individual tokens or groups of tokens into a single named entity using an entity database. These entities can be names of people, product brands, companies, non-governmental organizations, locations, currencies, and more. In NLTK, the POS tagged tokens are processed with the `ne_chunk` function to obtain NER tags. These NER tags enable the extraction of valuable contextual information, such as person names, locations, or product names. When applied to a query in question form, it helps to better understand the user's intent and to optimize search results:
  - o When searching for a name, such as “Who is Albert Einstein?”, prioritize web pages like Wikipedia, IMDb, Musicbrainz, and sports sites that users commonly visit to gather information about prominent individuals.
  - o When a query includes time, date, or age information, like “Who won the F1 race last weekend?”, enhance the visibility of news articles or utilize the extracted time/date to conduct a temporal range query.
  - o When a query mentions a location, such as “What to do in Basel?”, prioritize regional content and provide a map of the named location to assist users with navigation.
  - o When a query involves product brands, like “Where is the latest iPhone available?”, boost advertisements and shopping sites, conduct a product search to offer a "best price" view, or provide recommendations for buyers.
- Chunking is a versatile technique that involves creating non-overlapping phrases using a defined grammar. For example, the grammar `NP: {<DT>?<JJ>*<NN>}` combines articles, adjectives, and nouns into a single group, facilitating the understanding of term relationships for more effective searching. For instance, “a red car” would form a parse tree that links the adjective “red” with the noun “car”. More intricate grammars enable the dissection of sentences into smaller components, allowing for reasoning about context and sentence meaning through additional dependency information between terms.
  - o A good online dem with deep NLP capabilities is available here: <https://corenlp.run>
- To analyze sentence structure, we require a grammar similar to that used in programming languages. Unlike programming languages, natural language grammar is imperfect and riddled with ambiguities, making it challenging for both humans and machines to grasp context. Grammar alone cannot resolve these ambiguities; context plays a crucial role in their resolution (as discussed in the preceding sentence).



- **spaCy** uses a neural network to predict POS and NER tags, however with different tag names than **nltk**. The left table below shows the POS tags, and right table the NER tags. The code is also simple:

```
nlp_spacy = spacy.load('en_core_web_sm')
tokens = nlp_spacy(text)
tagged_tokens = [(t.text, t.pos_) for t in tokens]
ner_entities = [(e.text, e.label_) for e in tokens.ents]
```

**spaCy** also offers support for various languages. Refer to their documentation to choose the suitable model.

POS	Description	Examples
ADJ	adjective	big, old, green, incomprehensible, first
ADP	adposition	in, to, during
ADV	adverb	very, tomorrow, down, where, there
AUX	auxiliary	is, has (done), will (do), should (do)
CONJ	conjunction	and, or, but
CCONJ	coordinating conjunction	and, or, but
DET	determiner	a, an, the
INTJ	interjection	psst, ouch, bravo, hello
NOUN	noun	girl, cat, tree, air, beauty
NUM	numeral	1, 2017, one, seventy-seven, IV, MMXIV
PART	particle	's, not,
PRON	pronoun	I, you, he, she, myself, themselves
PROPN	proper noun	Mary, John, London, NATO, HBO
PUNCT	punctuation	., (, ), ?
SCONJ	subordinating conjunction	if, while, that
SYM	symbol	\$, %, \$, ©, +, -, ×, ÷, =, :, 😊
VERB	verb	run, runs, running, eat, ate, eating
X	other	sfpkdspxmsa
SPACE	space	

NER	Description
PERSON	People, including fictional.
NORP	Nationalities or religious or political groups.
FAC	Buildings, airports, highways, bridges, etc.
ORG	Companies, agencies, institutions, etc.
GPE	Countries, cities, states.
LOC	Non-GPE locations, mountain ranges, bodies of water.
PRODUCT	Objects, vehicles, foods, etc. (Not services.)
EVENT	Named hurricanes, battles, wars, sports events, etc.
WORK_OF_ART	Titles of books, songs, etc.
LAW	Named documents made into laws.
LANGUAGE	Any named language.
DATE	Absolute or relative dates or periods.
TIME	Times smaller than a day.
PERCENT	Percentage, including "%".
MONEY	Monetary values, including unit.
QUANTITY	Measurements, as of weight or distance.
ORDINAL	"first", "second", etc.
CARDINAL	Numerals that do not fall under another type.

- Finally, the transformers library offers two pipelines for extracting POS and NER tags using trained neural networks. It also supports fine-tuning of NER tags to adapt to specific document collections and scenarios. Since transformers models are continuously advancing, we present the general code structure here and recommend visiting the Hugging Face website to access the latest models for these pipelines:

```
nlp_bert = pipeline("token-classification",  
                    model="vblagoje/bert-english-uncased-finetuned-pos",  
                    aggregation_strategy="max")  
tokens = nlp_bert(text)  
tagged_tokens = [(token['word'], token['entity_group']) for token in tokens]  
  
nlp_bert = pipeline("ner",  
                    model="dslim/bert-base-NER",  
                    aggregation_strategy="max")  
tokens = nlp_bert(text_ner)  
ner_entities = [(t['word'], t['entity_group']) for t in tokens]
```

The aggregation strategy combines tokens, typically sub-words, to reconstruct words or n-grams, especially for names. Without an aggregation strategy, the model assigns entity values to individual model tokens, potentially splitting words into smaller tokens without grouping them into entities.

Please consult the model description to determine the specific POS and NER tags used, as these may vary between different models. In addition to the English version used here, there are models available for other languages as well.

- Example: The table on the right (blue) shows the POS tags for the methods discussed on an English sentence (punctuation and repeating words were removed). On the right side (green), we see the NER tags for the sentence: “Jack Higgins, wearing Nike shoes, deposits £50,000 with BestBank in London at Jermyn Street close to Piccadilly Circus”. And finally, the lower, right tables (orange) list the POS tags and their frequency in an English novel.

term	nltk (standard)	nltk (universal)	spaCy	BERT
this	DT	DET	PRON	PRON
was	VBD	VERB	AUX	AUX
a	DT	DET	DET	DET
lofty	JJ	ADJ	ADJ	ADJ
chamber	NN	NOUN	NOUN	NOUN
lined	VBN	VERB	VERB	VERB
and	CC	CONJ	CCONJ	CCONJ
littered	VBN	VERB	VERB	VERB
with	IN	ADP	ADP	ADP
countless	JJ	ADJ	ADJ	ADJ
bottles	NNS	NOUN	NOUN	NOUN
broad	NNP	NOUN	ADJ	ADJ
low	JJ	ADJ	ADJ	ADJ
tables	NNS	NOUN	NOUN	NOUN
were	VBD	VERB	AUX	AUX
scattered	VBN	VERB	VERB	VERB
about	IN	ADP	ADP	ADV
which	WDT	DET	PRON	PRON
bristled	VBD	VERB	VERB	VERB
retorts	NNS	NOUN	NOUN	NOUN
little	JJ	ADJ	ADJ	ADJ
their	PRP\$	PRON	PRON	PRON
blue	JJ	ADJ	ADJ	ADJ
there	EX	DET	PRON	PRON
only	RB	ADV	ADV	ADV
one	CD	NUM	NUM	NUM

NER entity	nltk	spaCy	BERT
50,000		MONEY	
BestBank	ORGANIZATION	ORG	ORG
Jack Higgins	PERSON	PERSON	PER
Jermyn Street	FACILITY	FAC	LOC
London	GPE	GPE	LOC
Nike	PERSON	ORG	MISC
Piccadilly Circus	PERSON	ORG	LOC

POS (nltk, standard)	freq	POS (nltk, universal)	freq	POS (spaCy)	freq
NN	6906	NOUN	10953	PUNCT	8080
IN	5602	VERB	9292	NOUN	7433
DT	4620	PRON	5653	PRON	6745
PRP	4031	ADP	5602	VERB	6025
VBD	3459	.	5550	ADP	5073
JJ	3070	DET	5260	DET	4451
,	2959	ADJ	3234	SPACE	3844
NNP	2558	ADV	2520	AUX	3055
.	2433	CONJ	1786	ADJ	2877
RB	2171	PRT	1359	ADV	2265
CC	1786	NUM	345	CCONJ	1698
VB	1730	X	49	PROPN	1648
NNS	1470			SCONJ	1449
PRP\$	1358			PART	993

## 4.6 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. In our 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  $3.45 * \text{airplane} + 0.34 * \text{bird}$ . We might interpret this as “flying”. However, the model can also generate a mathematically justified description as  $3.45 * \text{airplane} + 0.34 * \text{flower}$ , 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  $\mathbf{v}_i$  of length 1, satisfying the equation for matrix  $\mathbf{A}$ :

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

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$$

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  $r \times r$  matrix) containing the eigenvectors and  $\mathbf{\Lambda}$  (an  $r \times r$  diagonal matrix) containing the corresponding eigenvalues.

- 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  $m \times n$ -matrix of rank  $r$ . There exists an  $r \times r$ -diagonal matrix  $\mathbf{S}$ , an orthonormal  $m \times r$ -matrix  $\mathbf{U}$ , and an orthonormal  $n \times r$ -matrix  $\mathbf{V}$  such that:

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

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  $\mathbf{A}^T\mathbf{A}$  and  $\mathbf{A}\mathbf{A}^T$ :

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

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

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

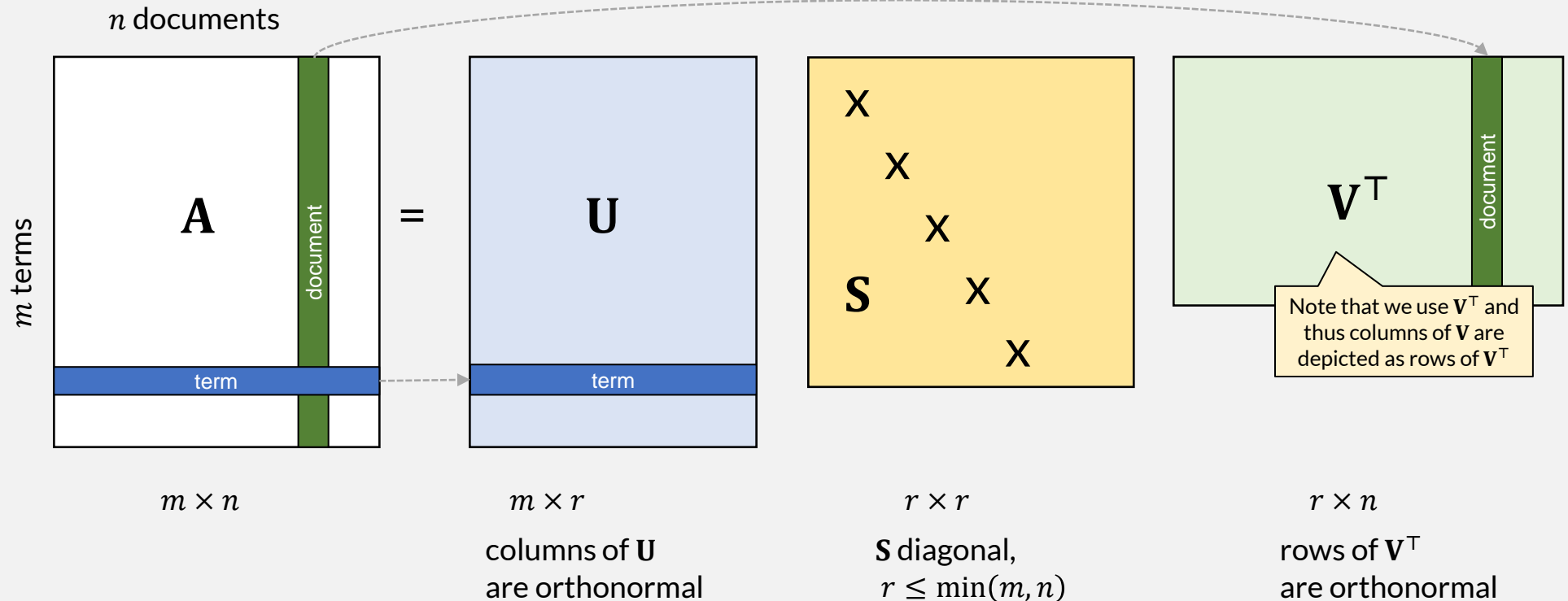
$$\mathbf{A} = s_1(\mathbf{u}_1\mathbf{v}_1^T) + s_2(\mathbf{u}_2\mathbf{v}_2^T) + \dots + s_r(\mathbf{u}_r\mathbf{v}_r^T)$$

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.

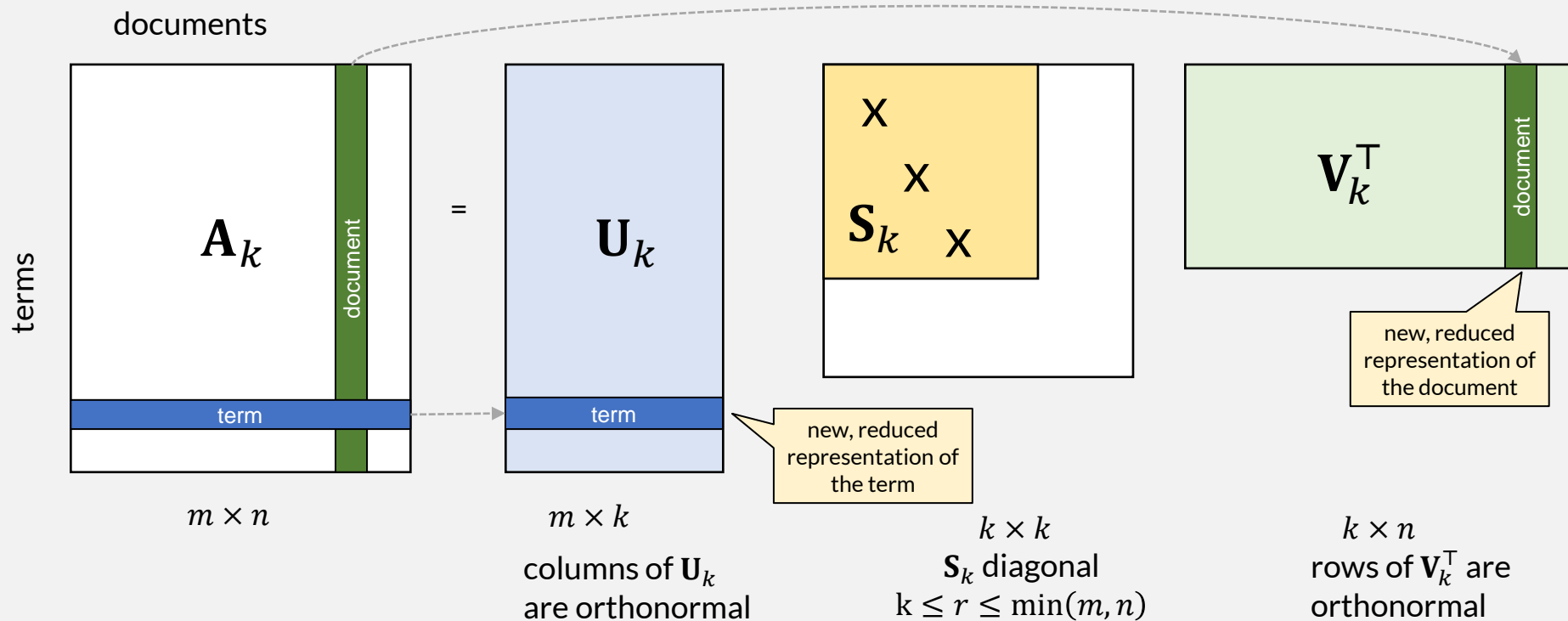
## 4.6.1 Application in Text Retrieval

- In text retrieval, we can apply the **Singular Value Decomposition (SVD)** to the document-term matrix **A**, typically using *tf · idf* weighted components. SVD decomposes **A** into matrices **U**, **S**, and **V**, reducing them to the intrinsic rank  $r \leq \min(m, n)$ . Matrix **U** represents the  $m$  terms of the vocabulary in an  $r$ -dimensional space, **S** contains singular values (usually sorted by decreasing value on the diagonal), and **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 **A**. Removing singular values and their corresponding columns in **U** and **V** reduces the dimensionality of the new term and document representations.

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



- **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  $\mathbf{V}_k^T$  (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  $\mathbf{V}_k^T$  (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 approach that delays the need for renewed SVD computations.

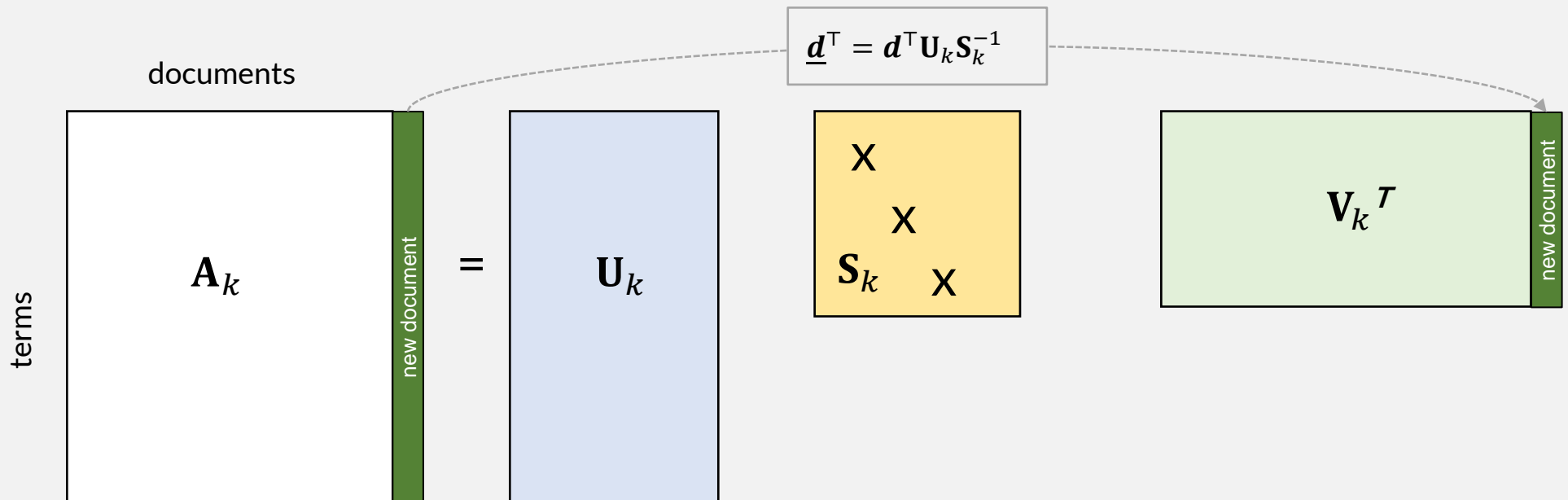


- **Inserting new documents (approximation):** Using the approximate representation  $\mathbf{A}_k$  from the reduced singular value decomposition, we can derive a mapping from the original term space to the topic space in three steps as follows: 1) transpose both sides of the equation, 2) multiply first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

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

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  $\underline{\mathbf{t}}^\top = \mathbf{t}^\top \mathbf{V}_k \mathbf{S}_k^{-1}$ . However, because a new term appears only in a new document, the term vector  $\mathbf{t}$  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  $\underline{\mathbf{t}}$  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, akin to newly added documents, to the reduced topic space:

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

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

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

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

- The primary difference is that we are now comparing two dense vectors. Since the query vector usually has non-zero values in all dimensions, the inverted index method cannot be used to expedite the search. Instead, we must evaluate similarity for each document and arrange them by score. Despite the lower dimensionality of LSI vectors compared to the original document vectors, we need to process substantially more data and cannot eliminate documents as efficiently as with inverted indexes. We will consider indexing methods for dense vector search in a later chapter. For now, it is vital to understand that we must strike a balance between having more topics for a richer semantic representation of the corpus's latent topics and fewer dimensions to minimize retrieval costs.
- While it is feasible to reuse the mapping from document vectors to a compact, lower-dimensional representation when using the same vocabulary in different collections, the common approach is to apply LSI separately for each corpus. This is because LSI not only learns the vital topics from terms but also their context within the documents. Consequently, the mapping for an IT article collection will differ significantly from that of news articles. Employing a generic mapping would diminish topic quality and potentially harm retrieval performance.

## 4.6.2 A Simple Example with LSI

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

c1	<a href="#">Human</a> machine interface for Lab ABC <a href="#">computer</a> applications
c2	A survey of user opinion of <a href="#">computer</a> system response time
c3	The EPS user interface management system
c4	System and <a href="#">human</a> 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 observe that the collection comprises two distinct document subgroups: one focusing on human interfaces and the other on graph algorithms. When we query this collection with “[human-computer interaction](#)”, we notice that not all relevant documents c1...c5 contain the query terms. With classical retrieval models, query expansion is necessary to introduce synonyms and broader terms for improved search quality (e.g., “human” to “user”, “computer” to “system”). Now, let's assess how LSI performs in 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 **A** below. The results are shown below.

**U =**

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

**S =**

3.3409								
	2.5417							
		2.3539						
			1.6445					
				1.5048				
					1.3064			
						0.8459		
							0.5601	
								0.3637

**V<sup>T</sup> =**

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

- To simplify document presentation, we choose  $k = 2$  and adjust all matrices accordingly. The matrix  $V_k^T$  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.

$U_k$	$S_k$	$V_k^T$
<div>0.2214 -0.1132</div> <div>0.1976 -0.0721</div> <div>0.2405 0.0432</div> <div>0.4036 0.0571</div> <div>0.6445 -0.1673</div> <div>0.2650 0.1072</div> <div>0.2650 0.1072</div> <div>0.3008 -0.1413</div> <div>0.2059 0.2736</div> <div>0.0127 0.4902</div> <div>0.0361 0.6228</div> <div>0.0318 0.4505</div>	<div>3.3409</div> <div>2.5417</div>	<div>0.1974 0.6060 0.4629 0.5421 0.2795 0.0038 0.0146 0.0241 0.0820</div> <div>-0.0559 0.1656 -0.1273 -0.2318 0.1068 0.1928 0.4379 0.6151 0.5299</div>

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  $A$  with  $k = 2$ . While it may not closely resemble the original document-term matrix,  $A_k$  shown below is the best rank-2 representation for  $A$  under the Frobenius norm.

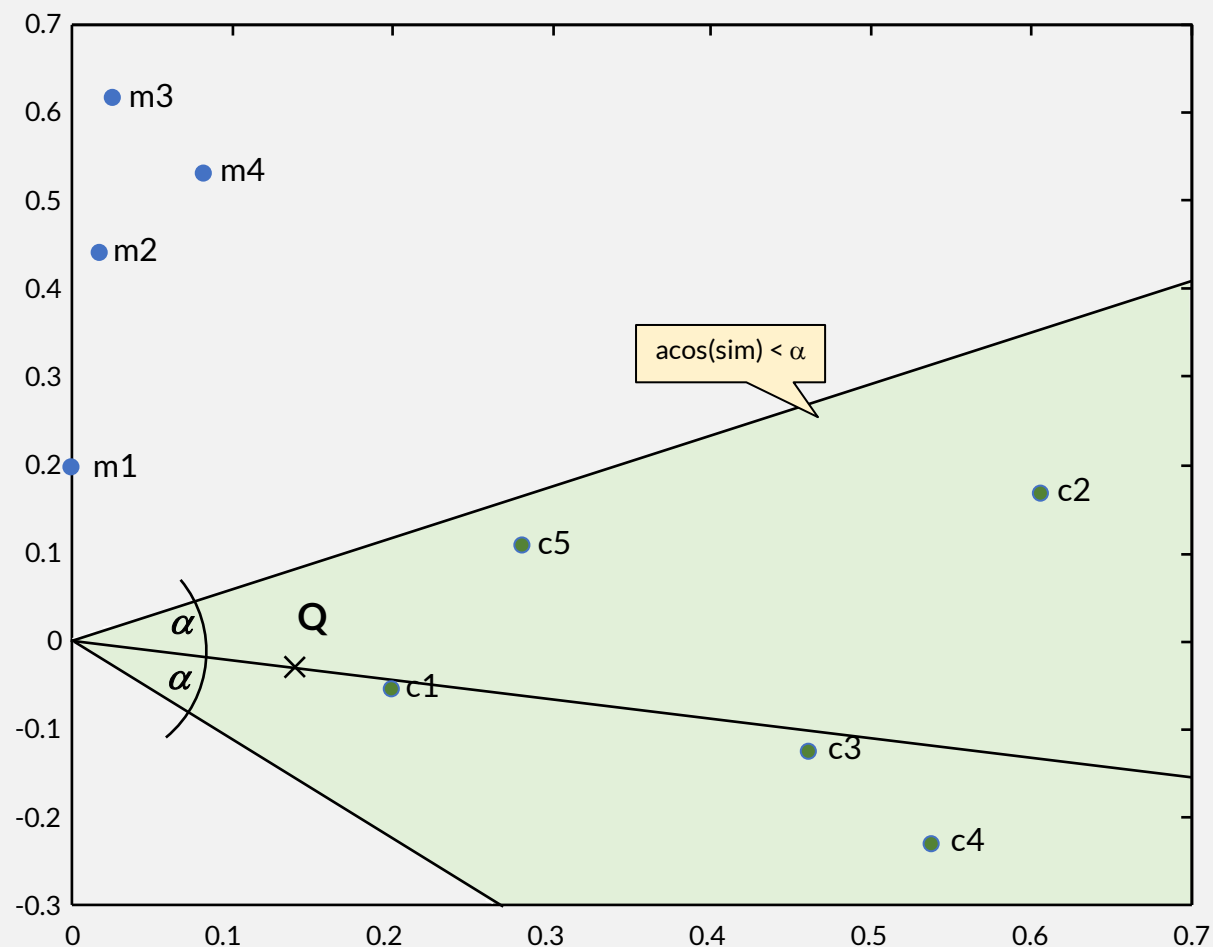
$A_k =$	$q$		$\underline{q}$
<div>0.1621 0.4005 0.3790 0.4676 0.1760 -0.0527 -0.1151 -0.1591 -0.0918</div> <div>0.1406 0.3698 0.3290 0.4004 0.1650 -0.0328 -0.0706 -0.0968 -0.0430</div> <div>0.1524 0.5050 0.3579 0.4101 0.2362 0.0242 0.0598 0.0869 0.1240</div> <div>0.2580 0.8411 0.6057 0.6974 0.3923 0.0331 0.0832 0.1218 0.1874</div> <div>0.4488 1.2344 1.0509 1.2658 0.5563 -0.0738 -0.1547 -0.2096 -0.0489</div> <div>0.1596 0.5817 0.3752 0.4169 0.2765 0.0559 0.1322 0.1889 0.2169</div> <div>0.1596 0.5817 0.3752 0.4169 0.2765 0.0559 0.1322 0.1889 0.2169</div> <div>0.2185 0.5496 0.5110 0.6281 0.2425 -0.0654 -0.1425 -0.1966 -0.1079</div> <div>0.0969 0.5321 0.2299 0.2118 0.2665 0.1368 0.3146 0.4444 0.4250</div> <div>-0.0613 0.2321 -0.1389 -0.2656 0.1449 0.2404 0.5461 0.7674 0.6637</div> <div>-0.0647 0.3353 -0.1456 -0.3014 0.2028 0.3057 0.6949 0.9766 0.8487</div> <div>-0.0431 0.2539 -0.0967 -0.2079 0.1519 0.2212 0.5029 0.7069 0.6155</div>	<div>1</div> <div>0</div> <div>1</div> <div>0</div> <div>0</div> <div>0</div> <div>0</div> <div>0</div> <div>0</div> <div>0</div> <div>0</div> <div>0</div>	$U_k S_k^{-1}$	<div>0.1382</div> <div>-0.0276</div>

- The right side visualizes the document collection. We notice that topic 1 (x-dimension) aligns more with documents c1 to c5, while topic 2 (y-dimension) aligns with documents m1 to 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  $U_k$  matrix (first column).

$0.64 \cdot \text{system} + 0.40 \cdot \text{user} +$   
 $0.30 \cdot \text{eps} + 0.27 \cdot \text{time} +$   
 $0.27 \cdot \text{response} + 0.24 \cdot \text{computer}$

and for topic 2:

$0.62 \cdot \text{graph} + 0.49 \cdot \text{trees} +$   
 $0.45 \cdot \text{minors} + 0.27 \cdot \text{survey}$



## 4.7 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, 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.

## 4.7.1 Word2vec, GloVe, and fastText

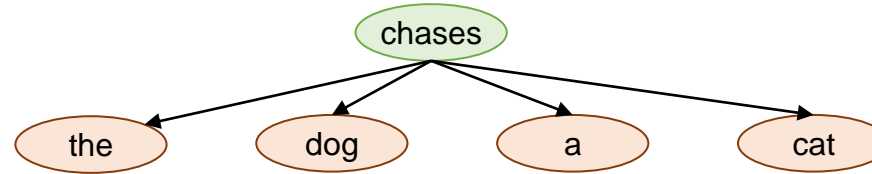
- Word2Vec, initially introduced in 2013 by Mikolov et al., aims to optimize the mapping of vocabulary (words) to a  $d$ -dimensional vector space. This mapping relies on context windows around words and is trained in a self-supervised manner, eliminating the need for labeling. Word2Vec offers two variations:
  - 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:

$$P(\text{the dog the cat} \mid \text{chases}) = P(\text{the} \mid \text{chases}) \cdot P(\text{dog} \mid \text{chases}) \cdot P(\text{a} \mid \text{chases}) \cdot P(\text{cat} \mid \text{chases})$$

We can illustrate this relationship graphically as follows:



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

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

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:

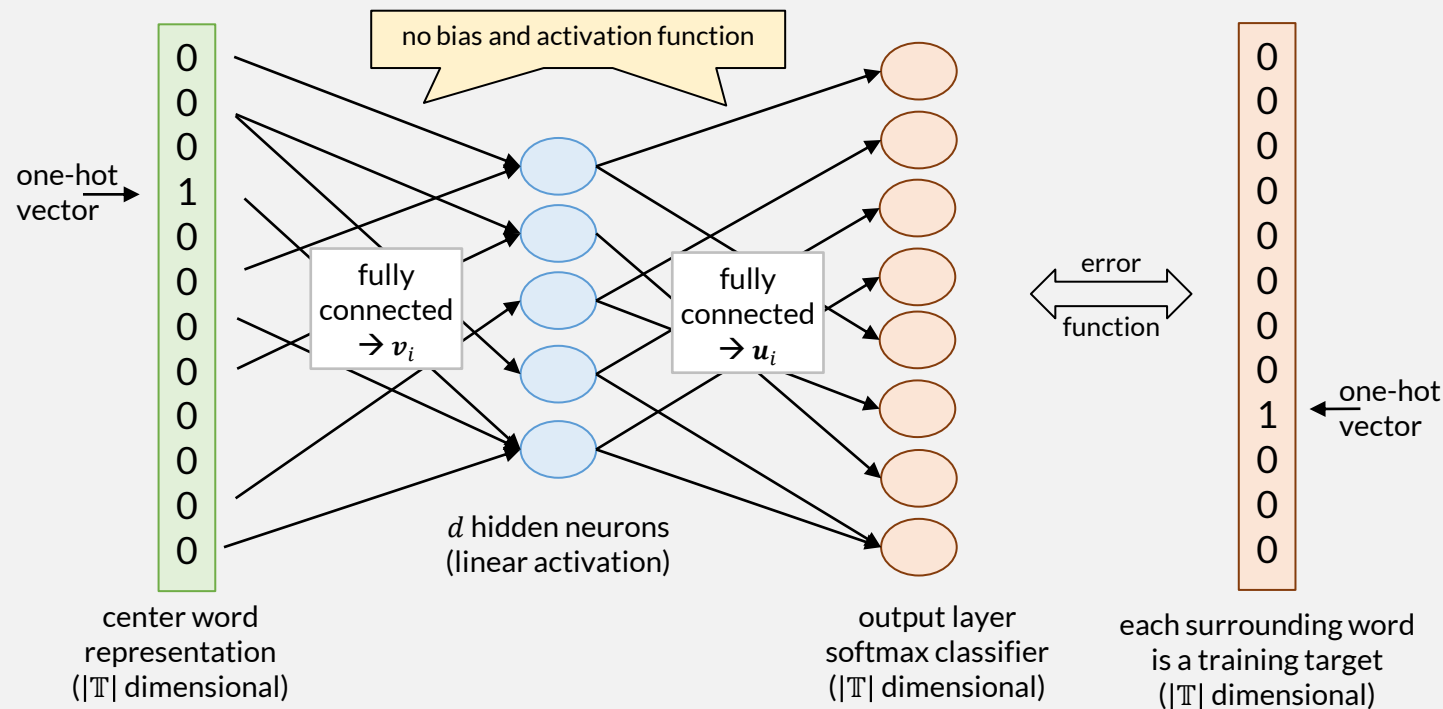
$$\prod_{i=m+1}^{n-m} \prod_{j=i-m, j \neq i}^{i+m} P(w_j \mid w_i)$$

with a context window of size  $2m + 1$ . The objective is to find vectors  $\mathbf{v}_i \in \mathbb{R}^d$  and  $\mathbf{u}_i \in \mathbb{R}^d$  that maximize the likelihood function, and we then can use the mapping  $w_i \rightarrow \mathbf{v}_i$  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  $v_i$ . 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  $u_i$ . 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:
  - o We enumerate all windows of size  $2m + 1$  in the corpus. To avoid incorrect associations across sentence boundaries, we can split texts into sentences and ensure that windows are confined within sentences.
  - o 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.
  - o 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.
  - o 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  $u_i$  and  $v_i$ . This process is iterated until the loss function reaches a sufficiently low value. The outcome is a vocabulary-to-vector mapping, represented as  $v_i$ .
- 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.

```
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
```

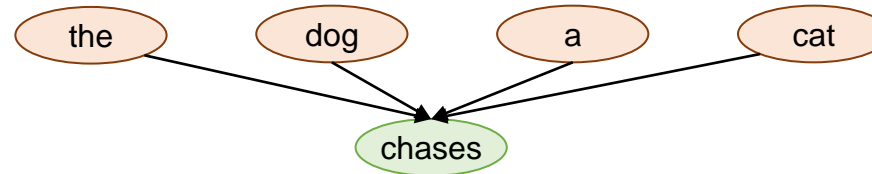
```
# sketch of training process
model = SkipGramModel(vocab_size=len(vocab), embedd_size=100)
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()

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:

$$P(\text{chases} \mid \text{the dog the cat})$$

We can illustrate this relationship graphically as follows:



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

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

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:

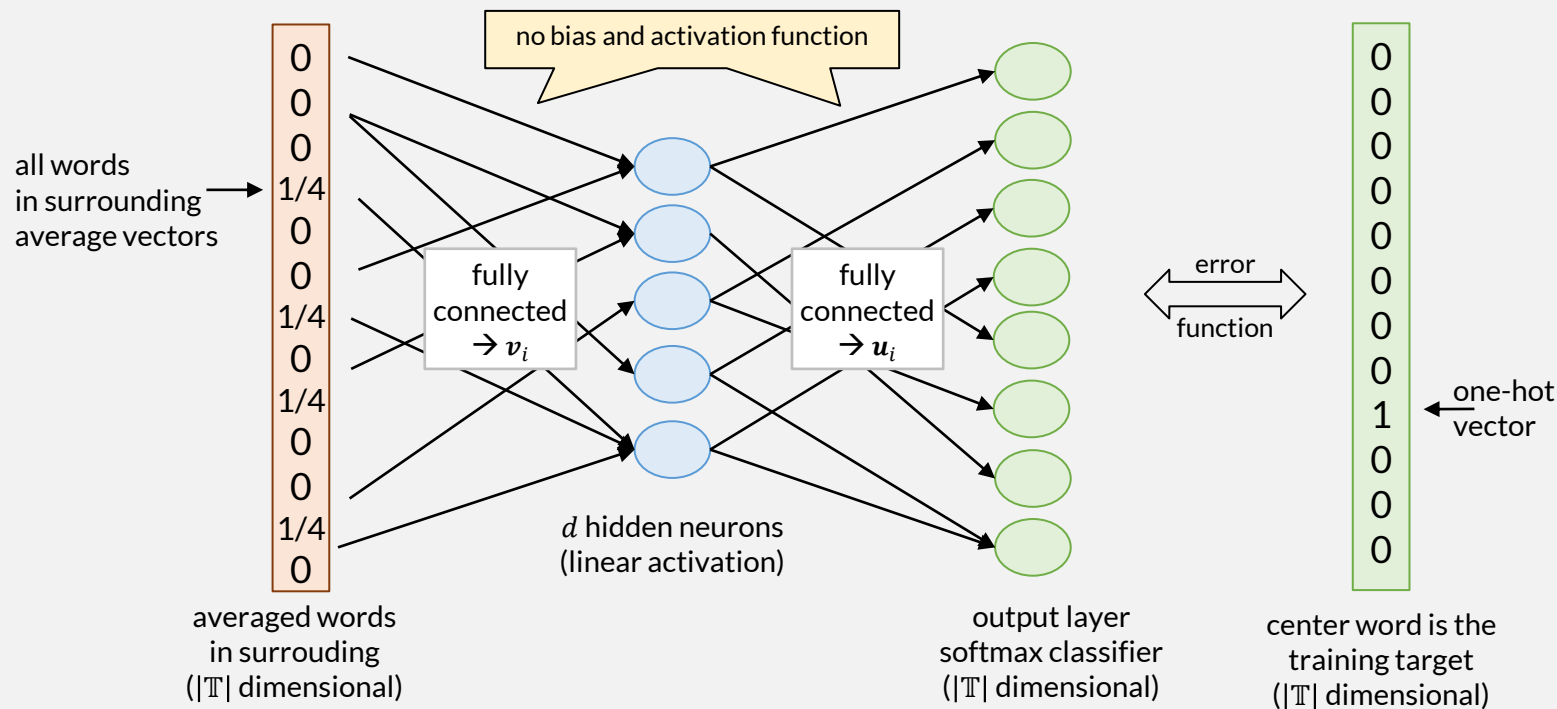
$$\prod_{i=m+1}^{n-m} P(w_c \mid w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m})$$

with a context window of size  $2m + 1$ . The objective is to find vectors  $\mathbf{v}_i \in \mathbb{R}^d$  and  $\mathbf{u}_i \in \mathbb{R}^d$  that maximize the likelihood function, and we then can use the mapping  $w_i \rightarrow \mathbf{v}_i$  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-gra-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  $\mathbf{v}_i$ . 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  $\mathbf{u}_i$ . 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:
  - o We enumerate all windows of size  $2m + 1$  in the corpus as with the skip-gram model.
  - o 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).
  - o 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  $u_i$  and  $v_i$ . This process is iterated until the loss function reaches a sufficiently low value. The outcome is a vocabulary-to-vector mapping, represented as  $v_i$ .
- 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

```
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
```

```
# 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()
```

- **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 occurrences of 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

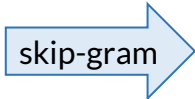
- We introduce vectors  $u_i$  to represent words in a  $d$ -dimensional space and vectors  $v_k$  to represent context words in a  $d$ -dimensional space (adopting notation similar to word2vec, deviating from the original paper's notation). Additionally, we require biases  $b_i$  for words and  $c_j$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

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

The function  $f(x) = \min((x/x_{max})^\alpha, 1)$  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, along with the special “<chases>” sub-word. 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:

$$v_i = \sum_{g \in \mathbb{Z}} z_g$$



$$P(w_s | w_c) = \frac{e^{u_s^T v_c}}{\sum_{i \in \mathbb{T}} e^{u_i^T v_c}}$$

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,  $m$  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  $3 \times 10^8$  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.



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

- 1) To train your model, **gensim** offers both word2vec and fastText, as shown in the code. To enhance the quality of embeddings, it is essential to use large collections of sentences. The example with movie reviews may not yield robust embeddings.
- 2) The **gensim** package provides various pre-trained embeddings, including GloVe vectors, which offer generic embeddings primarily for English. However, these models are trained on a wide range of topics and may not produce optimal results for collection-specific embeddings.
- 3) Embeddings are commonly used to explore word relationships. We begin by creating vector representations (e.g., for 'cat') and utilize these vectors to identify the most similar terms, where similarity is based on shared contexts (e.g., 'dog' is highly similar). Additionally, we can perform mathematical operations on these vectors, as seen in 'Paris + Germany - France = Berlin.' Another valuable application involves detecting words that are outliers in a given list, indicating they do not belong with the other list members.

**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 `doesn't 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>

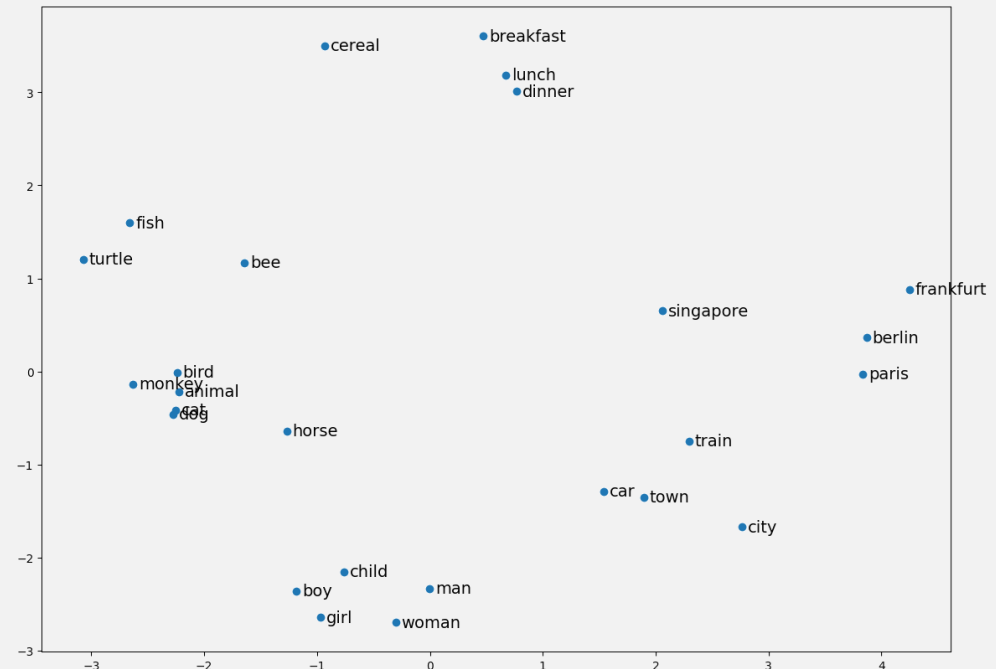
```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import gensim.downloader

# 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]

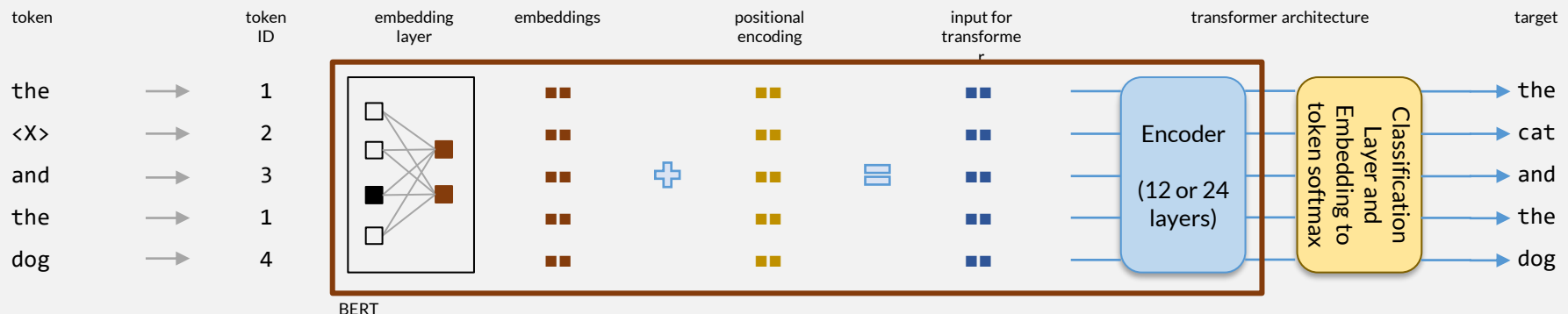
# apply a PCA to map to 2 dimensions
pca = PCA(n_components=2)
result = pca.fit_transform(vectors)

# 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()
```



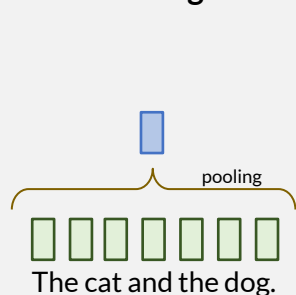
## 4.7.2 Transformer-based Embeddings

- Previously in this chapter, we have already explored the fundamental structure of modern transformer-based language models. The diagram below provides a visual representation of the various layers with the example of BERT, an early version of a bi-directional transformer-based encoder model:
  - BERT uses word pieces to form tokens from text. Tokens are represented by unique token IDs, and the input sequence consists of a vector of token IDs. These input vectors are then passed through the embedding layer. While we conceptually describe embedding layers as transforming one-hot vectors into lower-dimensional, dense vectors, the implementation is a column lookup using the token ID in the weight matrix eliminating the need to store and compute with one-hot vectors. Note that embedding layers do not include bias or activation functions.
  - The output, which is a sequence of  $d$ -dimensional embeddings, is added to a positional encoding represented by a sinusoidal signal function. This positional encoding is crucial because the transformer model relies on fully connected layers rather than recurrent network structures. Positional encoding helps the model to understand the order of tokens within the sequence which otherwise would be lost.
  - The encoded vectors are subsequently given to the transformer encoder model, which can comprise multiple layers (BERT base with 12 layers; BERT large with 24 layers). The encoder output passes a fully connected classification layer, and the embeddings matrix together with a softmax function predicts the output token ID.
  - In the training process, we simultaneously learn the parameters of both the transformer model and the weights in the embedding layer. One approach to self-supervised training is to randomly mask words in the input sequence (represented as  $\langle X \rangle$  in the figure), and the model is required to predict these masked words. By tackling the more challenging task of predicting masked tokens, the model inherently learns effective token embeddings.

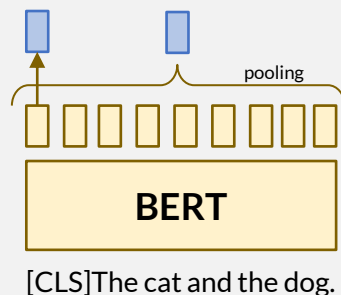


- After the training, we can extract the embedding weights. These weights, in combination with the vocabulary, establish the mapping from word pieces to  $d$ -dimensional vectors. Similar to fastText, we can construct word embeddings by aggregating (typically averaging or pooling) the embeddings of word pieces, also for new words.
- **Sentence Embeddings:** Let's consider the various options to create sentence embeddings (see below):
  - 1) The baseline approach is to first generate embeddings for all tokens in the sentence using methods like word2vec, fastText, or BERT. A pooling strategy, like summing up all embedding vectors or taking maximum values for each dimension overall all embeddings, is then applied. Normalizing sentence embeddings accelerates calculations for cosine similarity. Two sentences are more similar if their dot-product (=cosine similarity if normalized) is larger.
  - 2) Alternatively, we can input a complete sentence into BERT and utilize the encoder's output, either by applying a pooling strategy as previously mentioned or by using the output associated with BERT's special [CLS] token at the sentence's beginning. However, this approach does not consistently perform well because the model wasn't explicitly trained for this task. Also consider limits on input lengths of 512 tokens with BERT (~300 words).
  - 3) Sentence transformers with Bi-Encoders use a BERT model, and fine-tune it using sentence pairs and additional training epochs. The encoder's output for two sentences is passed through a classifier, and the BERT model is fine-tuned to generate encoder output that allows for sentence comparisons. After training, the fine-tuned BERT model can generate sentence embeddings and use the methods from 2) to compute embeddings
  - 4) Cross-encoders are designed for sentence comparison rather than embedding generation. In this architecture, two sentences are inputted, separated by the [SEP] token. The encoder's output for both sentences is processed by a classifier network, and the model learns to determine the similarity between the sentences based on training examples. After training, the model can be used to assess the similarity between two given sentences.

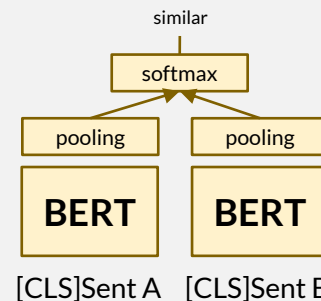
1) Pooling of token embeddings



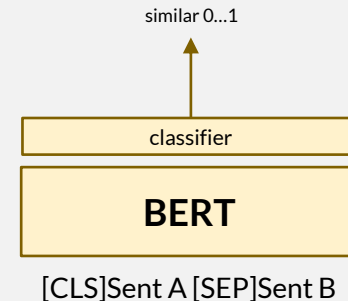
2) Pooling of encoder



3) Training bi-encoder



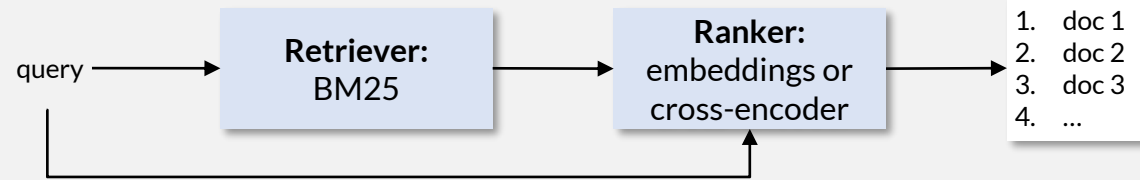
4) Cross-encoder



## 4.7.3 Embeddings in Text Retrieval

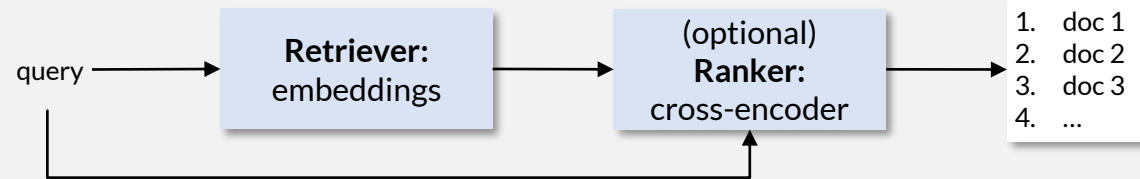
- Embeddings have become increasingly popular for semantic search, also in combination with traditional models such as BM25. We consider 3 models in the following: 1) retriever-ranker, 2) semantic search with retriever-reader, and 3) retrieval augmented generation. Before we consider the models in more details, let's consider different scenarios for an interactive retrieval system:
  - Factoid queries are retrieval queries that seek concise and factual answers to questions like “What is the capital of France?” or “Who invented the iPhone?”. These queries contrast with more complex or open-ended queries that require longer, more detailed responses.
  - Descriptive queries seek to provide a description or explanation of a topic, concept, or term. For example, “What does relativity mean?”, or “How does a democracy work?”. There are many possible ways to answer the question and we prefer to obtain them in form of a narrative or list of points.
  - Comparative queries (or decision making) request to compare two or more entities or concepts. For example, “Compare different mobile phones by features and price”, or “Compare the risk of investing into company A to company B”. This may involve a structured approach to enable decision making.
  - Procedural queries seek step-by-step instructions or guidance on how to perform a task or achieve a goal. For example, “How do I bake a cake?”, and “How can I pass the driver's license exam?”. The result should be a concise instruction with points that lead to the goal.
  - Time-based queries ask questions about time-related events in the past, present or the future. “When was Mozart born?”, or “When does the train to London leave?”. Expected results include time ranges, schedules, or timetables.
  - Opinion queries seek for (subjective) advice, recommendations, or reviews. For example, “What is the best Pizza restaurant in the town?”, or “Which movies shall I watch this evening?”. This may include the preferences, location, time, age and personal preferences to provide a best answer for the user's information need.
- This is not an exhaustive list, but it shows that we must approach each query type differently when generating user answers. Factoid queries benefit from concise answers with references, while descriptive queries need longer, natural text responses. Time-based queries can be cached for repeated (similar) questions, but opinion queries demand personalized responses for individual information needs. Assuming that each query type can be solved with a single approach or algorithm would be incorrect. Instead, enhancing retrieval system performance requires us to optimize for specific target query types.

- 1) **Retriever-Ranker:** the “retriever” component selects a pool of candidate documents from the index that match to the query, and the “ranker” component assigns a relevance score to each candidate to produce the result.



- For instance, we can employ a BM25 retriever component that generates numerous candidate documents or passages. To broaden the scope of potential answers, we can augment the query with extra keywords using either a dictionary or embeddings, as demonstrated earlier.
- We can subsequently utilize embeddings or a cross-encoder to compute semantic scores between the query and the candidate documents. However, it is important to note that this re-ranking process is constrained by the language model's maximum input sequence length. The outcome is a revised scoring that we use to produce the top-k results for the user.

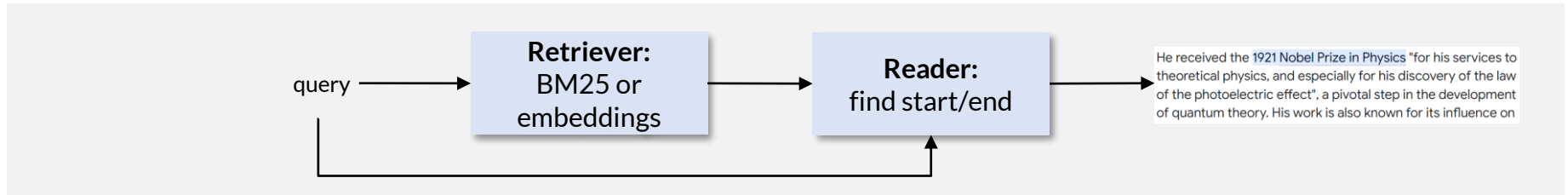
As an alternative, we can replace the retriever with semantic search using embeddings as follows:



- The embeddings for passages are generated during indexing and for the query during retrieval. We then identify passages with embeddings close to the query (e.g., using cosine similarity). Optionally, a cross-encoder can re-rank a larger candidate set provided by the retriever. To facilitate this search, a **high-dimensional vector search index** is essential, as the inverted files method is not applicable for dense vectors. It is important to note that we cannot use the cross-encoder as the retriever in larger scenarios due to computational inference costs as we compare the query with each passage. Thus, reducing the candidate document set with the retriever is necessary to maintain efficient inference through batch comparisons.

This pipeline enhances keyword-based search but cannot address the query types discussed on the previous page. It helps locate relevant documents, but users must extract answers from the documents themselves.

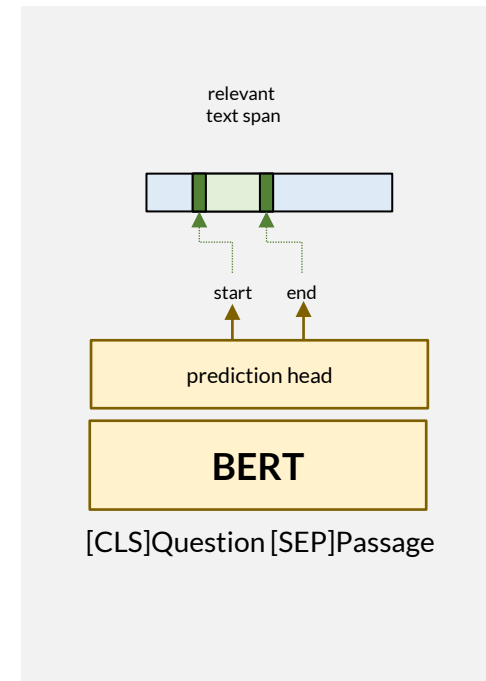
2) **Semantic Search with Reader** is a pipeline that targets factoid queries. It features first a semantic search retriever (we could also use a BM25 for the retriever), but is then identifying the text span in the retrieved passages that best answers the query. We can either highlight the answer in the found passage, or simple use the text span to answer the query:



- The retriever's role is to identify, as before, a broad set of potential passages. The reader, built upon a language model like BERT, employs an added prediction layer to identify the most relevant text segment's start and end. Models like [roberta-base-squad2](#), derived from BERT base, are trained to answer questions, using datasets like Stanford's SQuAD. With the best scoring text span, we can either highlight the text in the corresponding documents, or directly answer the query to the user.
- This pipeline is designed for factoid queries and may lack accuracy for other query types discussed earlier. However, it is highly valuable for scenarios where facts are not publicly available or continuously evolve from news and events, making it challenging for generative models to produce direct answers. For instance, for web queries like “who won the game on the weekend”.
- The code below demonstrates the reader component using the [transformers](#) library. It calculates a score and identifies the start and end positions when comparing the [query](#) with a [passage](#). If passages are lengthy, the reader divides them into smaller sections. Computational expenses may necessitate keeping the candidate list from the retriever relatively short.

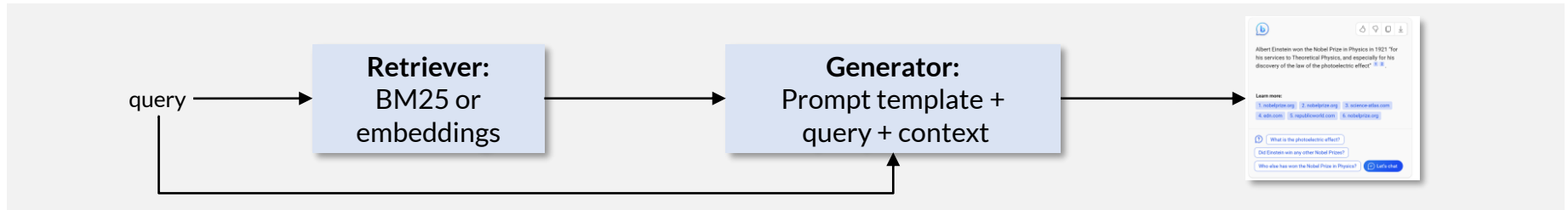
```
from transformers import pipeline

qa_model = pipeline("question-answering",
                    model="deepset/roberta-base-squad2")
answer = qa_model(question=query, context=passage)
answer['score'], answer['answer'], answer['start'], answer['end']
```



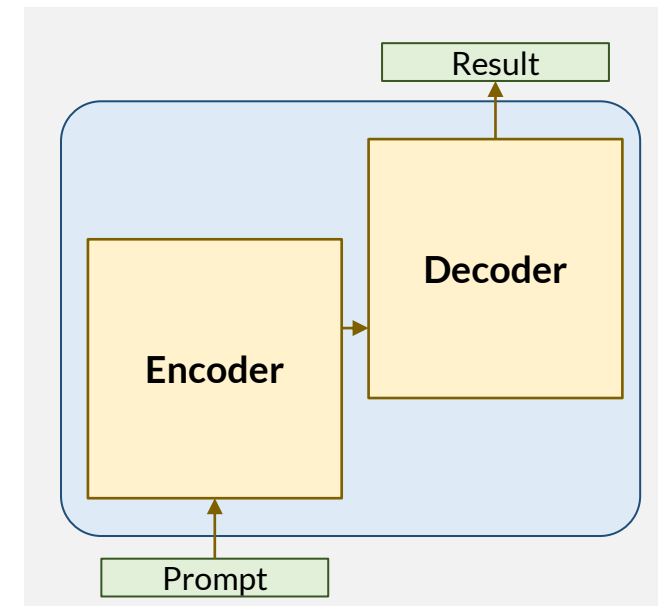


- 3) **Retrieval Augmented Generation (RAG)** takes an extra step by utilizing a retriever to generate relevant passages containing essential information. It combines this contextual information with a query within a prompt template for a generative model to produce the desired outcome:



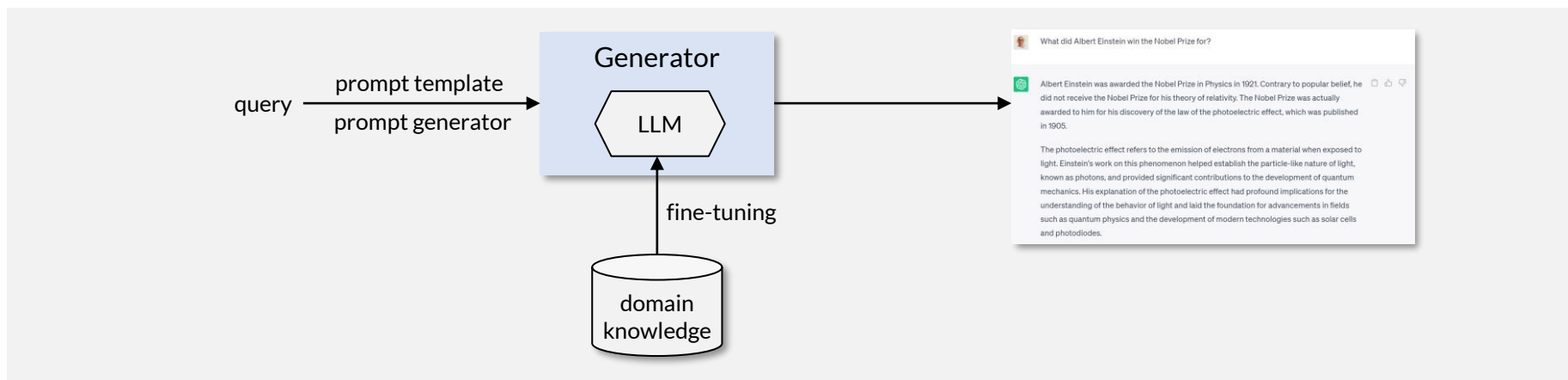
- Similar to the previous approach, the retriever can be either a BM25 or an embeddings-based semantic search model. It provides a few highly relevant passages based on the query. We then utilize, for instance, a **Generative Pre-trained Transformer 3 (GPT-3)** to produce the desired text to answer the user's question. We create a prompt template that instructs the model on what to do, enhancing it with the actual query and incorporating the retrieved passages as context. The generator then generates an answer based on this context rather than relying solely on its own knowledge base. This is particularly valuable when searching for non-public information or addressing queries related to recent news or events not present in the training dataset.
- Below is an example of how to use the [text2text](#) pipeline from the [transformers](#) library. The prompt is a predefined template; you can find examples by searching for "prompt engineering" online. We include the query and passages in the template and call the model to generate text.
- The challenges in this process include: a) creating an effective prompt (including passages for context), b) using a sufficiently large model to accommodate the context, and c) ensuring the model can generate understandable text. Currently, performing this task on your own is challenging, and you require external APIs for this step.

```
from transformers import pipeline
t2t_model = pipeline("text2text-generation",
                     model="google/flan-t5-base")
answer = t2t_model(prompt, max_length=200)
answer[0]['generated_text']
```





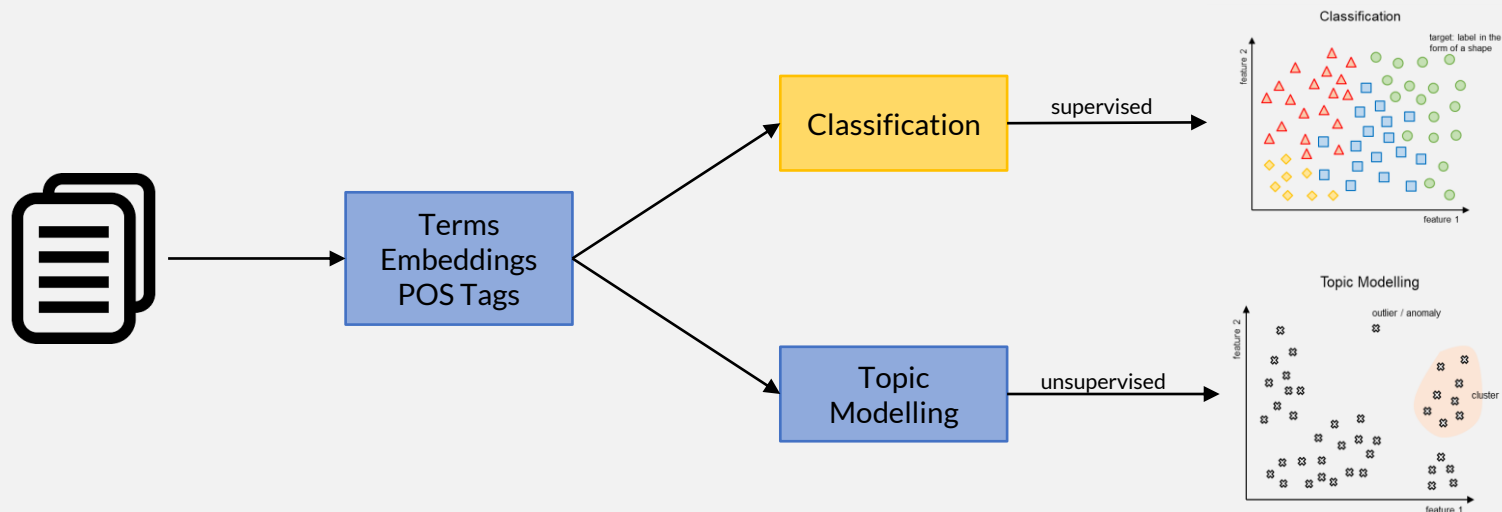
- Although not a retrieval model, we mention here also the standalone generator model as shown below. As on the previous page, we can use a GPT model to produce text, and utilize a prompt template (or generator) to transform the query into a suitable prompt that results in qualitative outcomes:



- The prompt template must align with the query types which we discussed at the beginning of this section. Therefore, a prompt generator is frequently employed to identify the user's question type and the anticipated output format. The prompt also provides extra instructions for the language model, e.g., “write in simple English”, “no more than 200 words”, to generate an answer from its trained knowledge base.
- While versatile for various question types, a significant drawback is the model's inability to access non-public corporate information and its frozen knowledge base lacking information about recent news, events, and changes in general. To address this partly, we can fine-tune an existing model using our own knowledge base. The fine-tuned model can adjust its answers to the current context and keep its knowledge base updated. However, like all generative AI models, continuous fine-tuning can be costly and frequently relies on cloud-based services.
- Generative models can generate false or biased information that was embedded in their training data, and we cannot verify the source of their answers. In some cases, the model can even self-generate wrong answers known as “hallucinations”. Retrieval augmented generation (RAG) is a more reliable approach as it provides references to the user and generates text closely aligned with retrieved passages, unlike fine-tuned generative models.

## 4.8 Text Classification

- In conclusion, this chapter provides examples of text classification methods for assigning documents to predefined classes. It covers language detection, sentiment analysis, text classification, and unsupervised clustering of text documents. We explore various methods to demonstrate these approaches to text classification.
- Topic modeling and clustering differs from text classification. The distinction lies in the approach: classification relies on supervised learning with predefined classes, learning how features align with these classes. In contrast, topic modeling and clustering is unsupervised, aiming to detect clusters or co-occurrences of terms within text documents, and assigning them topic labels (as illustrated in the figure below). LSI also employs unsupervised techniques to learn topics through singular value decomposition and thereby reduces the rank of the document-term matrix. While this process shares similarities with topic modeling methods like Latent Dirichlet Allocation (LDA) and Non-Negative Matrix Factorization (NMF), LSI's primary objective is not to extract and explain topics found within the collection. Instead, it leverages these abstract topics for semantic retrieval.
- While deep learning can handle various tasks, we should also explore cost-effective methods that provide satisfactory solutions. Training complex language models can be resource-intensive, whereas simpler techniques can yield comparable results, particularly when term occurrences is the dominating factor for class assignments.



## 4.8.1 Language Detection

- Language detection is the problem of determining the language in a text or document. This task is rather simple for long documents, but can become quite challenging for short texts or when a large number of languages have to be detected automatically. A related problem: detecting programming languages.
- Let's start with the simple method of detecting languages in longer texts. The most efficient approach is to apply a number of rules, and count stop words to detect the language on simple counts:
  - **Alphabet Diversity:** Each language has unique characters found in only a few related languages. Examples include Latin, Cyrillic, Greek, Arabic, Hebrew, Devanagari, Thai, Tamil, Bengali scripts, as well as Chinese, Hiragana, and Katakana characters. Languages often borrow words from others, leading to a mix of alphabets. To handle this, we can filter out rarely used alphabets.
  - **Character Diversity:** Some languages have special characters within an alphabet that are typical of their linguistic uniqueness. For instance, diacritical marks and accent symbols in Latin-based scripts, or tonal markers in certain Asian languages, add distinctive features to characters. Only a few Latin based languages use ä, ö, and ü.
  - **Stop word Counts:** Evaluating stop word frequencies in text can reveal a language. For instance, languages like English and French often employ frequent stop words, while others, like Mandarin Chinese, rely less on them. Using managed stop word lists allows us to guess a language simply by counting how often the stop words of a language occurs.
  - **Vocabulary Counts:** Examining the unique words or vocabulary in a text can also help identify the language. Different languages have distinct vocabularies, and by comparing word frequencies and diversity, it becomes possible to make language determinations with a degree of accuracy.
- For longer texts, these rules quickly lead to the identification of the language. However, the method does not easily scale to large numbers of languages. The alphabet and character rules are simple, but the stop word lists and vocabularies (most frequent words) require large amount of data to perform language detection.
- For shorter texts or brief phrases, these methods are less effective unless we have comprehensive vocabularies for all languages. In some cases, it may be challenging to identify the correct language, as a single word or short phrase can exist in multiple languages. Even more complex are phrases containing loanwords from other languages, such as IT terms in a German phrase like "mein computer."

- Modern language detectors operate at the sub-word level and incorporate rules like those mentioned earlier, such as Alphabet and Character rules. Additionally, they introduce new rules based on character-based n-grams that are specific to certain languages. The key distinction of these new methods, however, lies in utilizing the frequencies of character-based n-grams for language detection, and this is achieved using a Naïve Bayes learning model.
- Naïve Bayes employs a conditional probability model based on Bayes' theorem.

$$P(C_k|\mathbf{x}) = \frac{P(\mathbf{x}|C_k) \cdot P(C_k)}{P(\mathbf{x})} \quad \text{posterior} = \frac{\text{likelihood} \cdot \text{prior}}{\text{evidence}}$$

In this equation,  $\mathbf{x}$  represents a feature vector, and  $C_k$  is the class or target.  $P(C_k)$  is the **prior**, that is knowledge about the distribution (probability) of classes  $C_k$ .  $P(\mathbf{x}|C_k)$  is the **likelihood** of observing feature  $\mathbf{x}$  for a specific class  $C_k$ , and  $P(\mathbf{x})$  is the overall **evidence** of observing  $\mathbf{x}$ , regardless of class.  $P(C_k|\mathbf{x})$  represents the **posterior** which is the knowledge gained or predicted when observing feature  $\mathbf{x}$ , allowing us to infer its association with class  $C_k$ .

- Consider  $\mathbf{x}$  as a high-dimensional vector, often derived from a vast term space used in documents. Given the high dimensionality and the restricted training data, accurately modeling the probability distribution function in this sparse space is challenging. To simplify, naïve Bayes assumes conditional independence among features, resulting in the following simplification:

$$P(C_k|\mathbf{x}) = P(C_k|x_1, \dots, x_M) = \frac{1}{P(\mathbf{x})} \cdot P(C_k) \cdot \prod_{j=1}^M P(x_j|C_k)$$

Note that  $P(\mathbf{x})$  is a constant across classes  $C_k$  and only scales the probabilities. For our purposes, we do not require its value

- Using the probability model, we choose the most probable hypothesis, that is class  $C_{k^*}$  that maximizes the probability function. This selection principle is commonly referred to as **maximum a posteriori (MAP)**:

$$k^* = \operatorname{argmax}_k P(C_k|\mathbf{x}) = \operatorname{argmax}_k P(C_k) \cdot \prod_{j=1}^M P(x_j|C_k)$$

That is it! The equation describes the decision rule of Naïve Bayes. The only thing left are the estimates for the probabilities on the right hand side

- Now, we need to estimate the probabilities  $P(C_k)$  and  $P(x_j|C_k)$  based on observations from the training set.

- In our language detection scenario, we use character-based n-grams of varying lengths (e.g., n from 1 to 5). We count how often these n-grams appear in the text, resulting in a bag-of-words representation that forms a multinomial distribution. The feature vector  $\mathbf{x}$  represents these counts for a defined vocabulary for each language.
- The priors  $P(C_k)$  depend on the scenario: we can use a maximum likelihood estimator based on observations in the training set. Let  $N_k$  be the number of texts for the language of class  $C_k$ , and  $N$  be the total number of texts:

$$P(C_k) = \frac{N_k}{N} \quad \text{or if } N_k \text{ is not known: } P(C_k) = \frac{1}{K}$$

If we lack knowledge of the language distribution or wish to avoid training bias, we can select a constant prior for all classes, which can then be omitted from subsequent calculations since it only scales posteriors for all classes.

- To estimate the likelihoods  $P(x_j|C_k)$  from texts in a language represented by class  $C_k$ , we count the n-gram occurrences in the training data for that language (**multinomial distribution**). For each language, we establish first an appropriate vocabulary, using methods similar to word-pieces or BPE, to control vocabulary size. We prioritize the most frequent n-grams since they have the most influence on the posterior and impact language determination the most. Let  $n_{k,j}$  denote the total occurrences of n-gram  $t_j$  in all training texts for the language represented by class  $C_k$ :

$$p_{k,j} = \frac{n_{k,j}}{\sum_l n_{k,l}} \quad \text{or smoothed: } p_{k,j} = \frac{n_{k,j} + 1}{\sum_l n_{k,l} + M}$$

As we choose the vocabulary tailored to the target language and exclude infrequent or absent n-grams from the test set, we do not require the “+1” smoothing on the right-hand side. However, in other text classification tasks, smoothing prevents  $p_{k,j}$  from reaching 0 for rare tokens during predictions (which leads to a posterior of value 0).

- Finally, we can predict the language based on posteriors. Instead of multiplying probabilities as shown on the previous page, we rather use sums over log-probabilities:

$$k^* = \operatorname{argmax}_k P(C_k|\mathbf{x}) = \operatorname{argmax}_k \left( \log P(C_k) + \sum_{x_j > 0} x_j \log p_{k,j} \right)$$

We can obtain scores with a softmax classifier over the target languages, and select the language with highest score. If several languages reach a low threshold values, we can return multiple predictions.

- **Examples:** The [lingua-language-detector](#) is a highly efficient language detector with over 99% accuracy for more than 70 languages. Let's explore its functionality through examples.

```
from lingua import Language, LanguageDetectorBuilder
detector = LanguageDetectorBuilder.from_all_languages().build()
detector.detect_language_of("This is an example sentence")      # → Language.ENGLISH
detector.detect_language_of("Je suis un exemple de phrase")     # → Language.FRENCH
detector.detect_language_of("นี่คือข้อความที่ทดสอบ")           # → Language.THAI
```

- We can also inquire about the likelihood of a phrase belonging to a particular set of languages:

```
languages = [Language.ENGLISH, Language.FRENCH, Language.ITALIAN]
detector = LanguageDetectorBuilder.from_languages(*languages).build()
detector.compute_language_confidence_values("Je suis à New York")
↳ FRENCH: 0.45 ENGLISH: 0.37 ITALIAN: 0.18
```

- The detector also are able to predict the languages out of fragments:

```
confidence_values = detector.compute_language_confidence_values("hau mei")
↳ GERMAN: 0.82 ENGLISH: 0.10 ITALIAN: 0.07
```

- This also demonstrates that the detector operates at sub-word levels. The 3-grams “hau” and “mei” are more common in German texts than in English and Italian, resulting in higher confidence scores.
- Another Python library is [langdetect](#), which is also a rule and n-grams based language detector for 55 languages. It provides ISO-codes for the detected languages:

```
from langdetect import detect, detect_langs
detect("This is an example sentence")      # → en
detect("je suis un exemple de phrase")     # → fr
detect("Este es un ejemplo de frase")     # → es
detect("Dies ist ein Beispieltext")        # → de
detect("Questo è un esempio di frase")    # → it
detect("นี่คือข้อความที่ทดสอบ")           # → th
detect_langs("Je suis à New York")         # → [fr:0.86, en: 0.14]
detect_langs("hau mei")                   # → [cy: 1.00]
```

## 4.8.2 Sentiment Analysis

- Sentiment analysis deciphers human language to understand emotions and opinions. It's widely used to assess customer sentiment from reviews, social media, and support cases, aiding data-driven decisions for product improvement and customer satisfaction. It also can help to filter and moderate user-generated content to safeguard brand reputation and enforce community guidelines.
- In sentiment analysis, a fundamental task is to classify text polarity—identifying if it is positive, negative, or neutral. Advanced tasks can recognize emotions like anger, fear, disgust, joy, and surprise. Here, we focus on basic sentiment prediction with positive, negative, and neutral categories. Let's start with a look at some example, and the challenges machine learning models may encompass:
  - Many statements are straightforward and sentiment is often driven by a few key words:

“I like this product”	→ positive
“I was going to the town”	→ neutral
“The food was really bad”	→ negative
  - However, we can express ourselves in many, sometimes confusing ways that are difficult to analyze:

“I can't say I liked it”	→ negation handling
“Drinking wine is not my thing”	→ negative or neutral?
“What a fine artist you've become!”	→ potentially sarcastic
“I haven't ever owed anything to anyone”	→ lots of negation, but actually positive
- We consider 3 different approaches:
  - Naïve Bayes with the example of sentiment analysis in Twitter (now called X)
  - TextCNN, a convolutional network on embeddings to predict classes
  - Transformer based classification models

- **Naïve Bayes** is popular for its simplicity, speed, and accuracy. We used it before for language detection with a multinomial distribution, considering term presence and counts. In Twitter sentiment analysis, short texts mean terms usually occur only once, except for stop words. We use a set-of-word representation and assume a multivariate Bernoulli distribution for likelihood estimation. This examples uses two classes: positive and negative
  - The priors  $P(C_k)$  measure how likely a class is compared to the other one. We can use a maximum likelihood estimator based on observations in the training set. Let  $N_k$  be the number tweets for class  $C_k$ , and  $N$  be the total number of texts (with  $k$  = 'positive' or 'negative'):

$$P(C_k) = \frac{N_k}{N} \quad \text{or if } N_k \text{ is not known: } P(C_k) = \frac{1}{K}$$

If we do not know the sentiment distribution or wish to avoid training bias, we can select a constant prior for all classes, which can then be omitted from subsequent calculations since it only scales posteriors for all classes.

- Assuming a multivariate Bernoulli distribution for the set-of-word representations, we can estimate the likelihoods  $P(x_j|C_k)$  as follows. Let  $N_k(x_j = 1)$  denote the number of tweets from  $C_k$  that contain a term  $t_j$ :

$$p_{k,j} = \frac{N_k(x_j = 1)}{N_k} \quad \text{or smoothed: } p_{k,j} = \frac{\min(N_k - 1, \max(1, N_k(x_j = 1)))}{N_k}$$

We can use either smoothing to prevent  $p_{k,j} = 0$  if a term  $t_j$  does not occur in the tweets of class  $C_k$ , or we simply ignore terms that were not present in the training data of class  $C_k$  during predictions.

- Finally, we can predict the sentiment ('positive' or 'negative' class) based on posteriors. Instead of multiplying probabilities, we again use sums over log-probabilities (and ignore terms with  $p_{k,j} = 0$ ):

$$k^* = \operatorname{argmax}_k P(C_k|\mathbf{x}) = \operatorname{argmax}_k \left( \log P(C_k) + \sum_{j=1}^M (x_j \log p_{k,j} + (1 - x_j) \log(1 - p_{k,j})) \right)$$

- Instead of using the entire vocabulary, we can reduce features by selecting the most informative terms present in the document ( $x_j = 1$ ). In this case, the formula simplifies to the sum of  $\log p_{k,j}$  for the most informative terms in the document.



The code on the right hand side shows the Twitter sentiment implementation:

- 1) We utilize the Twitter samples data from the `nltk` corpus, consisting of 5,000 positive and 5,000 negative labeled tweets. These tweets are read and labeled accordingly. Additionally, we create a list of stop words, create a Snowball stemmer, and utilize a tweet tokenizer that recognizes Twitter-specific tokens like hashtags, user tags, and emoticons.
- 2) In the process of cleaning the tweets, we eliminate HTTP links and user tags, as they are not relevant for sentiment analysis. We employ the Twitter tokenizer and remove single-letter tokens, numbers, and stop words. However, we retain emoticons like “:-)” since they can carry sentiment information.
- 3) We divide the training and test data into an 80:20 ratio while ensuring an even distribution of positive and negative tweets in both the training and test subsets through stratification.
- 4) We obtain a classifier from the `nltk` Naïve Bayes training and then assess its training accuracy (0.999) and test accuracy (0.995). The Naïve Bayes classifier makes only 16 incorrect predictions out of 10,000 samples. Some of the most informative features for this classifier are “:-)” and “:(”, among others.

In this scenario, Naïve Bayes is not only highly accurate but also remarkably fast, with training and classification taking less than a second. None of the deep learning methods can compete with this speed.

```
# 1) get started with data and settings
tweets = [(t,"pos") for t in twitter_samples.strings("pos...")] + \
          [(t,"neg") for t in twitter_samples.strings("neg...")]
stopwords = nltk.corpus.stopwords.words('english')
stemmer = nltk.stem.SnowballStemmer('english')
tokenizer = nltk.tokenize.casual.TweetTokenizer()

# 2) cleaning all the tweets --> set of words model
def set_of_words(text):
    text = re.sub(HTTP_REGEX, '', text)
    text = re.sub("@[A-Za-z0-9_]+", "", text)
    tokens = tokenizer.tokenize(text)
    tokens = [stemmer.stem(t) for t in tokens
              if len(t)>1 and
              not t.isnumeric() and
              t not in stopwords]

    return {t:1 for t in tokens}

data = [(set_of_words(text), label) for text, label in tweets]

# 3) split training and test (stratify pos/neg samples)
pos_data = [x for x in data if x[1] == 'pos']
neg_data = [x for x in data if x[1] == 'neg']
pos_split = 80 * len(pos_data) // 100
neg_split = 80 * len(neg_data) // 100

train_data = pos_data[:pos_split] + neg_data[:neg_split]
test_data = pos_data[pos_split:] + neg_data[neg_split:]

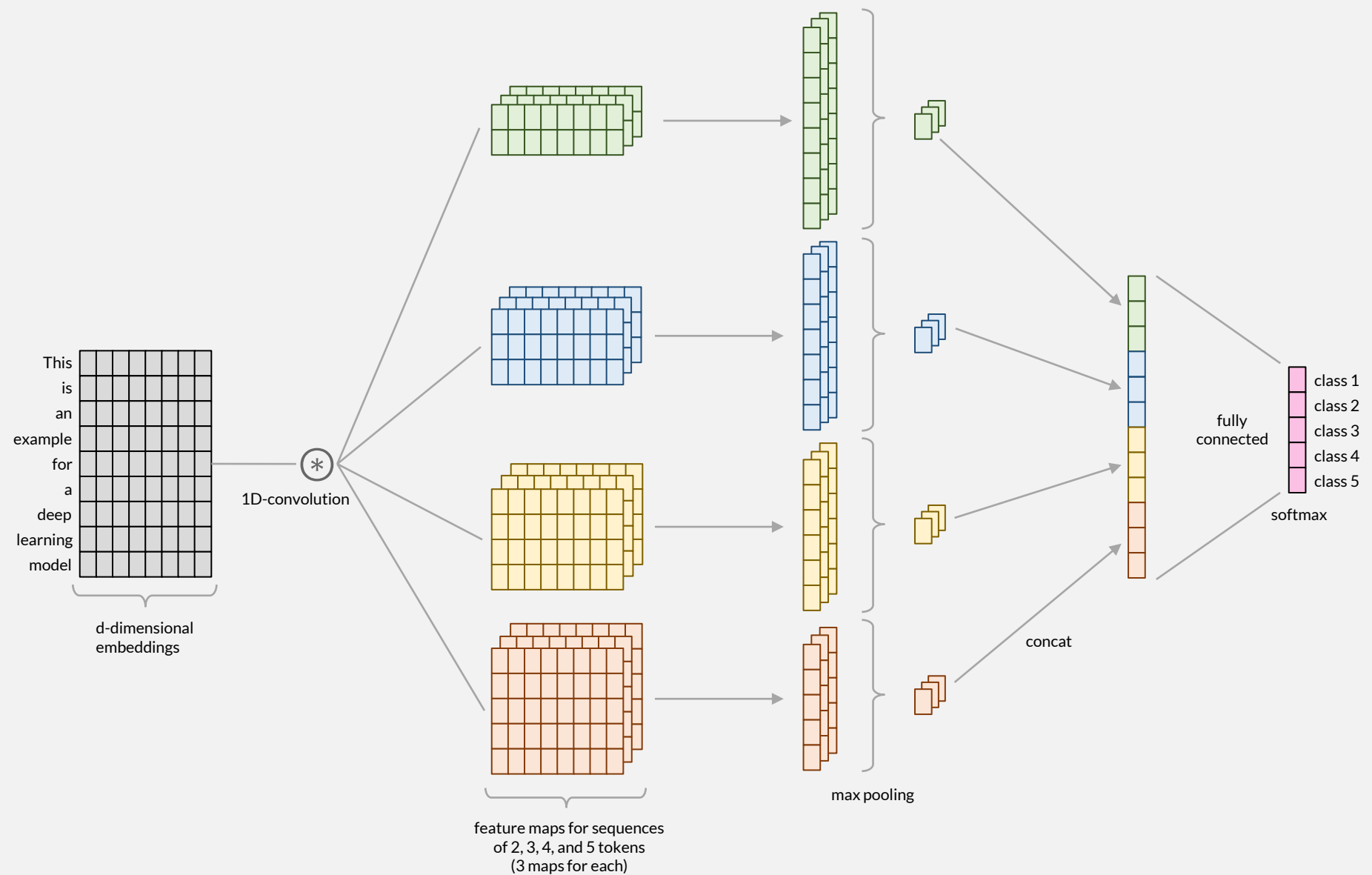
# 4) classify with Naive Bayes (bernoulli)
classifier = nltk.NaiveBayesClassifier.train(train_data)

print(nltk.classify.accuracy(classifier, train_data))
print(nltk.classify.accuracy(classifier, test_data))
print(classifier.show_most_informative_features(10))

false_predictions = [t for t in train_data
                     if classifier.classify(t[0]) != t[1]]
false_predictions += [t for t in test_data
                     if classifier.classify(t[0]) != t[1]]
```

## 4.8.3 Text Classification with Deep Learning

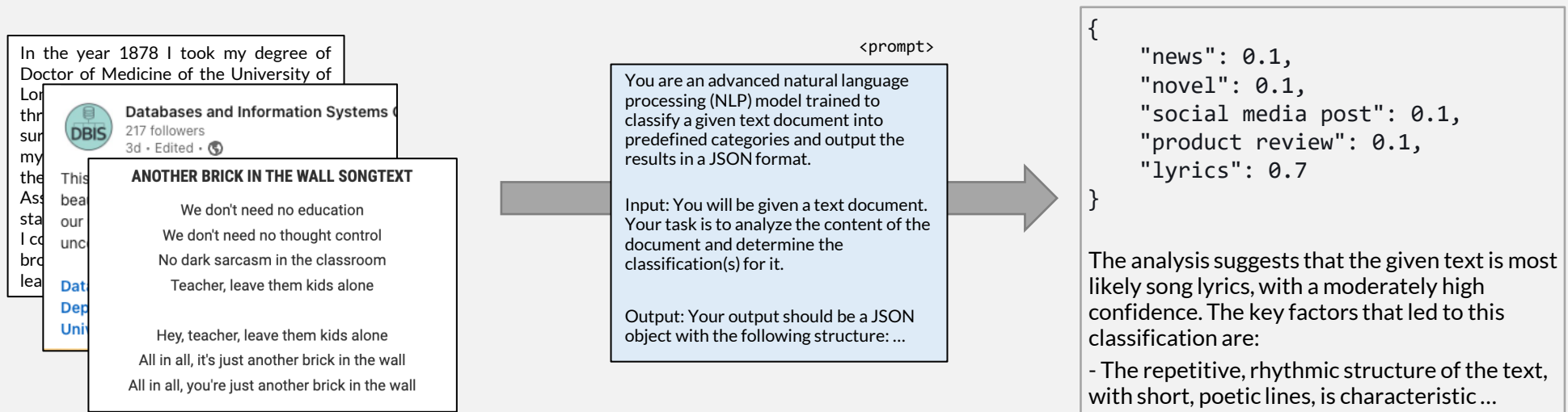
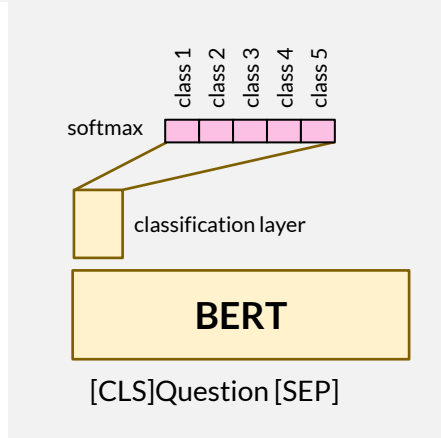
- Sentiment analysis falls under text classification, and Naïve Bayes methods can be expanded to handle broader classification tasks. In this section, we explore the application of deep learning techniques to tackle more complex classification challenges. It is essential to begin by discussing the fundamental differences beforehand:
  - Naïve Bayes is a straightforward, yet highly effective and efficient method capable of real-time classification with low resource demands. Training and re-training are quick and straightforward, and model parameters occupy minimal storage. Storage consumption and performance can be further enhanced by selecting a subset of the most informative features. Consequently, Naïve Bayes, along with other simple classifiers like XGBoost or SVM, serves as an excellent initial choice. More advanced methods should only be considered when they can substantiate increased resource requirements with significantly higher accuracy.
  - Consider the sentiment analysis results from previous sections. Naïve Bayes achieves high accuracy, scoring 0.995 with only 16 false predictions out of 10,000 samples. While theoretically, we could opt for a deep learning approach like a transformer-based sentiment analyzer, such a model would not classify tweets as quickly as Naïve Bayes. In fact, a basic RoBERTa model takes seconds to minutes for classifying 10,000 samples (dependent on available hardware). Even if it achieved perfect accuracy (100%), the enhanced quality would not justify the significantly greater resource requirements.
  - Simpler models like Naïve Bayes rely on the independence assumption. In many complex scenarios, this assumption does not hold, leading to a rapid decline in the performance of simple models. While we can employ lemmatization techniques to enhance quality, these models cannot capture dependencies. On the other hand, deep learning models can adapt to complex scenarios and are versatile enough to handle various classification tasks without requiring substantial architectural changes.
- In this section, we examine the architecture of TextCNN and transformer-based classification architectures which offer distinct approaches to text classification. TextCNN utilizes convolutional layers to extract features from text embeddings, making it effective for capturing local patterns in data. In contrast, transformers excel in handling long-range dependencies through self-attention mechanisms, making them ideal for tasks requiring a broader context understanding. While textCNN is computationally efficient and interpretable, transformers are highly flexible and excel in tasks demanding nuanced contextual understanding. The choice between the two depends on the specific requirements of the classification problem, with textCNN being suitable for simpler tasks, and transformers shining in more complex, context-sensitive scenarios.



- **TextCNN Architecture**

- Tokens are converted into  $d$ -dimensional embedding vectors and fed into the network. Unlike the transformers architecture, the sequence length is treated as an input dimension, not an architectural parameter. This allows us to handle sequences of arbitrary length and apply convolutions to both short and long sentences without the need for padding. We can select any method and dimensionality for the embeddings.
- We can utilize a set of feature maps to perform 1D convolutions on the sequence of embeddings. A feature map consists of weights of size  $n \times d \times m$ , where  $n$  represents the number of consecutive embeddings in the sequence window,  $d$  is the embedding dimensionality, and  $m$  denotes the number of output values from the convolution. The feature map traverses the sequence, applying 1D convolution to the next  $n$  embeddings, adding a bias, and applying an activation function for an output value. With  $m$  feature maps, we compute  $m$  output values for each position. With a sequence length of  $s$ , this results in  $s - n + 1$  values for each of the  $m$  feature maps.
- As the input sequence can vary in length, the next step employs max pooling to condense the  $s - n + 1$  values from each feature map into a single value. These resulting values are then concatenated into a vector of fixed length. In this example, we utilized feature maps of dimensions  $2 \times d \times 3$ ,  $3 \times d \times 3$ ,  $4 \times d \times 3$ , and  $5 \times d \times 3$ , resulting in a concatenated feature vector of dimensions  $4 * 3 = 12$  as the output of the convolutional layer.
- A fully connected network translates this 12-dimensional vector into  $k$  logits (in our example,  $k = 5$ ) and then applies a softmax function to predict the text's associated class.

- **Transformer-based classification** leverages pre-trained transformer models like BERT or GPT as the core, extending them with extra layers to make class predictions. Typically, in transformer-based classification, we initiate a sequence with a model-specific token (e.g., [CLS] for BERT) and utilize the corresponding encoder output vector. This vector is then passed through a deep classification layer, which computes the logits for the  $k$  classes related to the task. A softmax function is applied to determine the class to which the input text belongs.
- There are two ways to train the model for a given classification task:
  - The base transformer model (also called foundation model) is frozen and we only train the parameters of the additional classification layers. The foundation model can be shared across various classification tasks.
  - Both the base transformer model and the classification layer are trained together. This leads to a fine-tuned foundation model optimized for the classification task, but requires separate models for each classification task.
- **Modern large language models** can easily extract classification information from text documents through prompt engineering. By providing a text document, the prompt asks the language model to extract the desired classes in a specific output format, like JSON.



## 4.8.4 Text Clustering

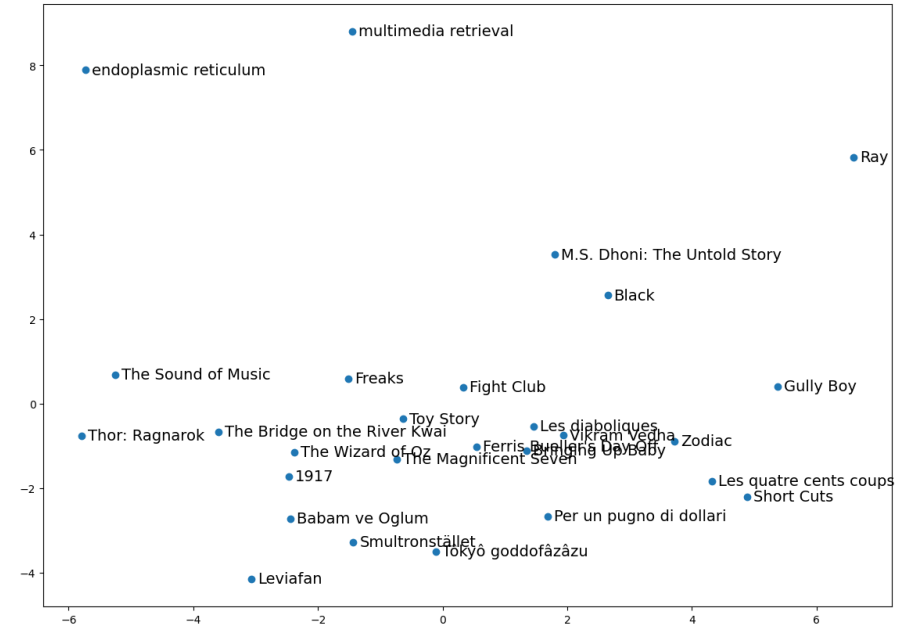
- Embeddings can assist in discovering clusters of related documents or sentences. These clusters can help in recognizing document groups or outliers. A basic method for visual clustering is to map high-dimensional embeddings into a 2D representation, often achieved using techniques like PCA.

```
from sentence_transformers import SentenceTransformer
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# 1) encode all movies with a sentence transformer
model_name = 'nq-distilbert-base-v1'
strf = SentenceTransformer(model_name)
vectors = strf.encode([m['title'] + m['summary'] for m in movies])

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

# 3) create a scatter plot of the projection
plt.figure(figsize=(14,10))
plt.scatter(result[:, 0], result[:, 1])
for i, r in enumerate(result):
    plt.annotate(movies[i]['title'], xy=(r[0]+0.1, r[1]-0.1))
plt.show()
```



- Another method is employing k-means clustering to group similar documents together. To detect outliers, we calculate the minimum distance to the established clusters. If this distance surpasses a defined threshold, we can classify the document as an outlier.
- PCA provides a visual representation that can quickly reveal the structure of collections in 2D space, but it may not be as precise for fine-grained clustering. On the other hand, k-means clustering is more systematic and precise in grouping similar documents, making it a robust choice for understanding the relationships between documents. When determining outliers, it calculates distances to clusters, allowing for a clear definition of what is an outlier.

## 4.9 Literature and Links

### Online Books

- S. Bird, E. Klein, and E. Loper. **Natural Language Processing with Python**. O'Reilly Media, 2009. Free online version: <http://www.nltk.org/book/>
- A. Zhang, Z. C. Lipton, Mu Li, A J. Smola. **Dive into Deep Learning**. Cambridge University Press, to be released. Free online version: <https://d2l.ai>
- I. Goodfellow, Y. Bengio, and A. Courville. **Deep Learning**. MIT Press, 2016, <http://www.deeplearningbook.org>

### Papers

- Susan T. Dumais (2005). **Latent Semantic Analysis**. Annual Review of Information Science and Technology. <https://doi.org/10.1002%2Faris.1440380105>
- 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>
- T. Kiss, and J. Strunk: **Unsupervised Multilingual Sentence Boundary Detection**. Computational Linguistics, 2006. <https://aclanthology.org/J06-4003.pdf>
- C. Paice. **Another Stemmer**. ACM SIGIR Forum 24.3 (1990). <https://dl.acm.org/doi/pdf/10.1145/101306.101310>
- P. Koehn, K. Knight. **Empirical Methods for Compound Splitting**. 10th Conference of the European Chapter of the Association for Computational Linguistics, 2003. <https://arxiv.org/ftp/cs/papers/0302/0302032.pdf>
- 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>
- 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>

## 4.9.1 Literature and Links

### Implementations

- Models: Huggingface, the AI community building the future. <https://huggingface.co>
- Data & Competitions: Kaggle: Your Machine Learning and Data Science Community. <https://www.kaggle.com>
- Transformers: State-of-the-art Machine Learning. <https://github.com/huggingface/transformers>,  
<https://huggingface.co/docs/transformers/index>
- LangChain: Framework for developing applications powered by language models: <https://python.langchain.com/>
- NLTK: Natural Language Toolkit. <https://www.nltk.org/>
- spaCy: Industrial-Strength Natural Language Processing. <https://spacy.io/>
- Apache OpenNLP: Machine learning based toolkit. <https://opennlp.apache.org/>
- WordNet: A Lexical Database for English. <https://wordnet.princeton.edu/>
- Snowball: Stemming Algorithms. <https://snowballstem.org/>
- GloVe: Global Vectors for Word Representation. <https://github.com/stanfordnlp/GloVe>