Chapter 1. An Introduction to Large Language Models

Objective of the Chapter

- What is Language AI?
- What are large language models?
- What are the common use cases and applications of large language models?
- How can we use large language models ourselves?

What Is Language AI?

- [Artificial intelligence is] the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.
- Language AI refers to a subfield of AI that focuses on developing technologies capable of understanding, processing, and generating human language. The term Language AI can often be used interchangeably with natural language processing (NLP) with the continued success of machine learning methods in tackling language processing problems.

History of Language AI

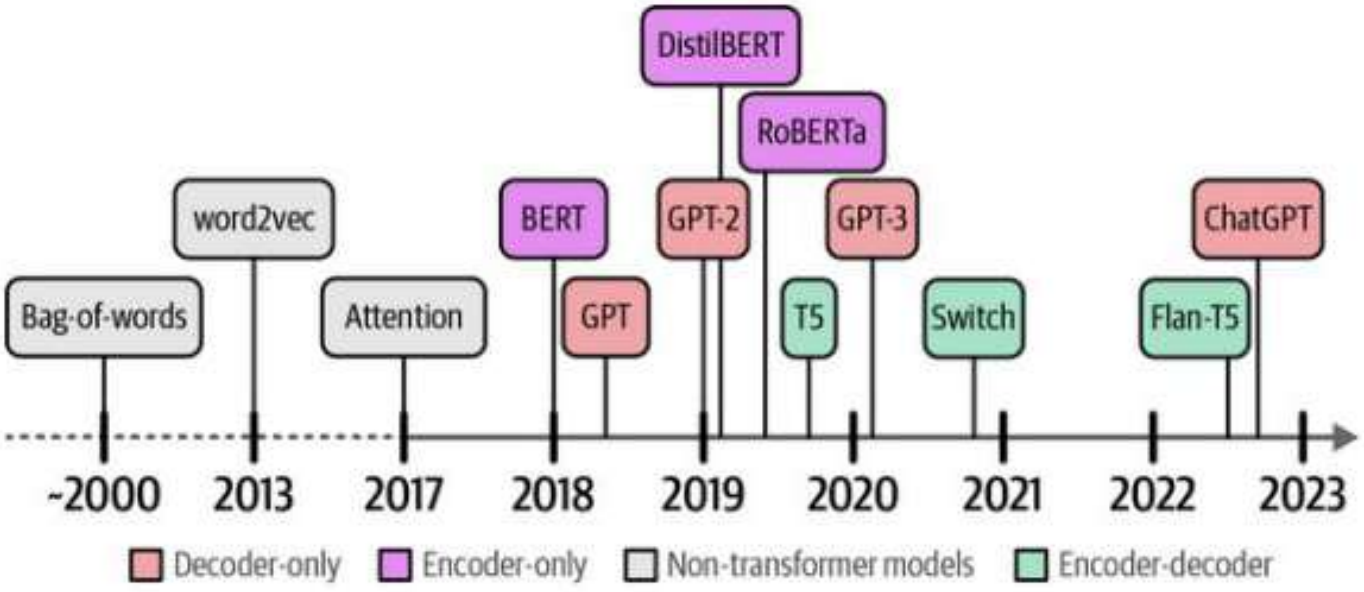


Figure 1-1. A peek into the history of Language AI.

Text is unstructured in nature and loses its meaning when represented by zeros and ones (individual characters). As a result, throughout the history of Language AI, there has been a large focus on representing language in a structured manner so that it can more easily be used by computers.

Representing Language as a Bag-of-Words

- Our history of Language AI starts with a technique called bag-of-words, a method for representing unstructured text. It was first mentioned aroundthe 1950s but became popular around the 2000s.
- Bag-of-words works as follows: let's assume that we have two sentences for which we want to create numerical representations. The first step of the bag-of-words model is tokenization, the process of splitting up the sentences into individual words or subwords (tokens),

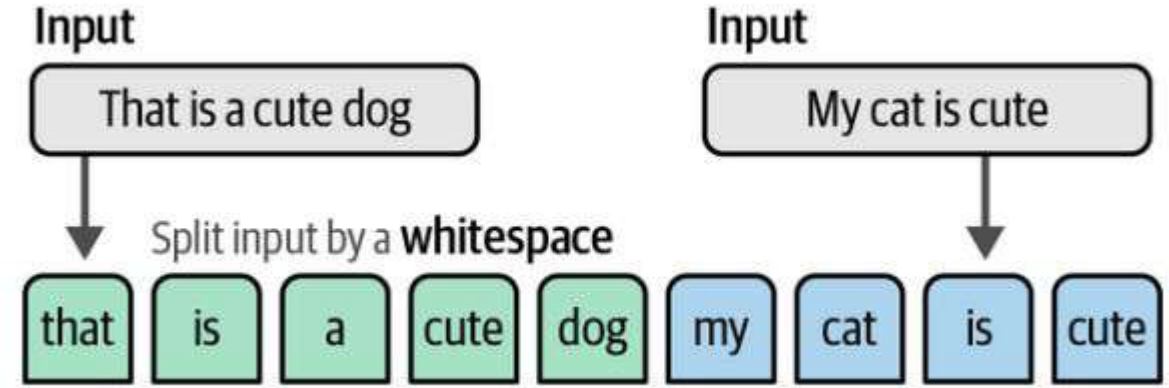


Figure 1-3. Each sentence is split into words (tokens) by splitting on a whitespace.

Creation of Vocabulary

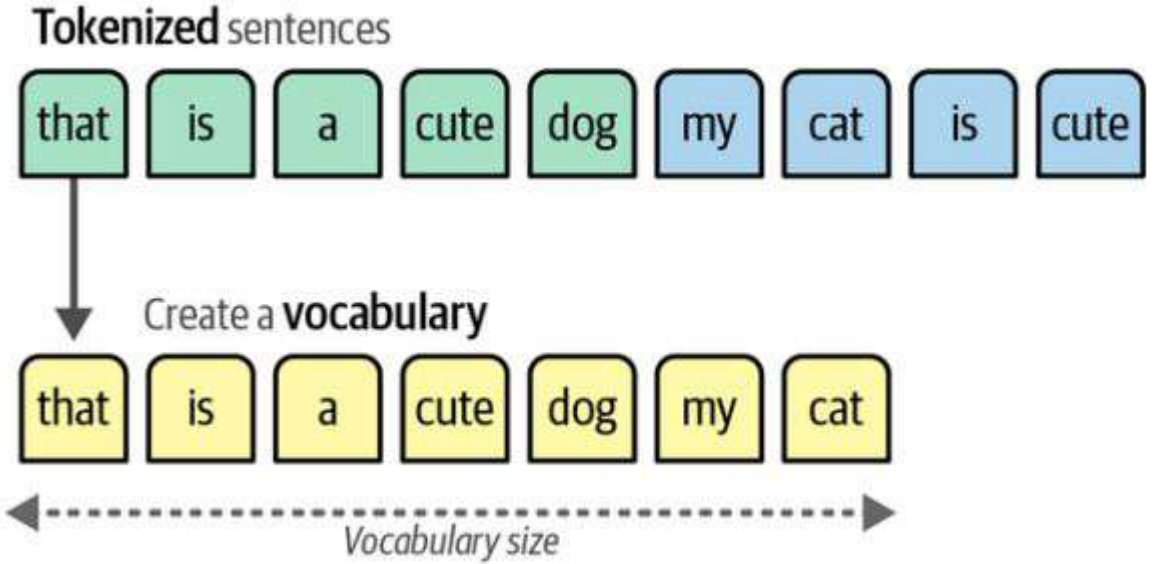


Figure 1-4. A vocabulary is created by retaining all unique words across both sentences.

As illustrated in Figure 1-4, after tokenization, we combine all unique words from each sentence to create a vocabulary that we can use to represent the sentences. Using our vocabulary, we simply count how often a word in each sentence appears, quite literally creating a bag of words.

Vector Representaion of Bag-of-Words

The Bag of Words model represents a sentence or document as a **vector of token frequencies**, ignoring grammar and word order. Each dimension corresponds to a word in the vocabulary.

Properties of BoW Vectors

- **Fixed length**: determined by vocabulary size
- **Sparse**: many zeros for unseen words
- **Orderless**: no information about word sequence
- **Simple**: useful for baseline models and feature extraction

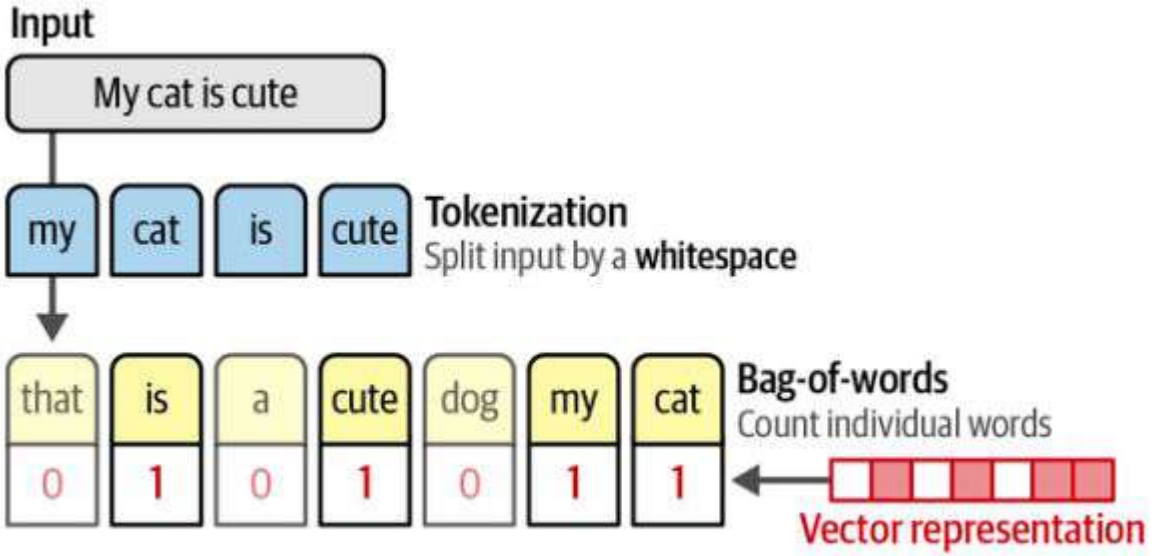


Figure 1-5. A bag-of-words is created by counting individual words. These values are referred to as vector representations.

Advantages and Disadvantages of BOW

Advantages: The most common method for tokenization is by splitting on a whitespace to create individual words. However, this has its disadvantages as some languages, like Mandarin, do not have whitespaces around individual words.

Disadvantages: The most common method for tokenization is by splitting on a whitespace to create individual words. However, this has its disadvantages as some languages, like Mandarin, do not have whitespaces around individual words.

Dense Vector Representation

- Bag-of-words, although an elegant approach, has a flaw. It considers language to be nothing more than an almost literal bag of words and ignores the semantic nature, or meaning, of text.
- Released in 2013, word2vec was one of the first successful attempts at capturing the meaning of text in embeddings. Embeddings are vector representations of data that attempt to capture its meaning.
- To generate these semantic representations, word2vec leverages neural networks. These networks consist of interconnected layers of nodes that process information.

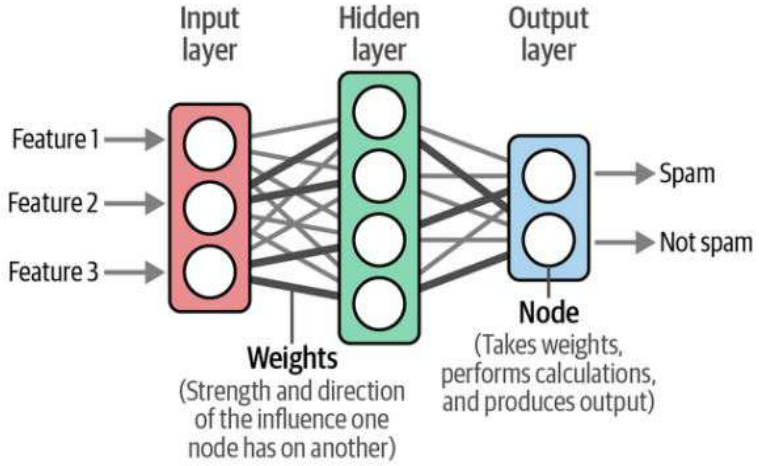


Figure 1-6. A neural network consists of interconnected layers of nodes where each connection is a linear equation.

How a Neural Network Works

Word2Vec Embeddings

- Using these neural networks, word2vec generates word embeddings by looking at which other words they tend to appear next to in a given sentence.

- If the two words tend to have the same neighbors, their embeddings will be closer to one another and vice versa.
- we take pairs of words from the training data and a model attempts to predict whether or not they are likely to be neighbors in a sentence.

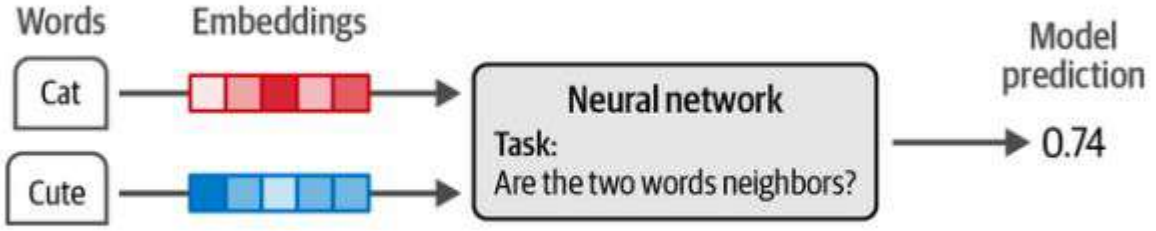


Figure 1-7. A neural network is trained to predict if two words are neighbors. During this process, the embeddings are updated to be in line with the ground truth.

How the training is done in Word2Vec

To illustrate this phenomenon, let's somewhat oversimplify and imagine we have embeddings of several words, namely "apple" and "baby." Embeddings attempt to capture meaning by representing the properties of words. For instance, the word "baby" might score high on the properties "newborn" and "human" while the word "apple" scores low on these properties.

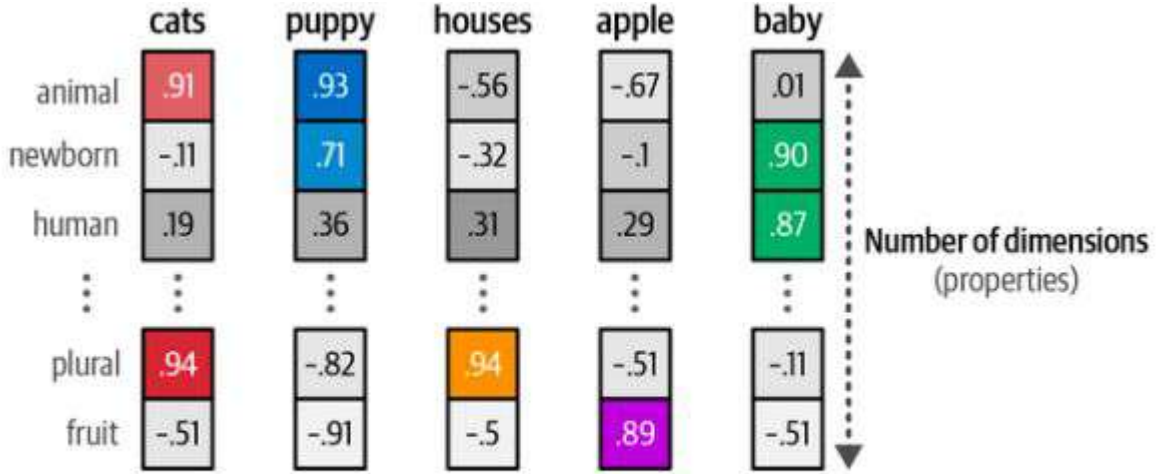


Figure 1-8. The values of embeddings represent properties that are used to represent words. We may oversimplify by imagining that dimensions represent concepts (which they don't), but it helps express the idea.

Embeddings provide Semantic Similarity

Embeddings are tremendously helpful as they allow us to measure the semantic similarity between two words. Using various distance metrics, we can judge how close one word is to another.

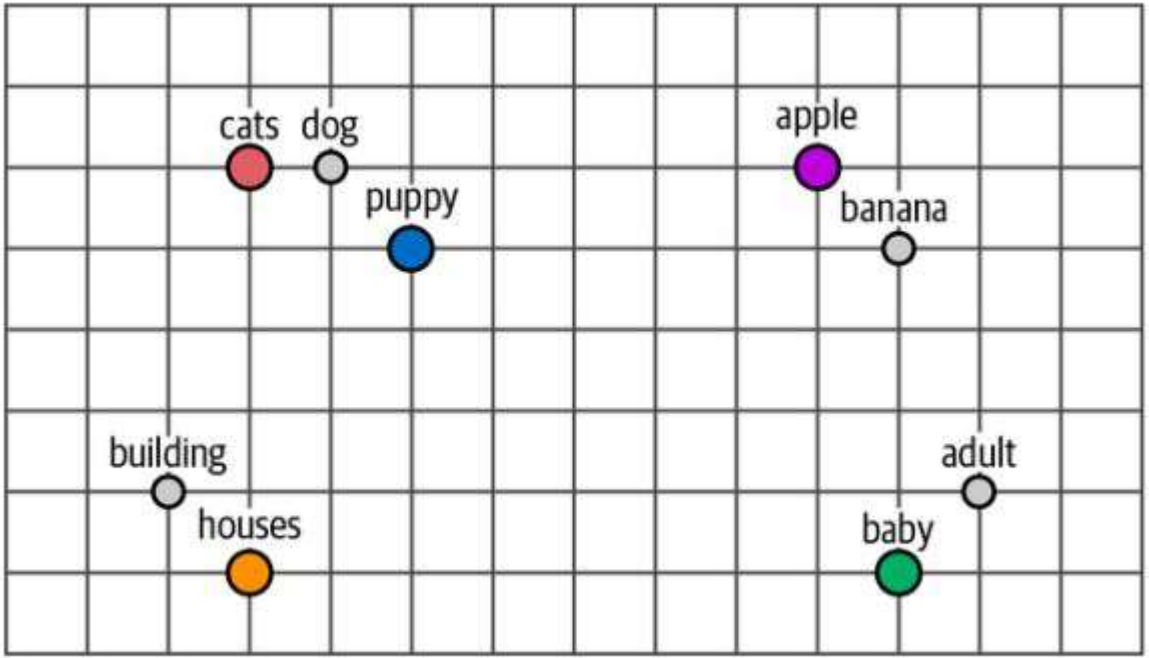


Figure 1-9. Embeddings of words that are similar will be close to each other in dimensional space.

RNN - To capture semantic meaning

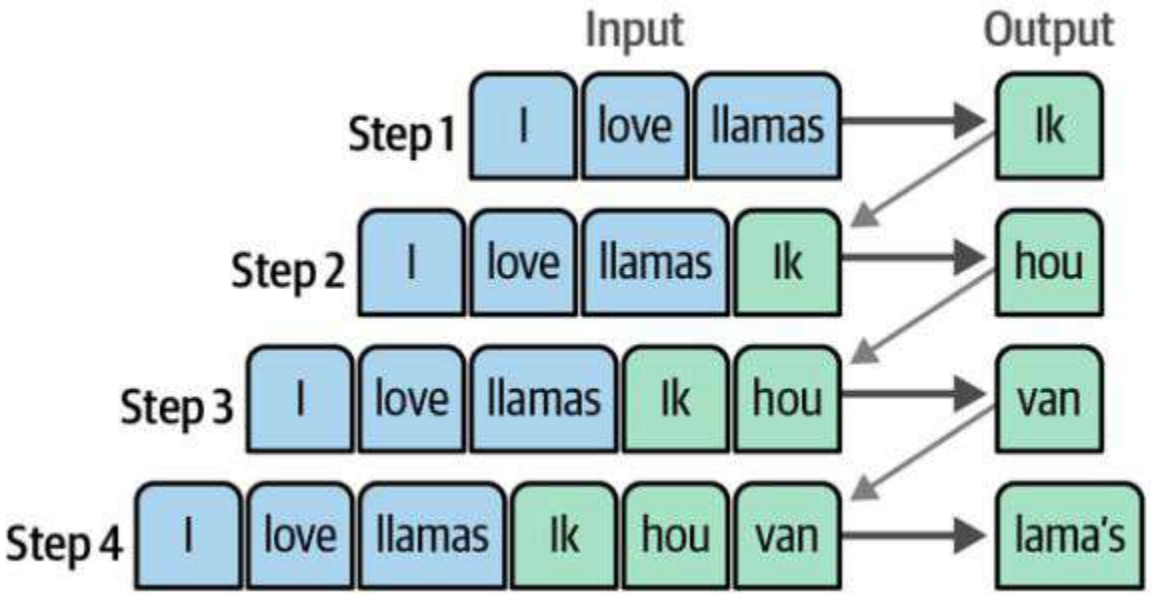


Figure 1-12. Each previous output token is used as input to generate the next token.

The word “bank” can refer to both a financial bank as well as the bank of a river. Its meaning, and therefore its embeddings, should change depending on the context.

A step in encoding this text was achieved through recurrent neural networks
(RNNs). These are variants of neural networks that can model sequences as an additional input.

Each step in this architecture is autoregressive. When generating the next
word, this architecture needs to consume all previously generated words,

Encoding and Decoding

The encoder takes a sequence of tokens (e.g., `"The fire burns bright"`) and transforms it into a series of **contextualized embeddings**.

These embeddings capture both the meaning of each token and its relationship to others in the sequence.

This is done using **self-attention** and **feed-forward layers**.

In Encoder-Decoder Models (e.g., BERT2GPT, T5):

- The decoder receives the encoder's output and generates a new sequence (e.g., a translation, summary, or answer).
- It uses **self-attention** to look at previously generated tokens and **encoder-decoder attention** to refer back to the input.

In Decoder-Only Models (e.g., GPT):

- There is no separate encoder. The input is fed directly into the decoder.
- The decoder uses **causal self-attention** to generate the next token based only on previous ones.

RNN + Attention Mechanism

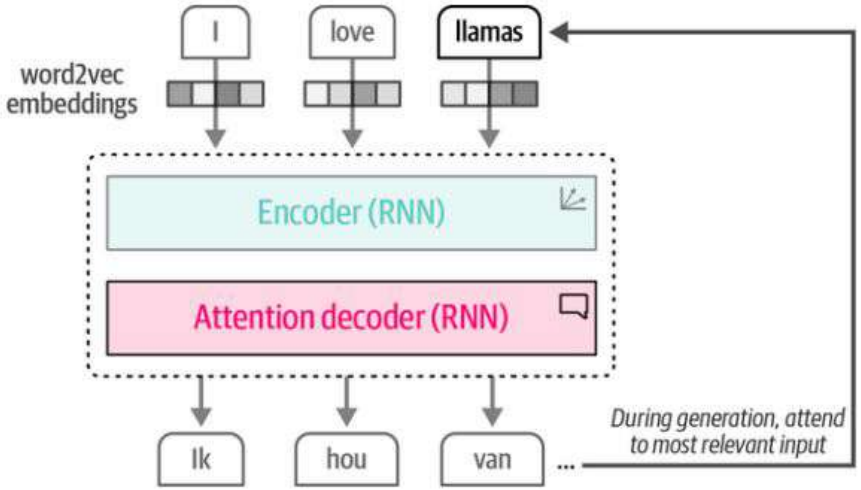


Figure 1-15. After generating the words "Ik," "hou," and "van," the attention mechanism of the decoder enables it to focus on the word "llamas" before it generates the Dutch translation ("lama's")

By adding these attention mechanisms to the decoder step, the RNN can generate signals for each input word in the sequence related to the potential output. Instead of passing only a context embedding to the decoder, the hidden states of all input words are passed.

Attention allows a model to focus on parts of the input sequence that are relevant to one another ("attend" to each other) and amplify their signal, as shown in Figure 1-14. Attention selectively determines which words are most important in a given sentence.

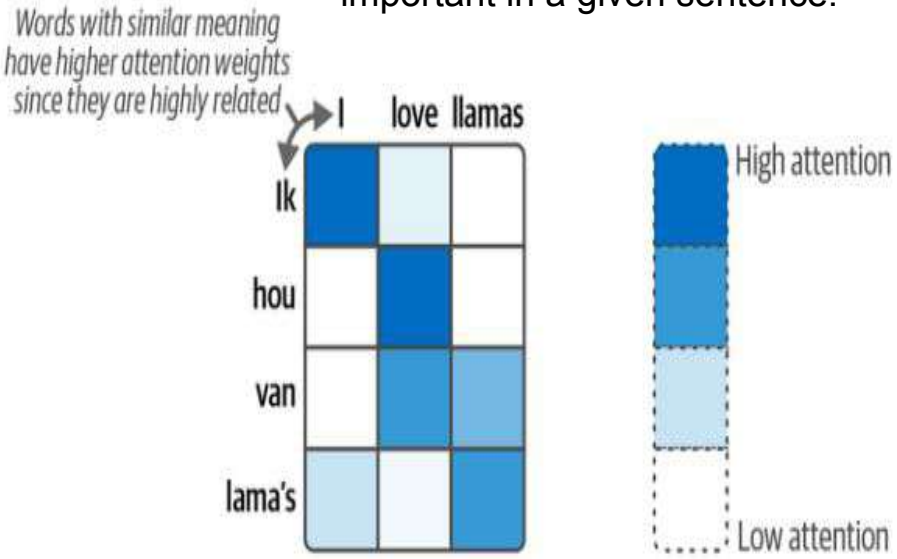


Figure 1-14. Attention allows a model to "attend" to certain parts of sequences that might relate more or less to one another.

Invention of Transformers - Attention Mechanism

- The authors proposed a network architecture called the Transformer, which was solely based on the attention mechanism and removed the recurrence network that we saw previously.

- Compared to the recurrence network, the Transformer could be trained in parallel, which tremendously sped up training.

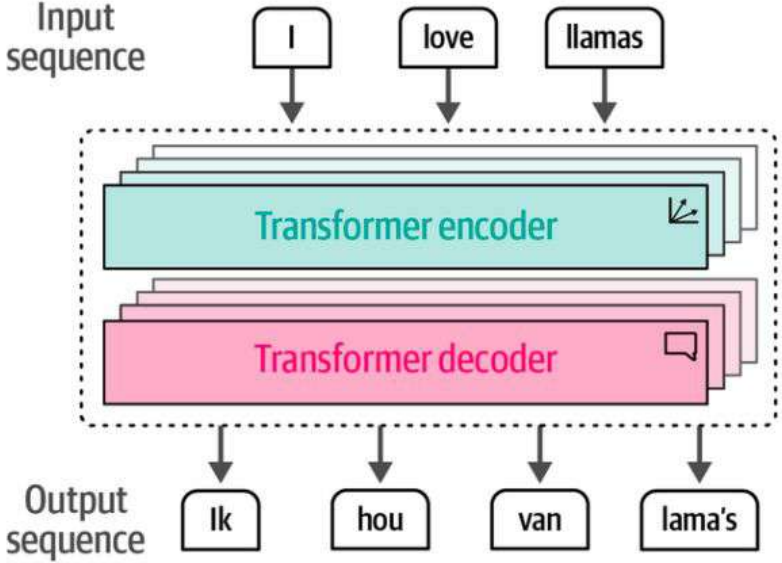


Figure 1-16. The Transformer is a combination of stacked encoder and decoder blocks where the input flows through each encoder and decoder.

In the Transformer, encoding and decoder components are stacked on top of each other, as illustrated in Figure 1-16. This architecture remains autoregressive, needing to consume each generated word before creating a new word.

Encoder Block

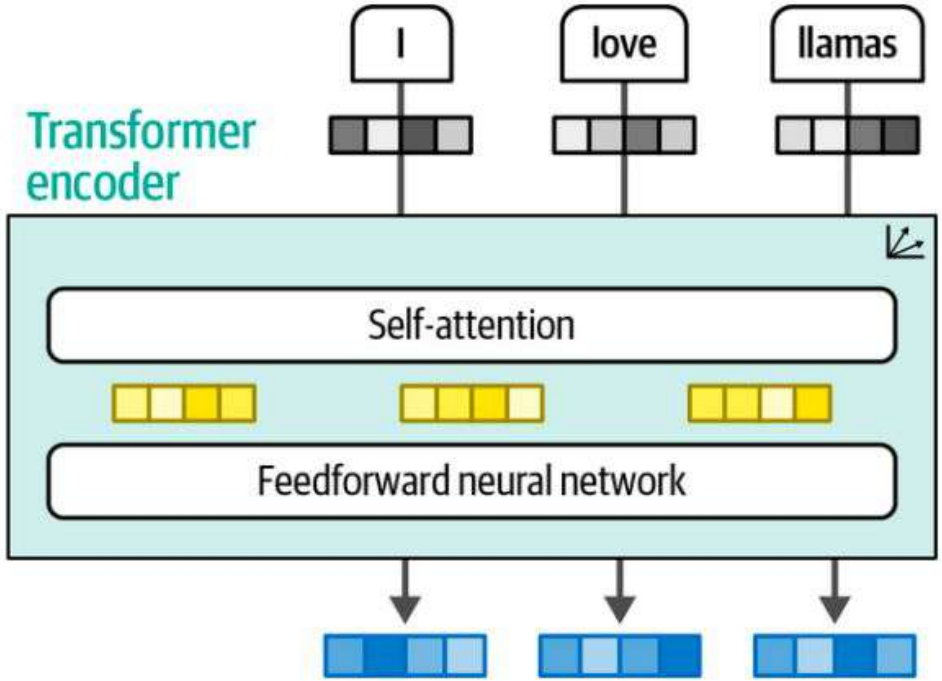


Figure 1-17. An encoder block revolves around self-attention to generate intermediate representations.

The encoder block in the Transformer consists of two parts, self-attention and a feedforward neural network, which are shown in Figure 1-17.

Self attention

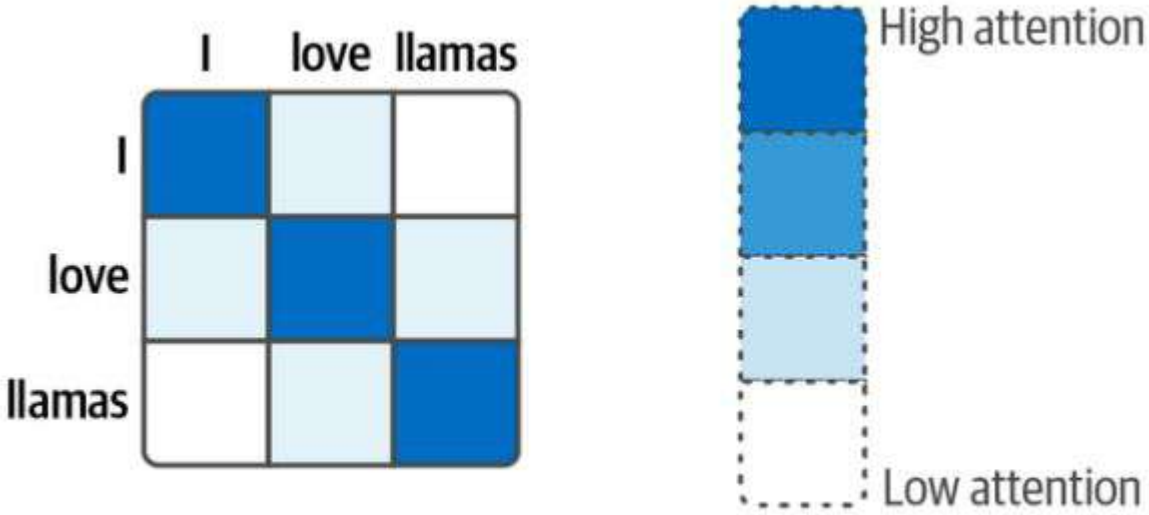


Figure 1-18. Self-attention attends to all parts of the input sequence so that it can "look" both forward and back in a single sequence.

Self-Attention

Mechanism	Attends To	Typical Use Case
Traditional Attention	One sequence attends to another	Encoder-decoder tasks (e.g. translation)
Self-Attention	Each token attends to all others in the same sequence	Contextual representation (e.g. summarization, classification)

	The	cat	sat	on	the	mat
The	0.1	0.2	0.1	0.1	0.3	0.2
cat	0.2	0.1	0.3	0.1	0.1	0.2
…						

Decoder Layer

Compared to the encoder, the decoder has an additional layer that pays attention to the output of the encoder (to find the relevant parts of the input). the self-attention layer in the decoder masks future positions so it only attends to earlier positions to prevent leaking information when generating the output. If it could “peek ahead” at future tokens during training, it would cheat—learning to copy instead of truly predict.

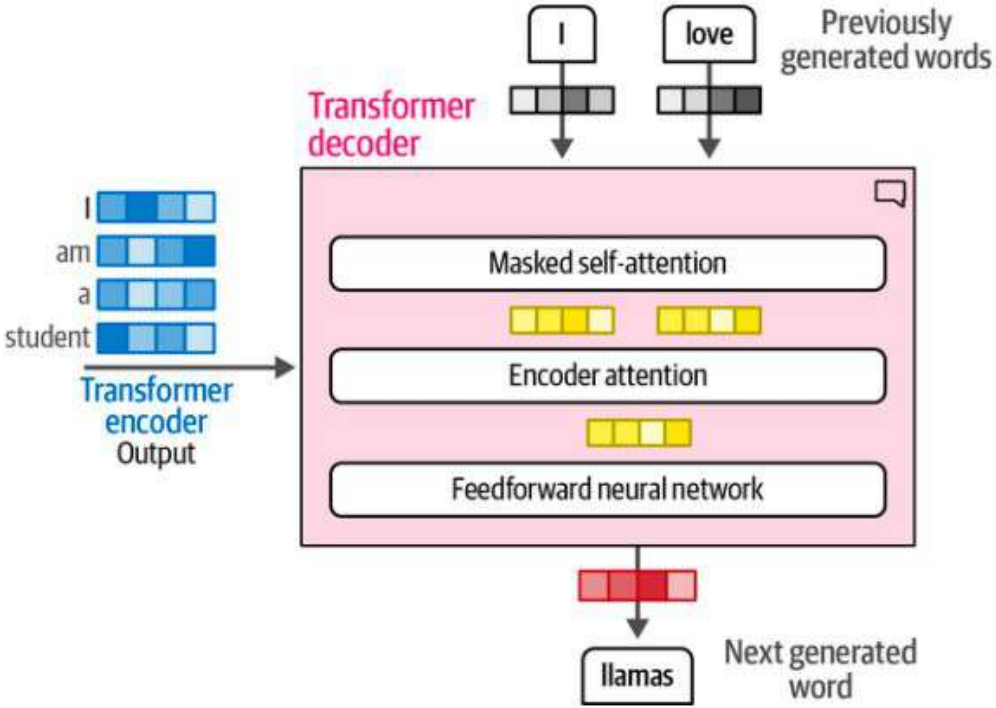


Figure 1-19. The decoder has an additional attention layer that attends to the output of the encoder.

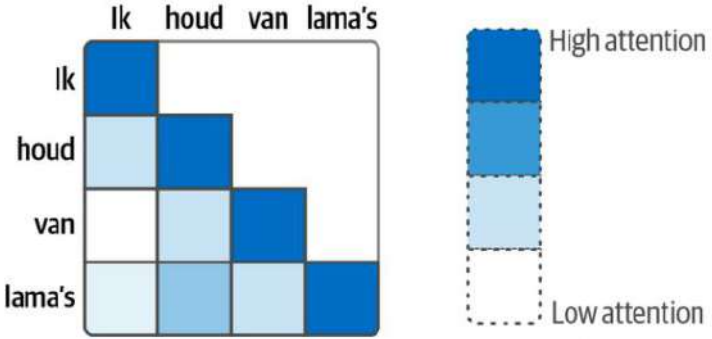


Figure 1-20. Only attend to previous tokens to prevent “looking into the future.”

Types of Transformers

Encoder Only

Decoder Only

Encoder-Decoder

BERT-Bidirectional Encoder Representations from Transformers

The original Transformer model is an encoder-decoder architecture that serves translation tasks well but cannot easily be used for other tasks, like text classification. In 2018, a new architecture called **Bidirectional Encoder Representations from Transformers (BERT)** was introduced that could be leveraged for a wide variety of tasks and would serve as the foundation of Language AI for years to come.6 BERT is an encoder-only architecture that focuses on representing language, as illustrated in Figure 1-21.

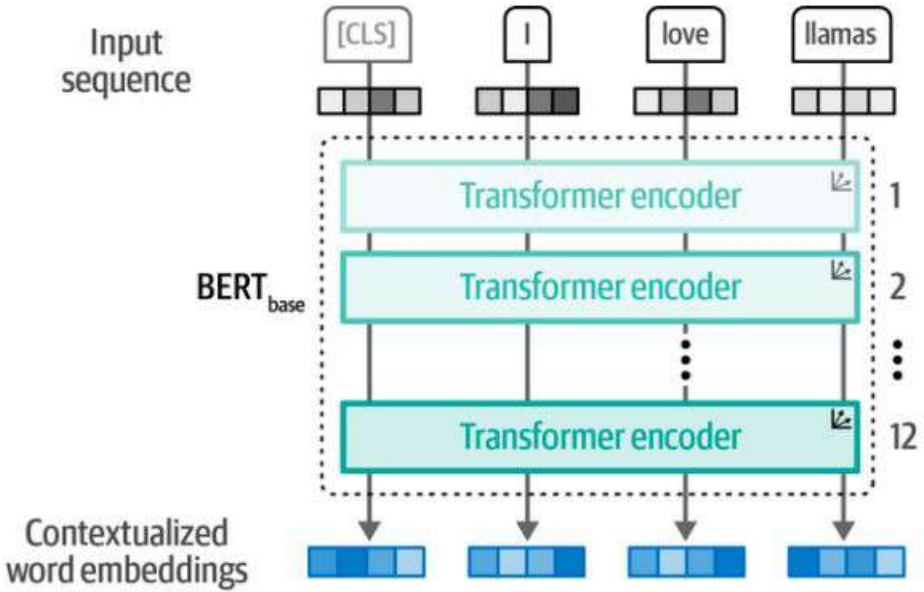


Figure 1-21. The architecture of a BERT base model with 12 encoders.

These encoder blocks are the same as we saw before: self-attention followed by feedforward neural networks. The input contains an additional token, the [CLS] or classification token, which is used as the representation for the entire input. Often, we use this [CLS] token as the input embedding for fine-tuning the model on specific tasks, like classification.

GPT-Decoder Only Model

Similar to the encoder-only architecture of BERT, a decoder-only architecture was proposed in 2018 to target generative tasks. This architecture was called a Generative Pre-trained Transformer (GPT) for its generative capabilities.

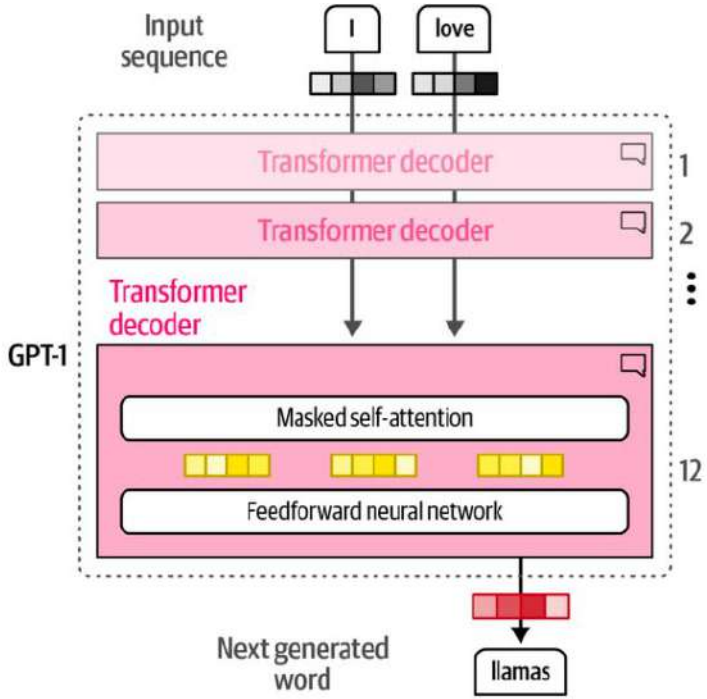


Figure 1-24. The architecture of a GPT-1. It uses a decoder-only architecture and removes the encoder-attention block.

Generative or Completion LLM

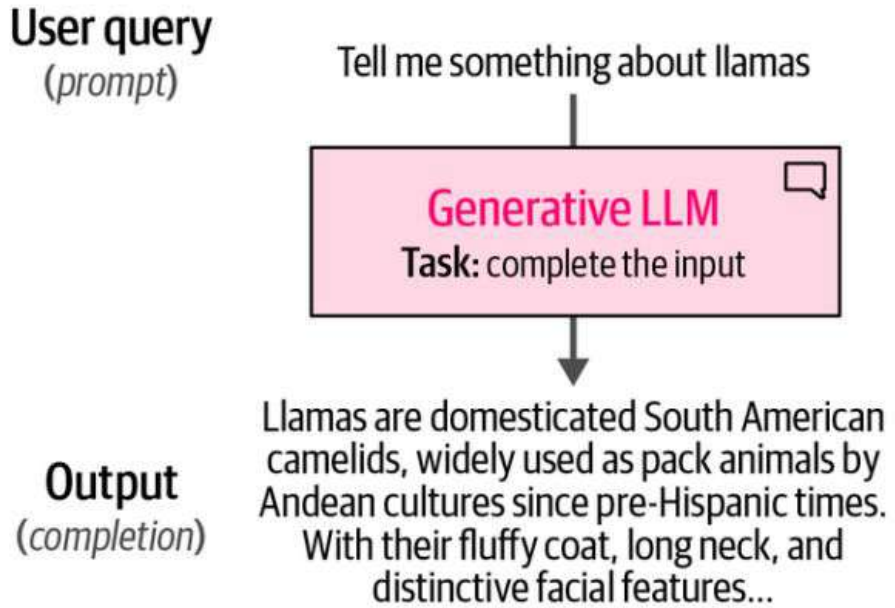


Figure 1-26. Generative LLMs take in some input and try to complete it. With instruct models, this is more than just autocomplete and attempts to answer the question.

Context Window

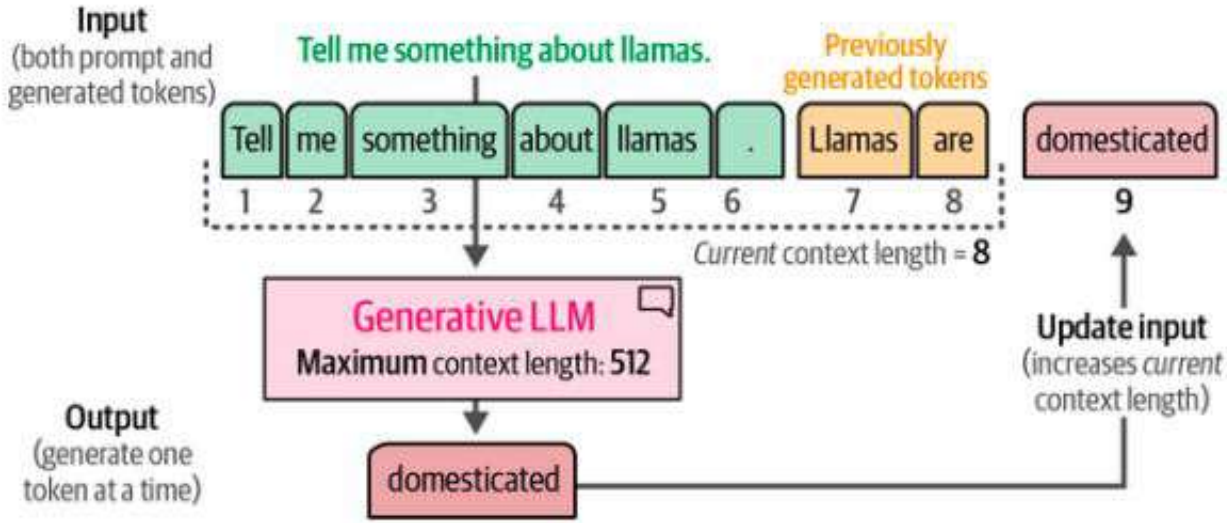


Figure 1-27. The context length is the maximum context an LLM can handle.

A vital part of these completion models is something called the context length or context window. The context length represents the maximum number of tokens the model can process, as shown in Figure 1-27. A large context window allows entire documents to be passed to the LLM. Note that due to the autoregressive nature of these models,

Pretrained Vs. Finetuned LLM

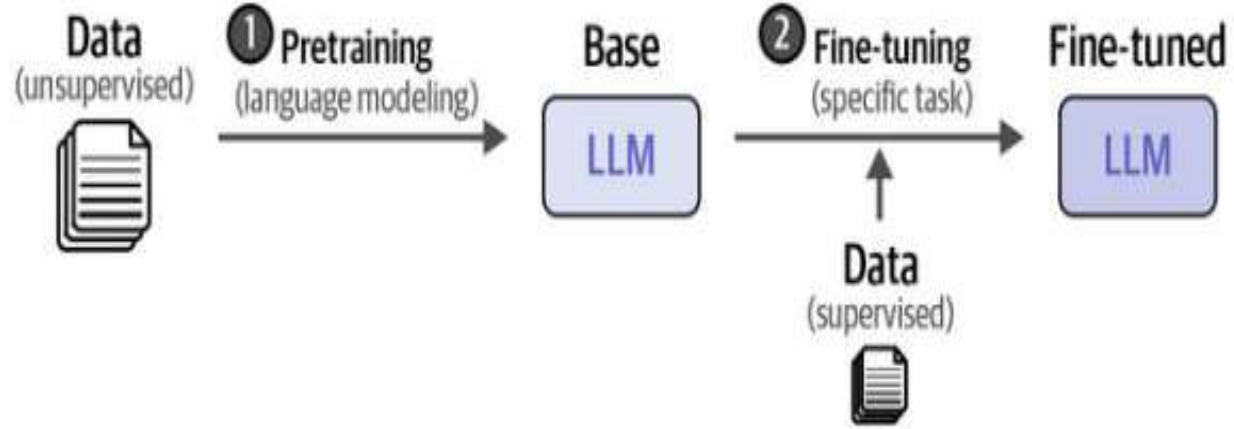


Figure 1-30. Compared to traditional machine learning, LLM training takes a multistep approach.

Chapter 2. Tokens and Embeddings

Tokenization and Embedding

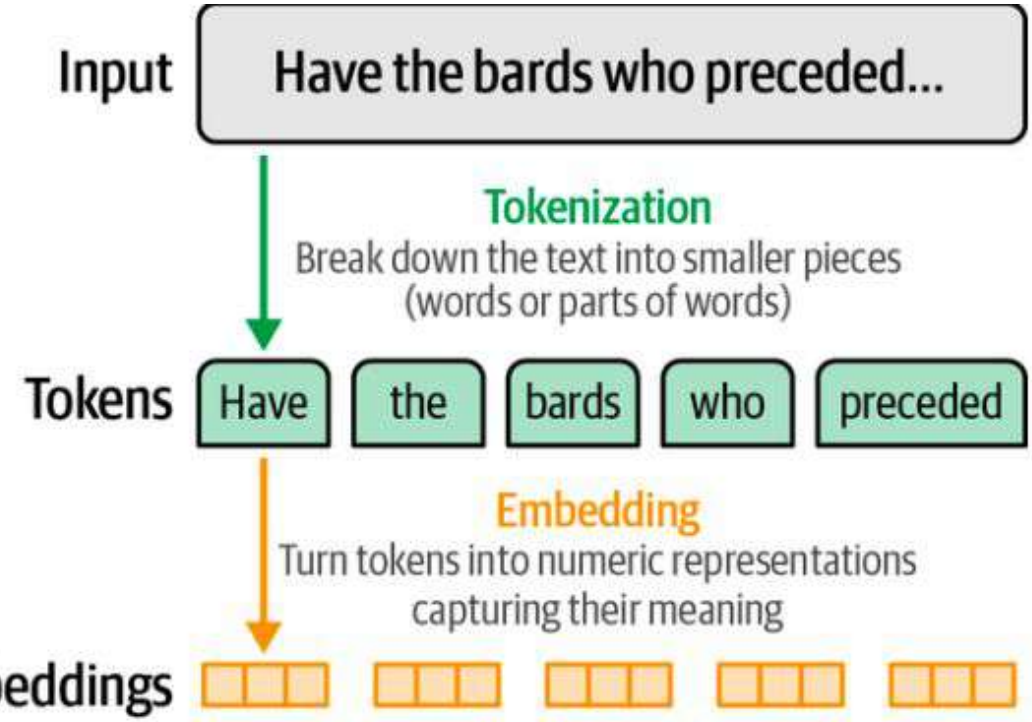


Figure 2-1. Language models deal with text in small chunks called tokens. For the language model to compute language, it needs to turn tokens into numeric representations called embeddings.

Loading and Running an LLM

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-
4k-instruct")
```

We can then proceed to the actual generation. We first declare our prompt, then tokenize it, then pass those tokens to the model, which generates its output. In this case, we're asking the model to only generate 20 new tokens:

```
prompt = "Write an email apologizing to Sarah for the tragic
gardening mishap. Explain how it happened.<|assistant|>"

# Tokenize the input prompt
input_ids = tokenizer(prompt,
return_tensors="pt").input_ids.to("cuda")

# Generate the text
generation_output = model.generate(
  input_ids=input_ids,
  max_new_tokens=20
)

# Print the output
print(tokenizer.decode(generation_output[0]))
```

Output:

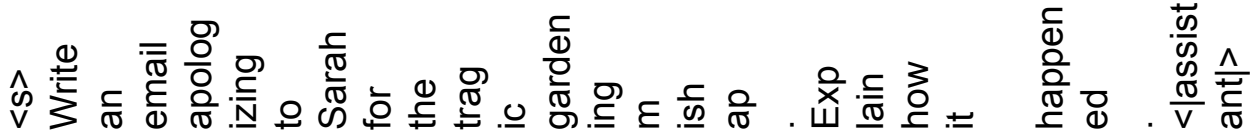
```
<s> Write an email apologizing to Sarah for the tragic
gardening mishap. Explain how it happened.<|assistant|> Subject:
My Sincere Apologies for the Gardening Mishap
```

We first declare our prompt, then tokenize it, then pass those tokens to the model, which generates its output. In this case, we're asking the model to only generate 20 new tokens:

Input to LLM is the Unique TokenID

Let’s print input_ids to see what it holds inside:
tensor([[1, 14350, 385, 4876, 27746, 5281, 304, 19235, 363, 278, 25305, 293, 16423, 292, 286, 728, 481, 29889, 12027, 7420, 920, 372, 9559, 29889, 32001]], device='cuda:0')

This reveals the inputs that LLMs respond to, a series of integers as shown. Each one is the unique ID for a specific token (character,word, or part of a word). These IDs reference a table inside the tokenizer containing all the tokens it knows.



Output of LLM are Tokens

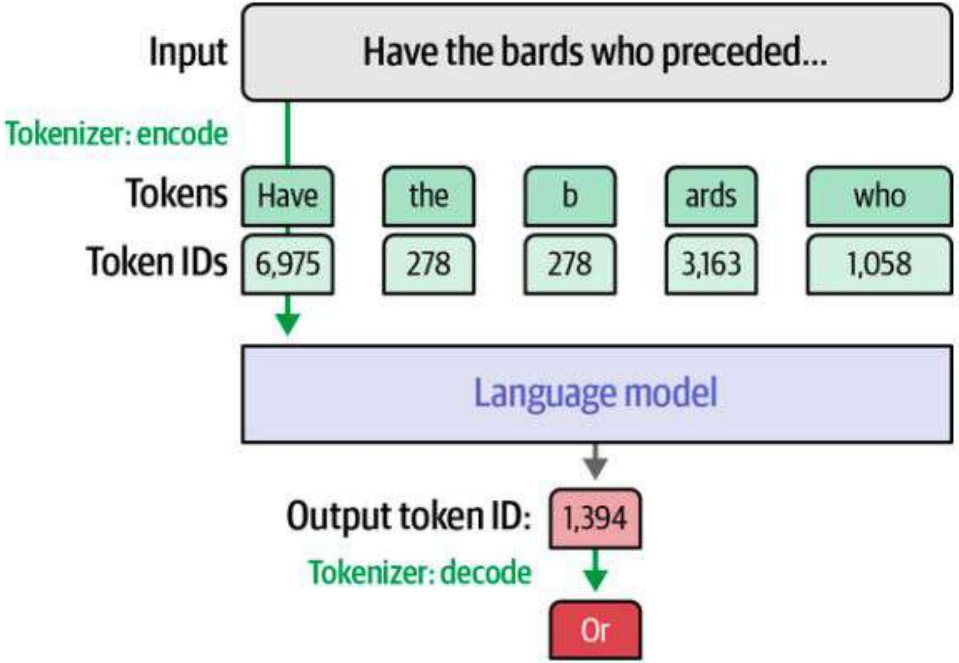


Figure 2-5. Tokenizers are also used to process the output of the model by converting the output token ID into the word or token associated with that ID.

In addition to being used to process the input text into a language model, tokenizers are used on the output of the language model to turn the resulting token ID into the output word or token associated with it.

Properties of Tokens

This is how the tokenizer broke down our input prompt. Notice the following:

- The first token is ID 1 (<s>), a special token indicating the beginning of the text.
- Some tokens are complete words (e.g., Write, an, email).
- Some tokens are parts of words (e.g., apolog, izing, trag, ic).
- Punctuation characters are their own token.
- The space character does not have its own token.

Input to LLM is the Unique TokenID

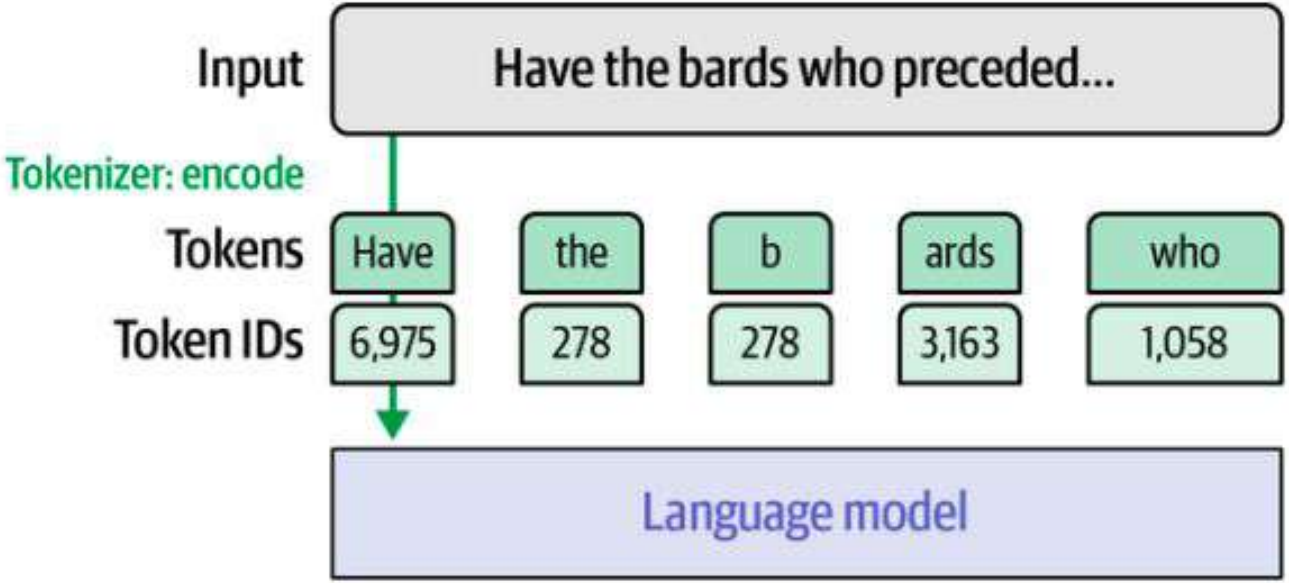


Figure 2-4. A tokenizer processes the input prompt and prepares the actual input into the language model: a list of token IDs. The specific token IDs in the figure are just demonstrative.

What Are Tokens?

Definition: Smallest units of text a model processes
Examples: Words, subwords, characters
Tokenization strategies: whitespace, byte-pair encoding (BPE), WordPiece

Why Tokens?

Language models don’t understand text the way humans do. They need **numerical input** — and raw text is too ambiguous, too varied, and too complex to feed directly into a model.

Tokenization is the bridge between human language and machine-readable format

Tokenization Methods

- Word
- Subword
- Character
- Byte

Word Level Tokenization

- It’s the process of **splitting a sentence or text into individual words**, treating each word as a separate token. These tokens are the basic units that NLP models use to understand and process language.
- **Simplifies text** for downstream tasks like classification, translation, or sentiment analysis.
- **Preserves semantic units** better than character-level tokenization.
- **Works well for languages with clear word boundaries**, like English

Input: "Machine learning is fascinating."
Output: ["Machine", "learning", "is", "fascinating"]

Disadvantages Word Level Tokenization

- One challenge with word tokenization is that the tokenizer may be **unable to deal with new words** that enter the dataset after the tokenizer was trained.
- Also results in a vocabulary that has a lot of tokens with minimal differences between them (e.g., apology, apologize, apologetic, apologist). This latter challenge is resolved by subword tokenization as it has a token for apolog, and then suffix tokens (e.g., -y,-ize, -etic, -ist)

Subword Level Tokenization

Subword-level tokenization is a clever middle ground between word-level and character-level tokenization. It breaks words into smaller, meaningful units—called **subwords**—which helps models handle rare words, misspellings, and morphological variants more gracefully.

Word: "unhappiness"
Subword tokens: ["un", "happi", "ness"]

Advantages

- **Handles out-of-vocabulary (OOV) words** by breaking them into known subwords.
- **Reduces vocabulary size** while maintaining semantic richness.
- **Improves generalization** across word forms (e.g., “run”, “running”, “runner”).

Disadvantages

- May produce **non-intuitive splits** (e.g., “##ing” or “@@ness”).
- Requires **pre-training** to build the subword vocabulary.
- Slightly more complex to implement than word-level tokenization.

Character Level Tokenization

Character-level tokenization is the process of breaking down text into **individual characters**, including letters, digits, punctuation, and even spaces. It's the most granular form of tokenization and can be surprisingly powerful in certain NLP tasks.

Input: "Token"
Output: ["T", "o", "k", "e", "n"]

Ådvantages

Character-level tokenization is especially useful when:

- **Spelling errors or typos** are common (e.g., user-generated content)
- **Languages lack clear word boundaries** (e.g., Chinese, Thai)
- **Creative or noisy text** is involved (e.g., social media, poetry)
- You want to **build models from scratch** without relying on predefined vocabularies
- **No out-of-vocabulary (OOV) issues**: Every character is known.
- **Language-agnostic**: Works across scripts and writing systems.
- **Fine-grained control**: Useful for tasks like spelling correction, name normalization, or stylometry.
-

Limitations

- **Longer sequences**: Text becomes much longer, increasing computational cost.
- **Loss of semantic units**: Characters alone don't carry meaning like words or subwords do.
- **Harder to model context**: Requires deeper architectures to capture long-range dependencies.

Byte-Level Tokenization

Byte tokenization method breaks down tokens into the individual bytes that are used to represent unicode characters.

It treats **text as a sequence of raw bytes**, rather than characters or words, allowing for **universal coverage** across languages, symbols, and even emojis.

Input: "café"
UTF-8 bytes: [99, 97, 102, 195, 169]
Tokens: ["c", "a", "f", "Ã", "©"]

Advantages

- **No unknown tokens**: Every character, emoji, or symbol can be represented as bytes.
- **Compact base vocabulary**: Just 256 tokens (one for each byte).
- **Language-agnostic**: Works across all scripts and writing systems.
- **Handles noisy or creative text**: Great for social media, code, or multilingual input.

Tokenization Methods

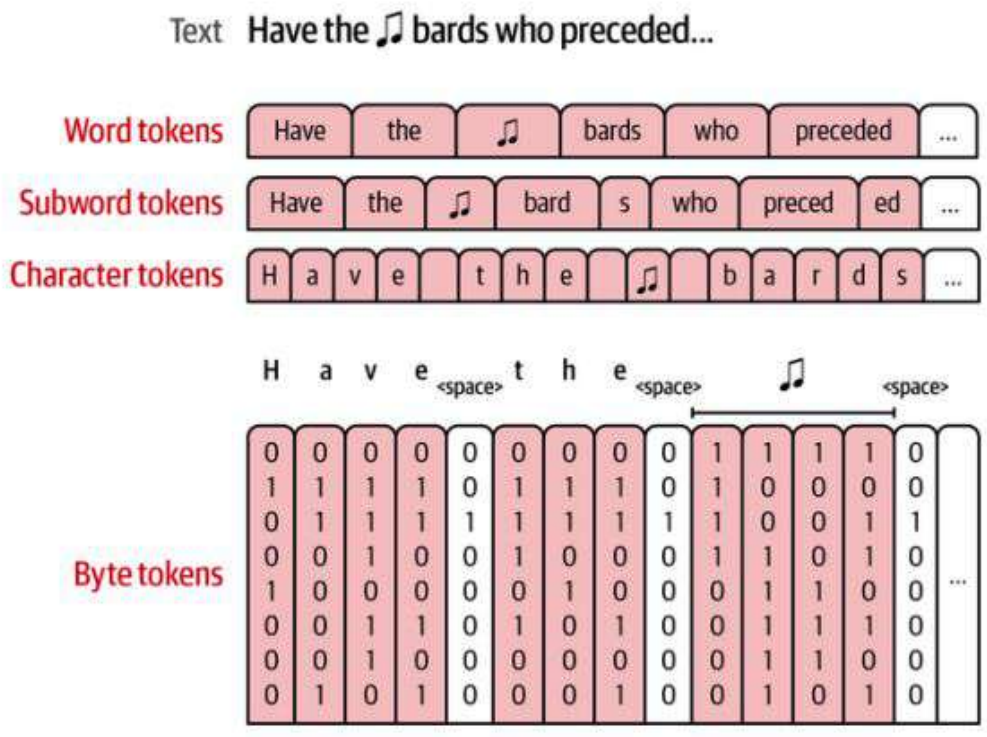


Figure 2-6. There are multiple methods of tokenization that break down the text to different sizes of components (words, subwords, characters, and bytes).

Problems Solved by Tokenization

Challenge	How Tokenization Helps
Words vary in form (run, running, ran)	Breaks into subwords → captures shared meaning
Rare or new words (photosynthesis)	Splits into known chunks → avoids unknown tokens
Misspellings or typos	Subword units still make sense (transfrmers)
Multilingual text	Uses shared vocabulary across languages
Model input size limits	Controls token count → fits within model window

Subword Level Tokenization Methods

Feature	Byte-Pair Encoding (BPE)	WordPiece	SentencePiece
Core Idea	Greedy merging of frequent character pairs	Likelihood-based subword merges	Unigram LM or BPE over raw text
Training Input	Pre-tokenized text	Pre-tokenized text	Raw text (no preprocessing needed)
Lossiness	Fully lossless (preserves spaces)	Lossy (spaces not preserved)	Partially lossless (preserves one space)
Sampling Capability	Deterministic	Deterministic	Can sample multiple tokenizations
Multilingual Support	Limited	Limited	Strong (used in multilingual models)
Vocabulary Construction	Frequency-based merges	Likelihood-based merges	Probabilistic model over subwords
Used In LLMs	GPT-2, GPT-3, GPT-4, Falcon, RoBERTa	BERT, DistilBERT, ALBERT	T5, XLNet, mT5, ByT5, PaLM

1. Byte Pair Encoding

Corpus of words, Number of merge operations N Tokenized corpus, Subword vocabulary
Preprocess: for *each word in corpus* **do**
end
Split word into characters Append end-of-word marker (e.g., _)
Initialize vocabulary with all unique characters and _
for *merge_step* $= 1$ *to* N **do**
end
Count frequency of all adjacent symbol pairs in corpus Identify most frequent pair (a, b) Merge (a, b) into new symbol ab Replace all occurrences of (a, b) with ab in corpus Add ab to vocabulary
return Final tokenized corpus and vocabulary

BPE - Example

Byte Pair Encoding Example: `aa abc abc`

Initial Corpus: `["aa", "abc", "abc"]`

Step 1: Preprocessing

- "aa" → `["a", "a", "_"]`
- "abc" → `["a", "b", "c", "_"]`
- "abc" → `["a", "b", "c", "_"]`

Step 2: Iterative Merging

1. **Iteration 1:**
   - Most frequent pair: `("a", "b")`
   - Merge into: `"ab"`
   - Corpus updates:
     - `["a", "a", "_"]`
     - `["ab", "c", "_"]`
     - `["ab", "c", "_"]`
   - Vocabulary: `["a", "b", "c", "_", "ab"]`
2. **Iteration 2:**
   - Most frequent pair: `("ab", "c")`
   - Merge into: `"abc"`
   - Corpus updates:
     - `["a", "a", "_"]`
     - `["abc", "_"]`
     - `["abc", "_"]`
   - Vocabulary: `["a", "b", "c", "_", "ab", "abc"]`

3. **Iteration 3:**
   - Most frequent pair: `("a", "a")`
   - Merge into: `"aa"`
   - Corpus updates:
     - `["aa", "_"]`
     - `["abc", "_"]`
     - `["abc", "_"]`
   - Vocabulary: `["a", "b", "c", "_", "ab", "abc", "aa"]`

Final Tokenized Corpus:

- `["aa", "_"]`
- `["abc", "_"]`
- `["abc", "_"]`

Final Vocabulary: `["a", "b", "c", "_", "ab", "abc", "aa"]`

Homework BPE

Imagine a Corpus of the following words mentioned along with their frequencies meaning `"hug"` was present 10 times in the corpus, `"pug"` 5 times, `"pun"` 12 times, `"bun"` 4 times, and `"hugs"` 5 times.

("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)

Using BPE algorithm iterate for 5 merge steps and return the final vocabulary as well as the final tokenized corpus.

Hint: In the first iteration the pair (u,g) is present (10+5+5=) 20 times, (p,u) is present 17 times and (u,n) is present 16 times.

Wordpiece - Training Algorithm

- WordPiece is the tokenization algorithm Google developed to pretrain BERT. It has since been reused in quite a few Transformer models based on BERT, such as DistilBERT, MobileBERT, Funnel Transformers, and MPNET. It's very similar to BPE in terms of the training, but the actual tokenization is done differently.
- Then, again like BPE, WordPiece learns merge rules. The main difference is the way the pair to be merged is selected. Instead of selecting the most frequent pair, WordPiece computes a score for each pair, using the following formula:

$$score=(freq\_of\_pair) / (freq\_of\_first\_element \times freq\_of\_second\_element)$$

- By dividing the frequency of the pair by the product of the frequencies of each of its parts, the algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary. For instance, it won't necessarily merge ("un", "##able") even if that pair occurs very frequently in the vocabulary, because the two pairs "un" and "##able" will likely each appear in a lot of other words and have a high frequency. In contrast, a pair like ("hu", "##gging") will probably be merged faster (assuming the word "hugging" appears often in the vocabulary) since "hu" and "##gging" are likely to be less frequent individually.

Wordpiece- Tokenization Algorithm

WordPiece Tokenization with Vocabulary: [UNK, a, aa, ab, abc, ##b, ##c]

Corpus: ["aa", "abc", "abc"]

Case 1: abc and ab are in the vocabulary

- "aa":
  - Longest prefix match: "aa" ∈ vocab
  - Tokenized: ["aa"]
- "abc":
  - Longest prefix match: "abc" ∈ vocab
  - Tokenized: ["abc"]
- Final Tokenized Corpus: [["aa"], ["abc"], ["abc"]]

Case 2: abc and ab are not in the vocabulary
Step-by-step Tokenization of "abc":

1. Start at position 0: Try longest match from vocab
   - Try "abc" → not in vocab
   - Try "ab" → not in vocab
   - Try "a" → match
   - Emit: "a"
2. Position 1: Remaining string is "bc"
   - Try "##bc" → not in vocab
   - Try "##b" → match
   - Emit: "b"
3. Position 2: Remaining string is "c"
   - Try "##c" → match
   - Emit: "c"

Final Tokenization: ["a", "##b", "##c"]
Note: WordPiece does **not** backtrack or look ahead to match "##bc" unless it's explicitly in the vocabulary. It only checks longest valid prefix at each position. **Fallback Behavior:**

- If no prefix match is found at any step, use [UNK].
- Example: "xyz" → ["[UNK]"]

How WordPiece Tokenization Works Post-Training

Once the vocabulary is trained (typically using likelihood maximization over a corpus), the tokenization process follows this **greedy strategy**:

1. **Start at the beginning of the word.**
2. **Find the longest substring** that matches a token in the vocabulary.
3. **Add that token to the output.**
4. **Move forward** in the word and repeat until the entire word is tokenized.

Let's say the WordPiece vocabulary includes: `["un", "##aff", "##able", "##ability"]`

Tokenizing `"unaffable"` would proceed as:

- Start with `"unaffable"`
- Match `"un"` → ✅
- Remaining: `"affable"`
- Match `"##aff"` → ✅
- Remaining: `"able"`
- Match `"##able"` → ✅
- Final tokens: `["un", "##aff", "##able"]`

SentencePiece

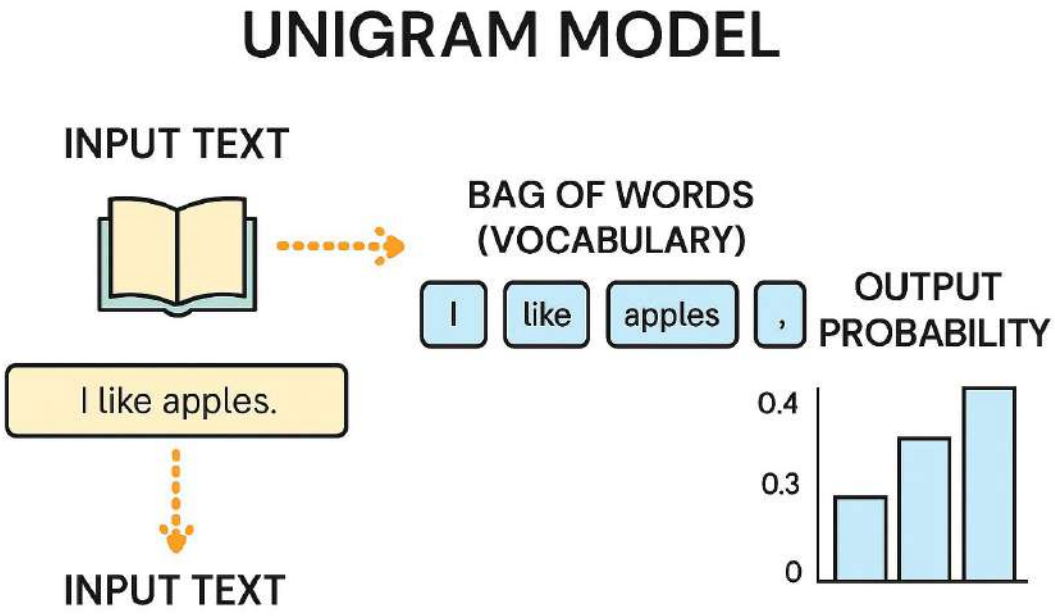
Overview

- SentencePiece is a language-independent subword tokenization algorithm developed by Google, de-signed to train directly on raw text without requiring pre-tokenization. Unlike WordPiece or BPE, which assume space-separated words as input, SentencePiece treats the input as a raw stream of Unicode characters and learns subword units using either Byte Pair Encoding (BPE) or Unigram Language Model.
- This approach enables consistent handling of whitespace, punctuation, and multilingual corpora. SentencePiece introduces a special meta-symbol to represent spaces, allowing
- it to model word boundaries explicitly.
- The Unigram Language Model is one of the core algorithms used in SentencePiece for subword tokenization. Unlike BPE, which builds the vocabulary by merging frequent pairs, the Unigram approach starts with a large initial vocabulary of candidate subwords and iteratively prunes it
- At each step, the algorithm evaluates the likelihood of the corpus given the current vocabulary and removes the subwords whose removal least increases the overall loss.
- The most used model for sentencepiece tokenization is the Unigram Model

Unigram Model

- The Unigram algorithm is used in combination with SentencePiece, which is the tokenization algorithm used by models like AlBERT, T5, mBART, Big Bird, and XLNet.

- The **Viterbi algorithm** is used to find the **most probable segmentation**—i.e., the sequence of tokens that maximizes the product of their unigram probabilities.



Viterbi Decoding in the Unigram Model: Example with ”abc

$$\text{Cost}(t_i) = -\log p_i$$

Vocabulary and Costs

We assume the following vocabulary and probabilities:

Token	**Probability** p	**Cost** $-\log p$
a	0.20	1.61
b	0.20	1.61
c	0.20	1.61
ab	0.15	1.90
bc	0.15	1.90
abc	0.10	2.30

Possible Segmentations of the word “abc”

Optimal Path via Viterbi Decoding
The Viterbi algorithm selects the path with the minimum total cost. In this case:

Best Segmentation = ["abc"], Total Cost = 2.30

This segmentation is the most probable under the Unigram model.

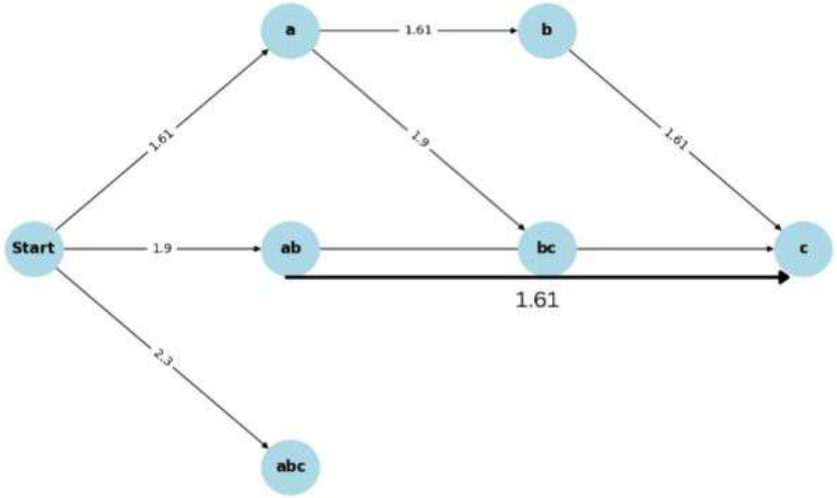


Figure 4: The cost lattice for ”abc”

- Segmentation: `["a", "b", "c"]`
  Cost: $1.61 + 1.61 + 1.61 = 4.83$
- Segmentation: `["ab", "c"]`
  Cost: $1.90 + 1.61 = 3.51$
- Segmentation: `["a", "bc"]`
  Cost: $1.61 + 1.90 = 3.51$
- Segmentation: `["abc"]`
  Cost: 2.30

Learned probabilities and the EM Algorithm

Learned Probabilities

The probability p(t) for a subword is not just the raw frequency in the corpus but rather a learned probability that reflects how often the subword appears in the valid segmentations of the corpus during training.

The Unigram model learns the probability of each subword by:

- Starts with a probability of subwords
- Iteratively adjusts the probability of each subword based on how likely it appears in valid segmentation of the training corpus.

Unigram Model uses the EM algorithm that iteratively refines token probabilities to maximize the likelihood of the corpus.

- E-step: Compute expected counts for each token based on all valid segmentations.
- M-step: Update token probabilities using normalized expected counts.

Example - Learned Probailities.

Example Corpus: aa abc abc

Token	Probability
a	0.1
aa	0.3
abc	0.6

Initial Vocabulary and Probabilities

Step -1 Find all the valid segmentations of the Corpus

🧮 Final Valid Segmentations of Entire Corpus

We now combine valid segmentations of each part:

1. `[aa] [abc] [abc]`
2. `[a, a] [abc] [abc]`

These are the **only valid segmentations** using the given vocabulary.

Step-2 Find the Total Probability for Each segmentation

🔹 Segmentation 1: `[aa] [abc] [abc]`

- Probability:
  $P = 0.3 \times 0.6 \times 0.6 = 0.108$P = 0.3 \times 0.6 \times 0.6 = 0.108

🔹 Segmentation 2: `[a, a] [abc] [abc]`

- Probability:
  $P = 0.1 \times 0.1 \times 0.6 \times 0.6 = 0.0036$
  P = 0.1 \times 0.1 \times 0.6 \times 0.6 = 0.0036

Step-3 Normalize the probability

Segmentation	Raw Probability	Normalized Probability
`[aa] [abc] [abc]`	0.108	$\frac{0.108}{0.1116} \approx 0.968$ \frac{0.108}{0.1116} \approx 0.968
`[a, a] [abc] [abc]`	0.0036	$\frac{0.0036}{0.1116} \approx 0.032$ \frac{0.0036}{0.1116} \approx 0.032

Step-4 Calculate expected count for each subword

Subword	Count in Segmentation 1 (`[aa] [abc] [abc]`)	Count in Segmentation 2 (`[a, a] [abc] [abc]`)	Expected Count
`a`	0	2	0·0.968 + 2·0.032 = **0.064**
`aa`	1	0	1·0.968 + 0·0.032 = **0.968**
`abc`	2	2	2·0.968 + 2·0.032 = **2.000**

Step-4 Calculate expected count for each subword

Subword	Expected Count	Updated Probability	Calculation Breakdown
`a`	0.064	**0.0211**	0.064 ÷ 3.032 = 0.0211
`aa`	0.968	**0.3193**	0.968 ÷ 3.032 = 0.3193
`abc`	2.000	**0.6596**	2.000 ÷ 3.032 = 0.6596

Conclusions

As seen the probability of tokens aa and abc increased and a decreased because the EM step saw that in the training corpus aa abc abc

- ‘a’ did not occur
- ‘aa’ occurred once so it increased slightly
- ‘abc’ occurred twice so it increased more

Final Tokenization of aa abc abc using Unigram

- Given the initial learned probabilities and the subwords, Unigram model finds the most probable segmentation using the viterbi algorithm then using the EM algorithm it updates their learned probabilities. The most probable segmentation then is chosen as the final tokenized corpus.
- So for the given vocabulary and corpus the best tokenization of “aa abc abc” is

“aa abc abc”

Embedding

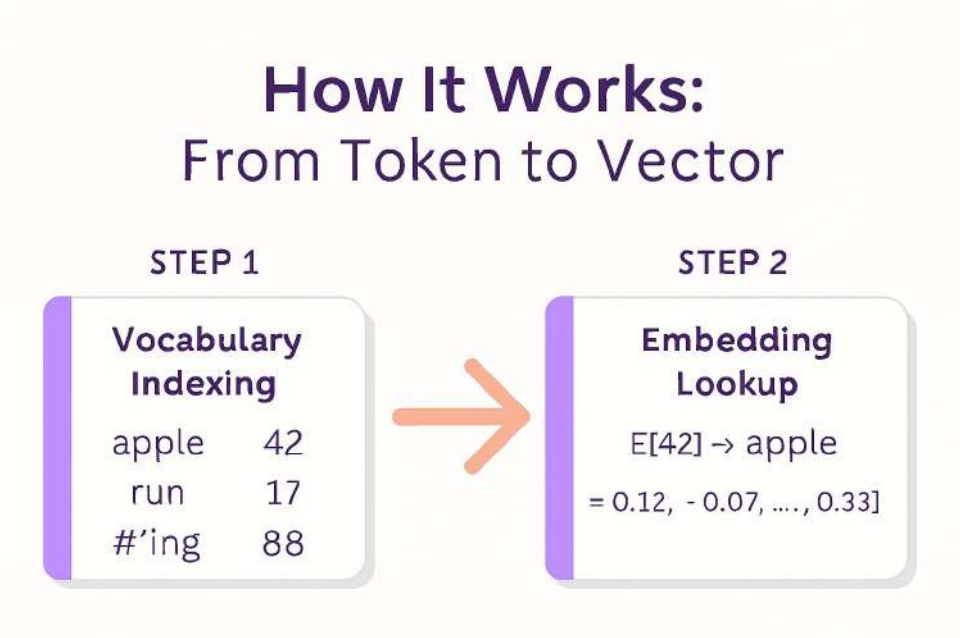
- In the starting we discussed that for a LLM model to understand process and generate language, the language must be modelled and represented as numericals. Tokenization solves the first part and models language as tokens.

Token embeddings are **dense vector representations** of tokens (words, subwords, or characters) that capture their **semantic and syntactic properties** in a continuous space.

Instead of treating tokens as discrete symbols (like one-hot vectors), embeddings allow models to **learn relationships** between tokens based on context and usage.

What Is `E[42]`?

- `E` is the **embedding matrix**: a learnable table of shape V×dV \times d, where:
  - VV = vocabulary size (e.g., 30,000 tokens)
  - dd = embedding dimension (e.g., 300 or 768)
- `E[42]` means: → Take the **42nd row** of this matrix → That row is the **embedding vector** for the token `"apple"`



Embedding Vector

- An **embedding vector** is a dense, numerical representation of a token (word, subword, or character).
- Instead of using sparse one-hot encodings, embeddings map tokens to **continuous vector spaces**.
- Each token is assigned a vector of fixed dimension (e.g., 300 or 768), learned during training.

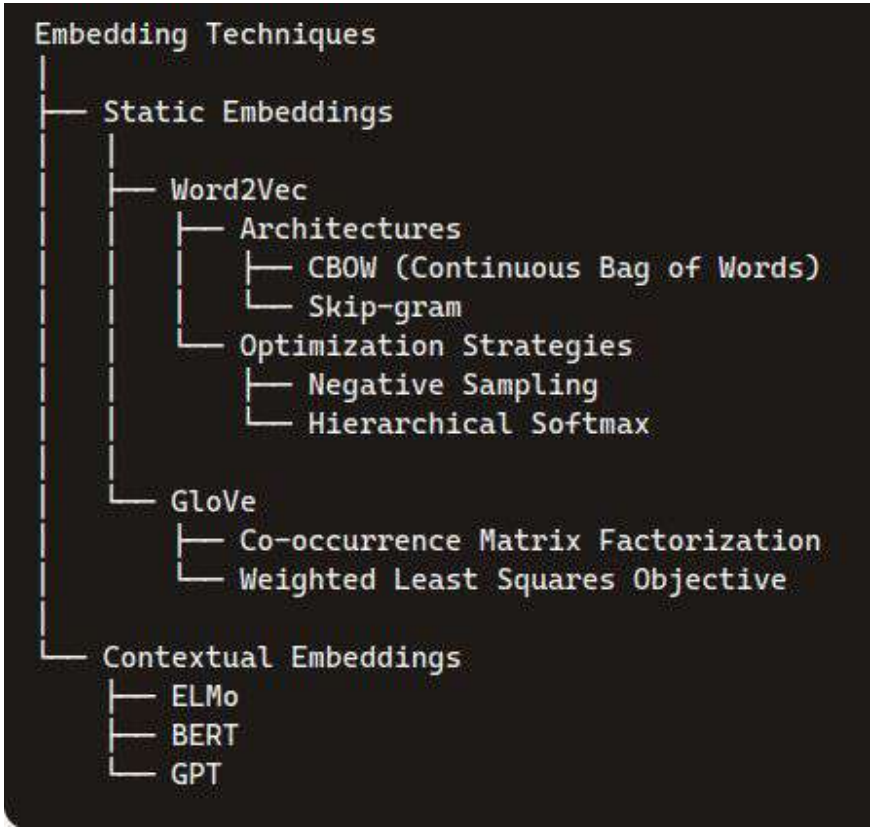
How Do Embeddings Help in Context?

- In **contextual models** (like BERT or GPT), embeddings are **dynamic**:
  - `"bank"` in `"river bank"` ≠ `"bank"` in `"money bank"`
- The model adjusts embeddings based on surrounding words, enabling:
  - **Disambiguation**
  - **Context-aware predictions**
  - **Better downstream performance** (classification, translation, etc.)

📊 Example Embedding Matrix (dim = 3)

Token	Embedding Vector
`king`	[0.52, 0.81, 0.30]
`queen`	[0.50, 0.79, 0.28]
`man`	[0.49, 0.75, 0.25]
`woman`	[0.47, 0.73, 0.23]

Types of Embedding



Static Embedding - Word2Vec

For each word, we consider **2 words before and 2 words after** as its context. Let's few examples:

🔹 **Central Word:** `make`

Context Window: `shalt`, `not`, `a`, `machine`

- **CBOW**:
  - Input: `[shalt, not, a, machine]`
  - Target: `make`
- **Skip-gram**:
  - Input: `make`
  - Targets: `shalt`, `not`, `a`, `machine`

🔹 **Central Word:** `likeness`

Context Window: `the`, `in`, `of`, `a`

- **CBOW**:
  - Input: `[the, in, of, a]`
  - Target: `likeness`
- **Skip-gram**:
  - Input: `likeness`
  - Targets: `the`, `in`, `of`, `a`

We move a window across the sentence, focusing on one **center word** at a time and collecting **2 words before and 2 words after** as context.

🧠 What's Happening?

- **CBOW**: Predict the center word from its context.
  - Learns: "If I see `shalt`, `not`, `a`, `machine`, the center is likely `make`."
- **Skip-gram**: Predict context words from the center word.
  - Learns: "If I see `make`, it's likely surrounded by `shalt`, `not`, `a`, `machine`."

Static Embedding - Word2Vec - Negative Sampling

If, however, we have a dataset of only a target value of 1, then a model can cheat and ace it by outputting 1 all the time. To get around this, we need to enrich our training dataset with examples of words that are not typically neighbors. These are called negative examples and are shown in Figure 2-13.

Word 1	Word 2	Target	
not	thou	1	Positive examples
not	shalt	1	
not	make	1	
not	a	1	
thou	apothecary	0	Negative examples
not	sublime	0	
make	def	0	
a	playback	0	

Figure 2-13. We need to present our models with negative examples: words that are not usually neighbors. A better model is able to better distinguish between the positive and negative examples.

Contextual Embeddings

Unlike traditional embeddings (Word2Vec, GloVe), which assign **one static vector per word**, contextual embeddings generate **different vectors for the same word depending on its sentence context**.

Example:

- `"bank"` in `"river bank"` → vector related to nature
- `"bank"` in `"money bank"` → vector related to finance

Same word, **different meaning**, and therefore **different embedding**.

What Are BERT Embeddings?

In BERT, each input token is transformed into a **contextual embedding**—a dense vector that reflects not just the token itself, but its meaning within the sentence.

Unlike Word2Vec or GloVe, BERT embeddings are **dynamic**:

The same word will have different embeddings depending on its context.

Chapter 3. Looking Inside Large Language Models

The Inputs and Outputs of a Trained Transformer LLM

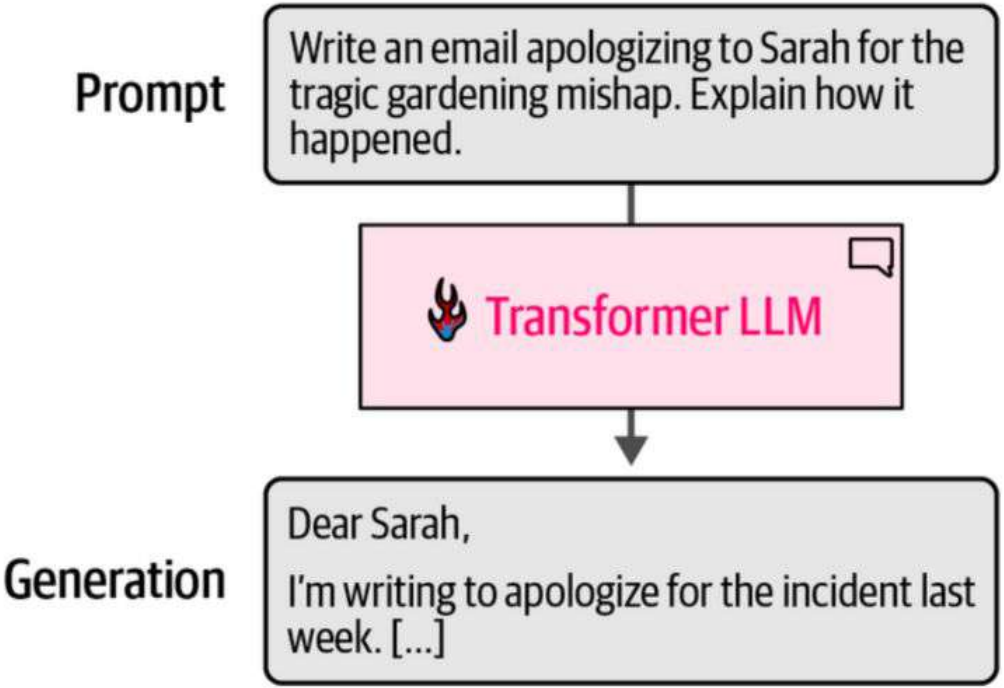


Figure 3-1. At a high level of abstraction, Transformer LLMs take a text prompt and output generated text.

The model does not generate the text all in one operation; it actually generates one token at a time.

Figure 3-2 shows four steps of token generation in response to the input prompt.

Each token generation step is one forward pass through the model (that's machine-learning speak for the inputs going into the neural network and flowing through the computations it needs to produce an output on the other end of the computation graph).

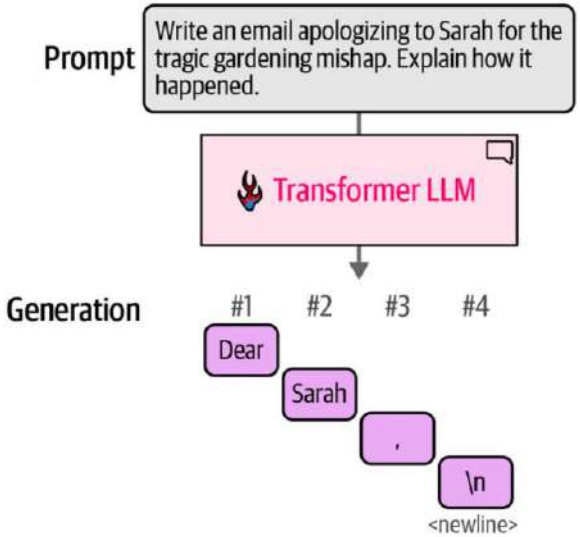


Figure 3-2. Transformer LLMs generate one token at a time, not the entire text at once.

Appending the Output again as Input

After each token generation, we tweak the input prompt for the next generation step by appending the output token to the end of the input prompt. We can see this in Figure 3-3.

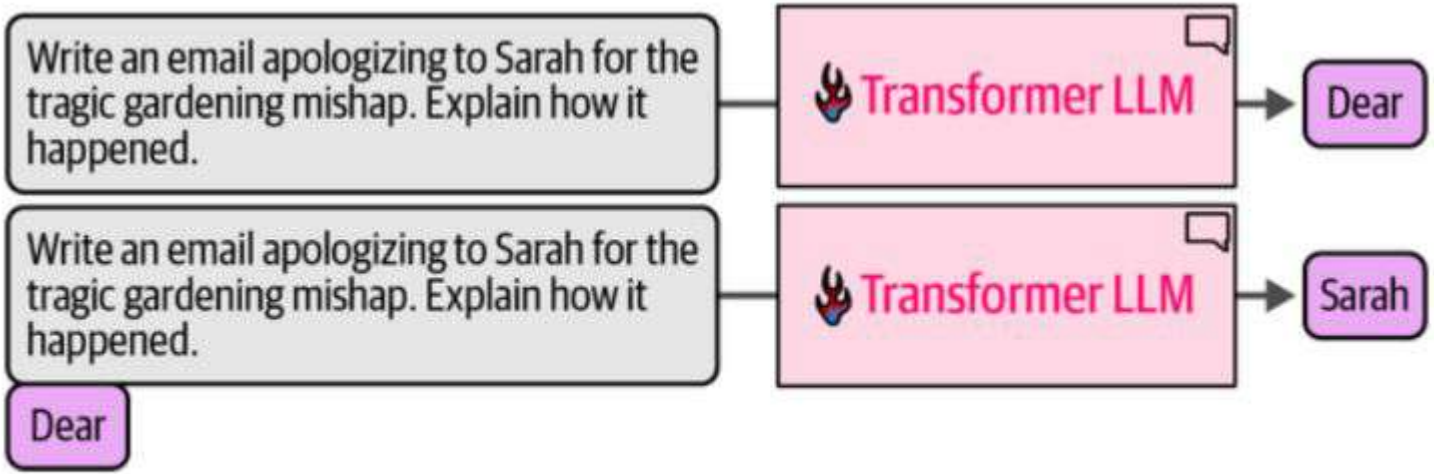


Figure 3-3. An output token is appended to the prompt, then this new text is presented to the model again for another forward pass to generate the next token.

Autoregressive vs. Representation models

There’s a specific word used in machine learning to describe models that consume their earlier predictions to make later predictions (e.g., the model’s first generated token is used to generate the second token). They’re called autoregressive models.

That is why you’ll hear text generation LLMs being called autoregressive models.

This is often used to differentiate text generation models from text representation models like BERT, which are not autoregressive.

The Components of the Forward Pass

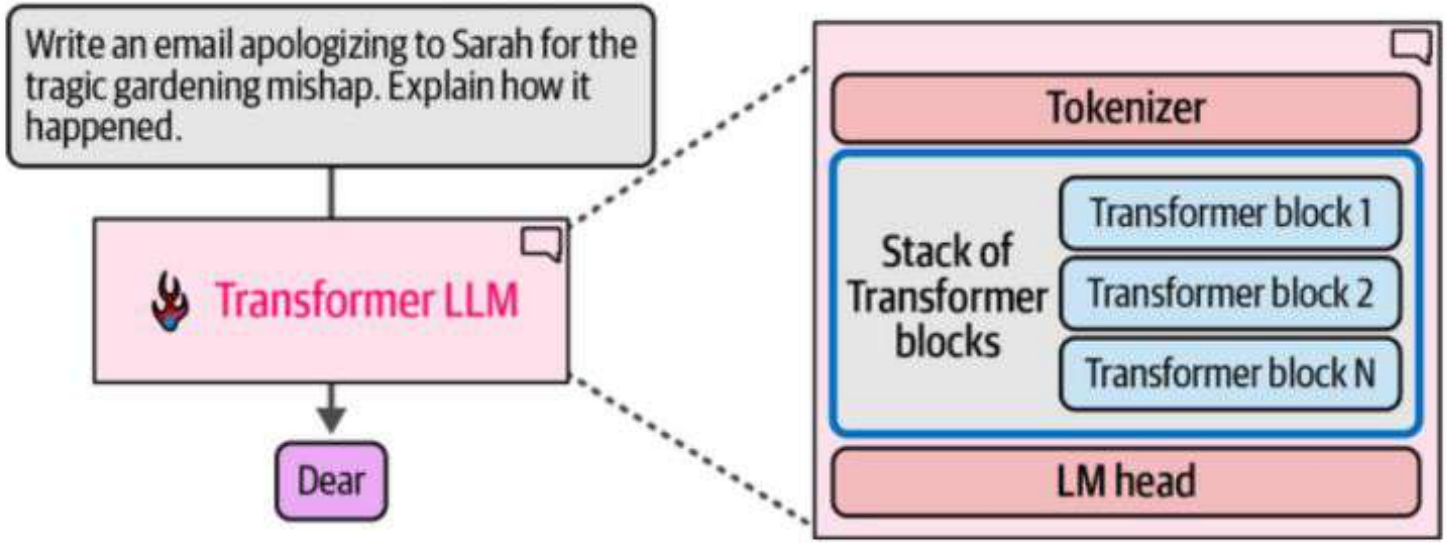


Figure 3-4. A Transformer LLM is made up of a tokenizer, a stack of Transformer blocks, and a language modeling head.

The tokenizer is followed by the neural network: a stack of Transformer blocks that do all of the processing. That stack is then followed by the LM head, which translates the output of the stack into probability scores for what the most likely next token is.

Flow of Control During Token Generation

For each generated token, the process flows once through each of the Transformer blocks in the stack in order, then to the LM head, which finally outputs the probability distribution for the next token, seen in Figure 3-6.

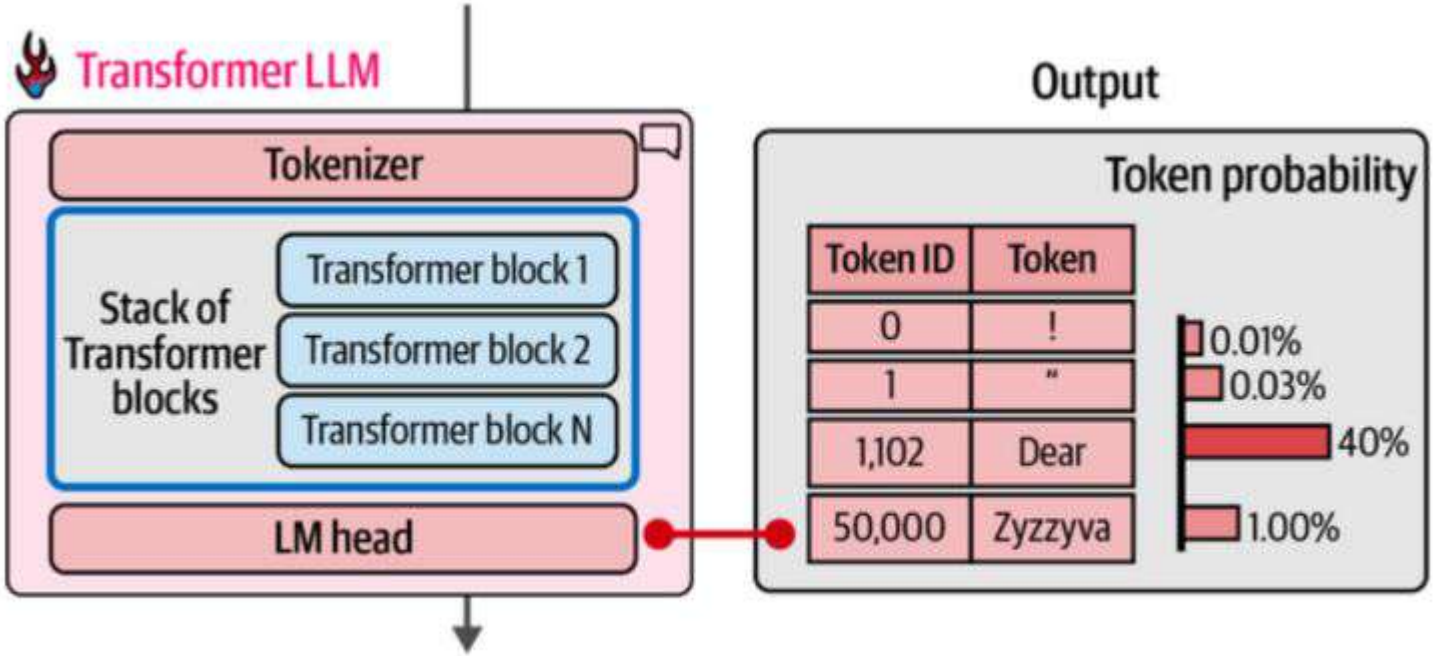


Figure 3-6. At the end of the forward pass, the model predicts a probability score for each token in the vocabulary.

Choosing a Single Token from the Probability Distribution (Sampling/Decoding)

At the end of processing, the output of the model is a probability score for each token in the vocabulary, as we saw previously in Figure 3-6. The method of choosing a single token from the probability distribution is called the decoding strategy.

The easiest decoding strategy would be to always pick the token with the highest probability score. In practice, this doesn't tend to lead to the best outputs for most use cases. A better approach is to add some randomness and sometimes choose the second or third highest probability token. The idea here is to basically sample from the probability distribution based on the probability score, as the statisticians would say.

Example - Token Probabilities

Let's look more closely at the code that demonstrates this process. In this code block, we pass the input tokens through the model, and then `lm_head`:

```
prompt = "The capital of France is"

# Tokenize the input prompt
input_ids = tokenizer(prompt, return_tensors="pt").input_ids

# Tokenize the input prompt
input_ids = input_ids.to("cuda")

# Get the output of the model before the lm_head
model_output = model.model(input_ids)

# Get the output of the lm_head
lm_head_output = model.lm_head(model_output[0])
```

Example - Token Probabilities

Now, `lm_head_output` is of the shape [1, 6, 32064]. We can access the token probability scores for the last generated token using `lm_head_output[0,-1]`, which uses the index 0 across the batch dimension; the index –1 gets us the last token in the sequence. This is now a list of probability scores for all 32,064 tokens. We can get the top scoring token ID, and then decode it to arrive at the text of the generated output token:

```
token_id = lm_head_output[0,-1].argmax(-1)
tokenizer.decode(token_id)
```

In this case this turns out to be:

```
Paris
```

Parallel Token Processing and Context Size

In text generation, we get a first glance at this when looking at how each token is processed. We know from the previous chapter that the tokenizer will break down the text into tokens. Each of these input tokens then flows through its own computation path (that's a good first intuition, at least). We can see these individual processing tracks or streams in Figure 3-8.

Current Transformer models have a limit for how many tokens they can process at once. That limit is called the model's context length. A model with 4K context length can only process 4K tokens and would only have 4K of these streams.

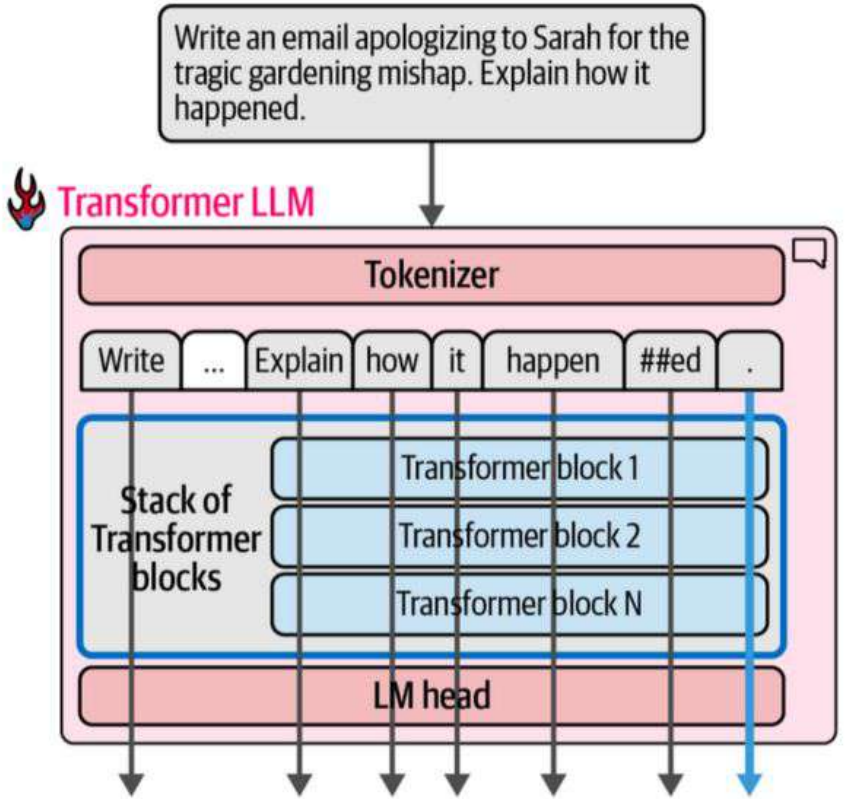


Figure 3-8. Each token is processed through its own stream of computation (with some interaction between them in attention steps, as we'll later see).

Speeding Up Generation by Caching Keys and Values

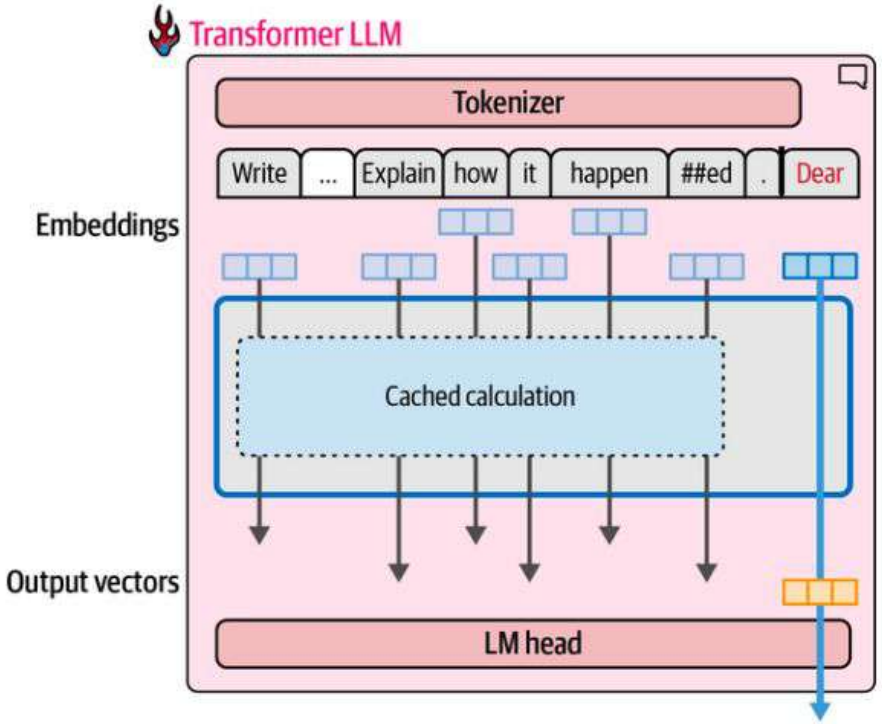


Figure 3-10. When generating text, it's important to cache the computation results of previous tokens instead of repeating the same calculation over and over again.

Inside the Transformer Block

As Figure 3-11 shows, Transformer LLMs are composed of a series Transformer blocks (often in the range of six in the original Transformer paper, to over a hundred in many large LLMs). Each block processes its inputs, then passes the results of its processing to the next block.

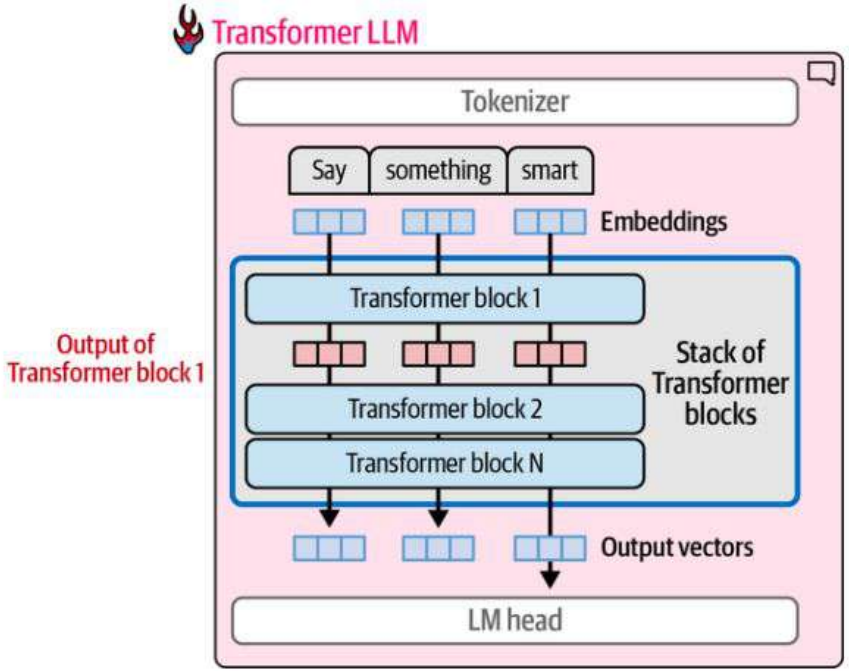


Figure 3-11. The bulk of the Transformer LLM processing happens inside a series of Transformer blocks, each handing the result of its processing as input to the subsequent block.

Each Transformer Block

A Transformer block (Figure 3-12) is made up of two successive components:

1. The attention layer is mainly concerned with incorporating relevant information from other input tokens and positions
2. The feedforward layer houses the majority of the model's processing capacity

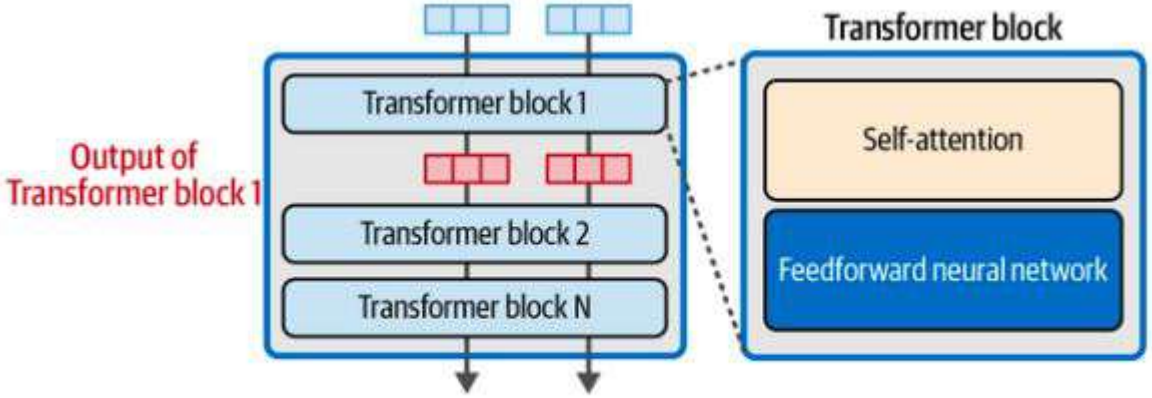


Figure 3-12. A Transformer block is made up of a self-attention layer and a feedforward neural network.

Feed Forward Neural Network

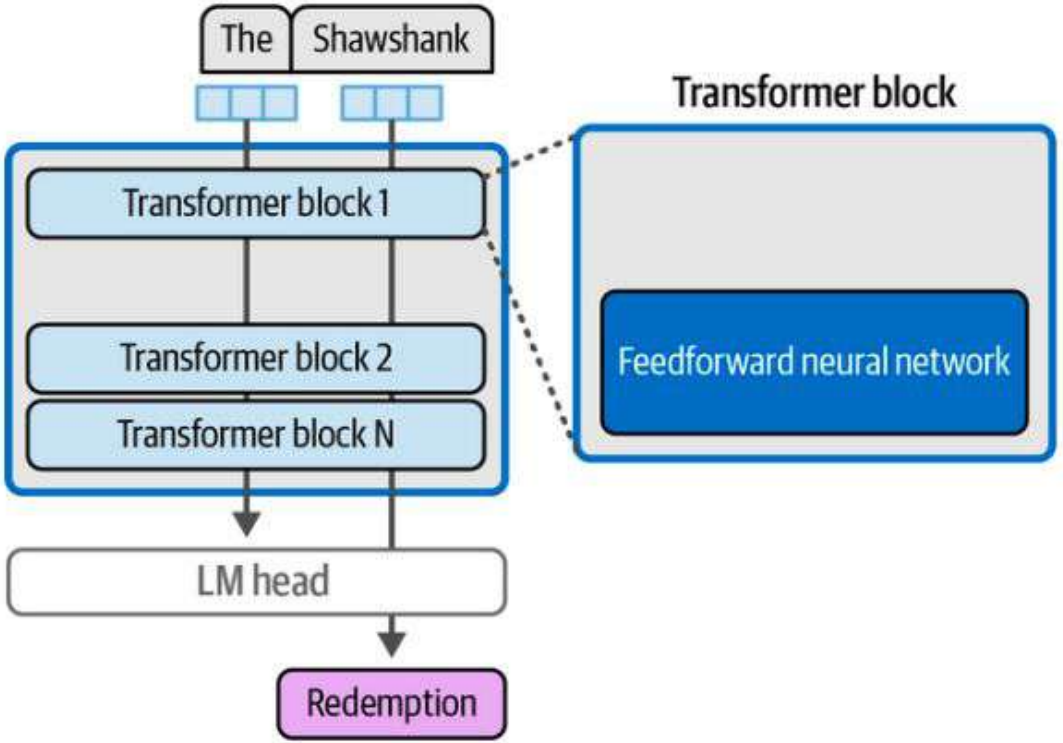


Figure 3-13. The feedforward neural network component of a Transformer block likely does the majority of the model's memorization and interpolation.

Attention Mechanism

- The Transformer architecture, introduced by Vaswani et al. (2017), marked a significant shift in natural language processing by replacing recurrent and convolutional structures with self-attention mechanisms.
- The Transformer architecture, proposed by Vaswani et al., introduces a novel approach that relies exclusively on attention mechanisms, eliminating the need for recurrence and convolutions Entirely.
- Attention is a mechanism that helps the model incorporate context as it'sprocessing a specific token. Think of the following prompt:

  "The dog chased the squirrel because **it**"

- For the model to predict what comes after "it," it needs to know what "it"refers to. Does it refer to the dog or the squirrel? In a trained Transformer LLM, the attention mechanism makes that determination. Attention adds information from the context into the representation of the "it" token. We can see a simple version of that in Figure 3-14.

Figure 3-15. It shows multiple token positions going into the attention layer; the final one is the one being currently processed (the pink arrow). The attention mechanism operates on the input vector at that position. It incorporates relevant information from the context into the vector it produces as the output for that position.

The attention layer helps the model understand each word by looking at the whole sentence and picking up useful clues from other words.

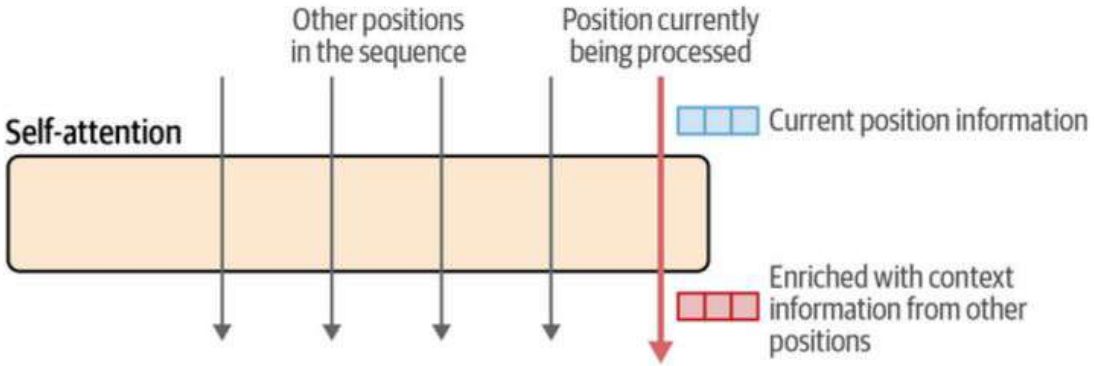


Figure 3-15. A simplified framing of attention: an input sequence and a current position being processed. As we're mainly concerned with this position, the figure shows an input vector and an output vector that incorporates information from the previous elements in the sequence according to the attention mechanism.

Two main steps are involved in the attention mechanism:

- A way to score how relevant each of the previous input tokens are to the current token being processed (in the pink arrow).
- Using those scores, we combine the information from the various positions into a single output vector.

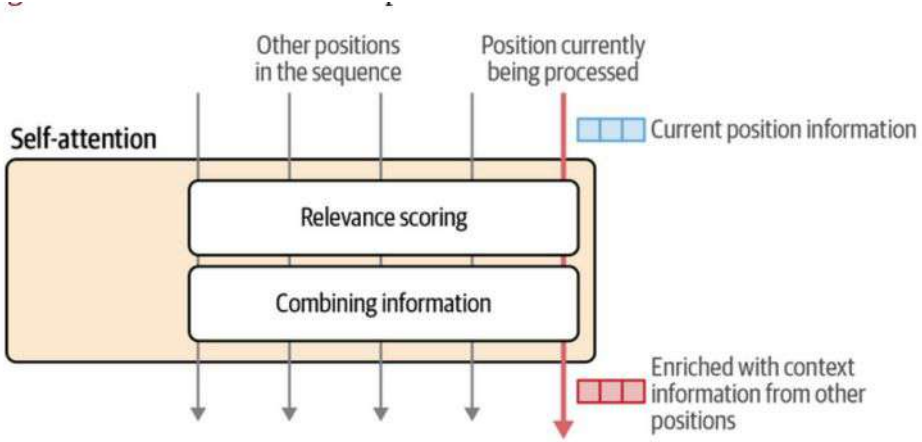


Figure 3-16. Attention is made up of two major steps: relevance scoring for each position, then a step where we combine the information based on those scores.

How attention is calculated

The training process produces three projection matrices that produce the components that interact in this calculation:

- A query projection matrix (Q)
- A key projection matrix (K)
- A value projection matrix (V)

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

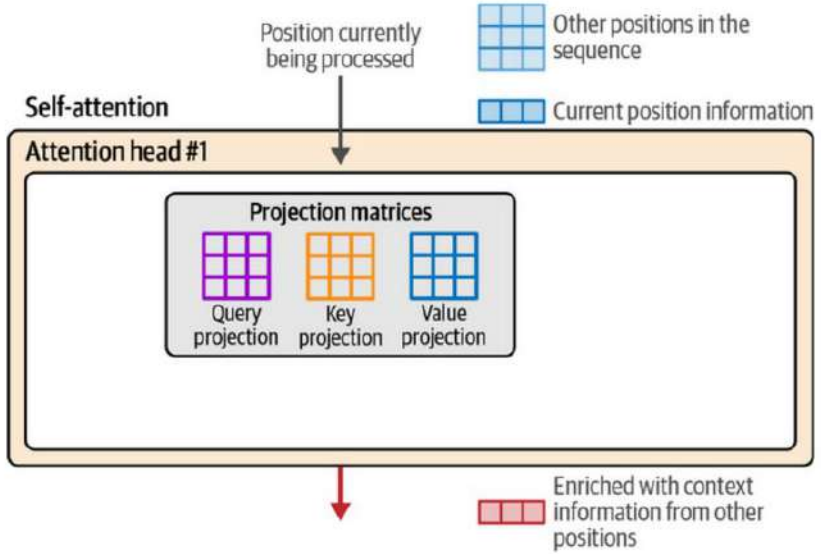


Figure 3-18. Before starting the self-attention calculation, we have the inputs to the layer and projection matrices for queries, keys, and values.

Attention starts by multiplying the inputs by the projection matrices to create three new matrices. These are called the queries, keys, and values matrices.

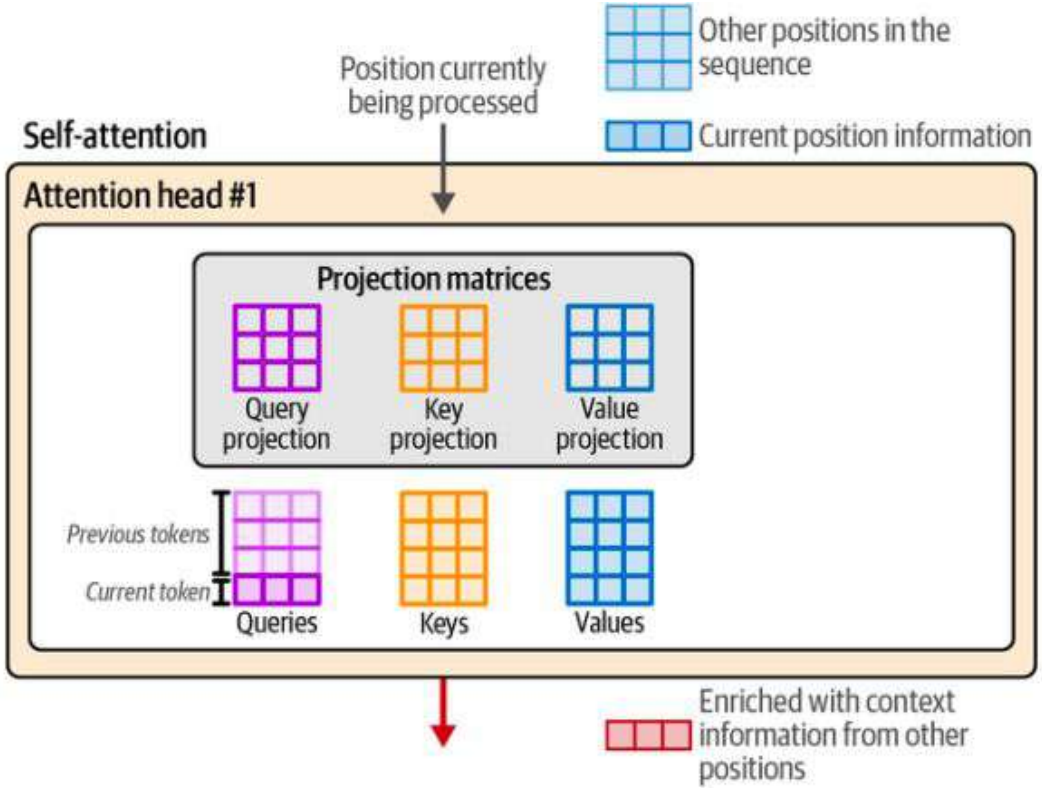


Figure 3-19. Attention is carried out by the interaction of the queries, keys, and values matrices. Those are produced by multiplying the layer's inputs with the projection matrices.

Relation between Q, K, V

Each token generates a **query** vector that represents what it is “looking for” in other tokens. This vector is used to compute similarity with the **key** vectors of all tokens in the sequence.

Let’s say Q_sat = [1, 0]

This means "sat" is looking for words that are strong in **feature 1**.

Now we compare Q_sat to the **K vectors of all other words**:

Word	K vector	Dot product with Q_sat
The	[1, 0]	1×1 + 0×0 = 1 ✅
cat	[0, 1]	1×0 + 0×1 = 0 ❌
mat	[1, 1]	1×1 + 0×1 = 1 ✅

So "sat" pays attention to "The" and "mat", but not "cat".

Then it uses the **V vectors** of those words to build a new meaning for "sat".

Softmax Function

The **softmax function** transforms a vector of raw scores into a probability distribution. It is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

This ensures that all output values lie between 0 and 1 and sum to 1. In the Transformer architecture, softmax is used to compute attention weights from similarity scores and to generate token probabilities in the final output layer.

Example: Consider the vector of scores:

$$\mathbf{z} = [2.0,\ 1.0,\ 0.1]$$

First, exponentiate each element:

$$e^{2.0} \approx 7.39, \quad e^{1.0} \approx 2.72, \quad e^{0.1} \approx 1.11$$

Next, compute the sum:

$$\sum e^{z_j} = 7.39 + 2.72 + 1.11 = 11.22$$

Now, apply softmax:

$$\text{softmax}(2.0) = \frac{7.39}{11.22} \approx 0.659, \quad \text{softmax}(1.0) = \frac{2.72}{11.22} \approx 0.242, \quad \text{softmax}(0.1) = \frac{1.11}{11.22} \approx 0.099$$

Thus, the output is:

$$[0.659,\ 0.242,\ 0.099]$$

This vector represents a probability distribution over the input scores.

Example - Calculating the Attention Scores

In the attention mechanism, each token computes its relevance to other tokens using the dot product between its query vector and the key vectors of all tokens. These raw scores are then scaled and normalized to produce attention weights.

Let dk be the dimension of the key vectors. To prevent extremely large dot products that could destabilize training, we scale the raw scores by √dk. The scaled scores are then passed through a softmax function to obtain the final attention weights.

In this example, we compute the attention weights for the token “crossed” in the sentence “The chicken crossed the road.” The goal is to determine how much “crossed” attends to each token in the sequence

Assume each token is projected into a 4-dimensional space, so $d_k = 4$ and the scaling factor is $\sqrt{4} = 2$. Let the query vector for "crossed" be:

$$Q_{\text{crossed}} = [1, 2, 1, 0]$$

And the key vectors for the tokens in the sentence "The chicken crossed the road" are:

$$\begin{aligned} K_{\text{The}} &= [1, 1, 0, 0] \\ K_{\text{Chicken}} &= [2, 1, 1, 0] \\ K_{\text{Crossed}} &= [1, 2, 1, 0] \\ K_{\text{The}} &= [1, 1, 0, 0] \\ K_{\text{Road}} &= [1, 1, 1, 0] \end{aligned}$$

We compute the dot product between Q_{crossed} and each K_i:

$$\begin{aligned} Q \cdot K_{\text{The}} &= 1 \times 1 + 2 \times 1 + 1 \times 0 + 0 \times 0 = 3 \\ Q \cdot K_{\text{Chicken}} &= 1 \times 2 + 2 \times 1 + 1 \times 1 + 0 \times 0 = 5 \\ Q \cdot K_{\text{Crossed}} &= 1 \times 1 + 2 \times 2 + 1 \times 1 + 0 \times 0 = 6 \\ Q \cdot K_{\text{Road}} &= 1 \times 1 + 2 \times 1 + 1 \times 1 + 0 \times 0 = 4 \end{aligned}$$

Then we scale each score by dividing by $\sqrt{d_k} = 2$:

$$\text{Scaled}_{\text{The}} = \frac{3}{2} = 1.5$$

$$\text{Scaled}_{\text{Chicken}} = \frac{5}{2} = 2.5$$

$$\text{Scaled}_{\text{Crossed}} = \frac{6}{2} = 3.0$$

$$\text{Scaled}_{\text{Road}} = \frac{4}{2} = 2.0$$

Apply softmax to these scaled scores:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Let's compute:

$$
\begin{aligned}
e^{1.5} &\approx 4.48 \\
e^{2.5} &\approx 12.18 \\
e^{3.0} &\approx 20.09 \\
e^{2.0} &\approx 7.39 \\
\text{Sum} &= 4.48 + 12.18 + 20.09 + 7.39 = 44.14
\end{aligned}
$$

Final attention weights:

$$
\begin{aligned}
\alpha_{\text{The}} &= \frac{4.48}{44.14} \approx 0.10 \\
\alpha_{\text{Chicken}} &= \frac{12.18}{44.14} \approx 0.28 \\
\alpha_{\text{Crossed}} &= \frac{20.09}{44.14} \approx 0.46 \\
\alpha_{\text{Road}} &= \frac{7.39}{44.14} \approx 0.17
\end{aligned}
$$

These weights determine how much "crossed" attends to each token. The highest attention is to itself, followed by "chicken" and "road."

Positional Embeddings (RoPE)

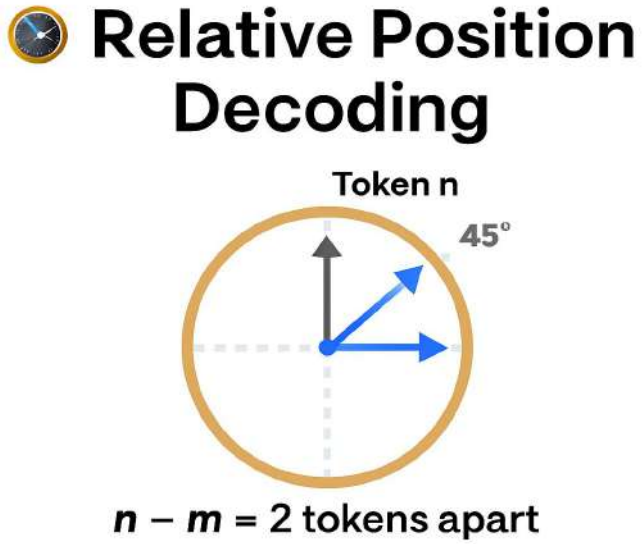
- Positional embeddings have been a key component since the original Transformer. They enable the model to keep track of the order of tokens/words in a sequence/sentence, which is an indispensable source of information in language.
- From the many positional encoding schemes proposed in the past years, rotary positional embeddings (or “RoPE,” introduced in “RoFormer: Enhanced Transformer with rotary position embedding”) is especially important to point out.
- Transformers treat input tokens as unordered. Without positional cues, “The cat chased the mouse” and “The mouse chased the cat” look the same. *Positional embeddings* give each token a sense of “where” it is in the sequence.
- **RoPE = Rotary Positional Embedding.** Instead of adding position vectors (like sinusoidal embeddings), RoPE *rotates* the query and key vectors in attention using trigonometric functions.
- Each token’s vector is rotated by an angle based on its position. The dot product between rotated vectors encodes *relative position*.

RoPE Encodes Position Like a Clock

Each token's vector rotates like a clock hand based on its position in the sequence. This rotation encodes *relative position*, which helps the Transformer understand how far apart tokens are.

Decoding in RoPE

- Each token’s vector is rotated based on its position.
- During decoding, the model compares these rotated vectors to compute attention scores.
- The angle between vectors encodes how far apart tokens are.



Text Clustering and Topic Modeling

Text clustering

Text clustering aims to group similar texts based on their semantic content, meaning, and relationships.

As illustrated in Figure 5-1, the resulting clusters of semantically similar documents not only facilitate efficient categorization of large volumes of unstructured text but also allow for quick exploratory data analysis.

Language is more than a bag of words, and recent language models have proved to be quite capable of capturing that notion.

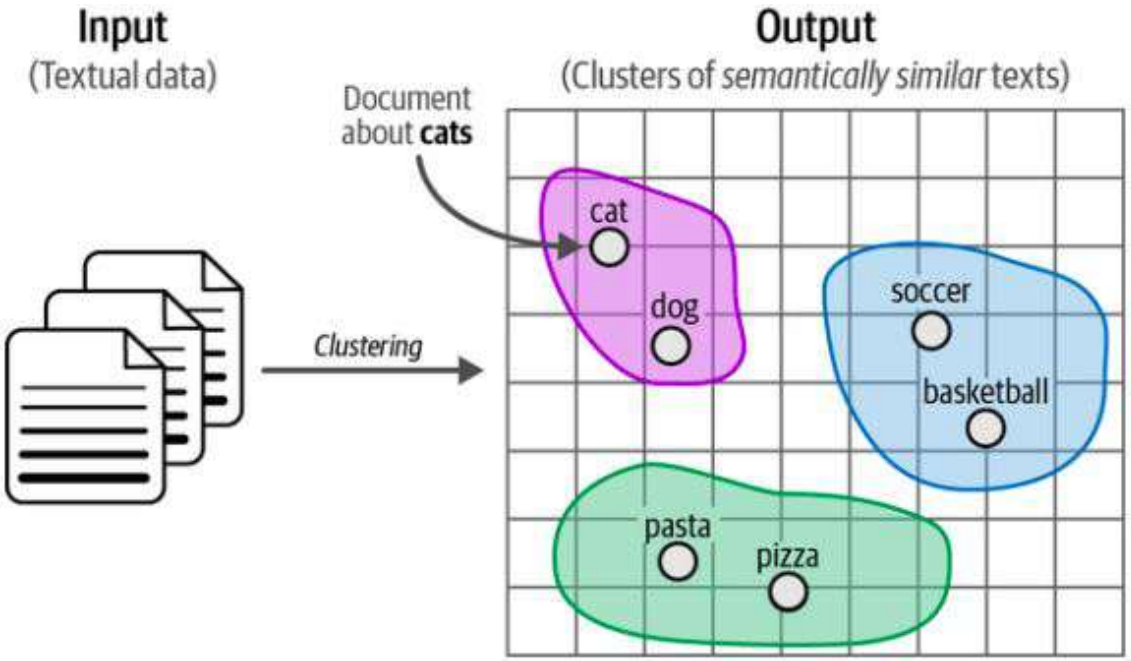


Figure 5-1. Clustering unstructured textual data.

Topic Modelling

Text clustering has also found itself in the realm of topic modeling, where we want to discover (abstract) topics that appear in large collections of textual data. As shown in Figure 5-2, we generally describe a topic using keywords or keyphrases and, ideally, have a single overarching label.

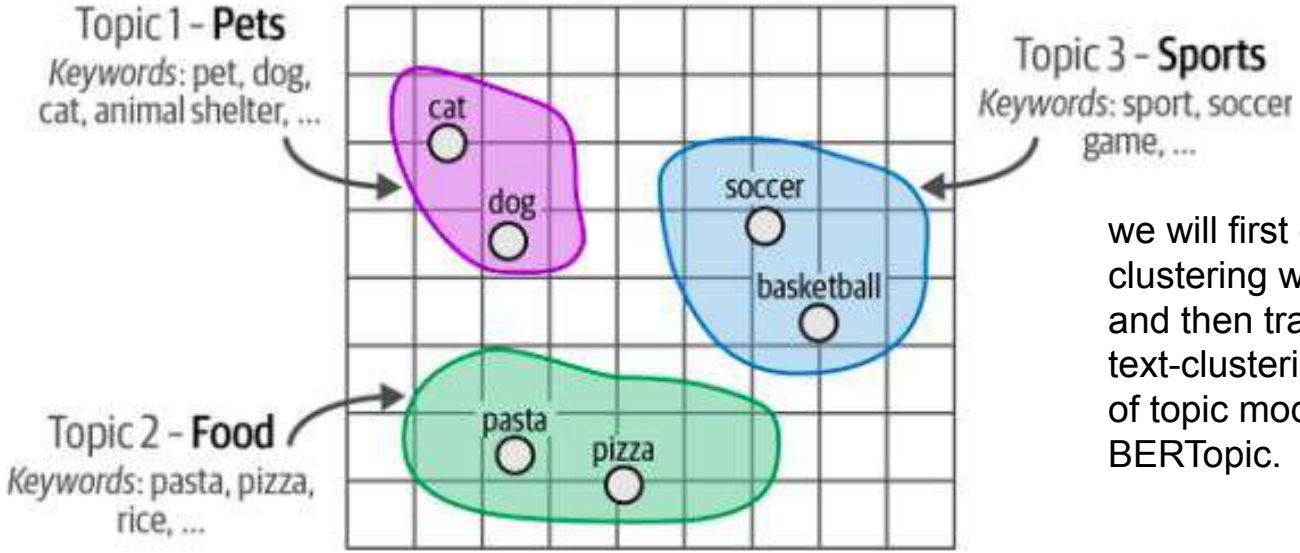


Figure 5-2. Topic modeling is a way to give meaning to clusters of textual documents.

we will first explore how to perform clustering with embedding models and then transition to a text-clustering-inspired method of topic modeling, namely BERTopic.

A Common Pipeline for Text Clustering

Although there are many methods for text clustering, from graph-based neural networks to centroid-based clustering techniques, a common pipeline that has gained popularity involves three steps and algorithms:

1. Convert the input documents to embeddings with an embedding model.
2. Reduce the dimensionality of embeddings with a dimensionality reduction model.
3. Find groups of semantically similar documents with a cluster model.

Step 1 - Embedding Documents

The first step is to convert our textual data to embeddings, as illustrated in Figure 5-3. Recall from previous chapters that embeddings are numerical representations of text that attempt to capture its meaning.

most embedding models at the time of writing focus on just that, semantic similarity.

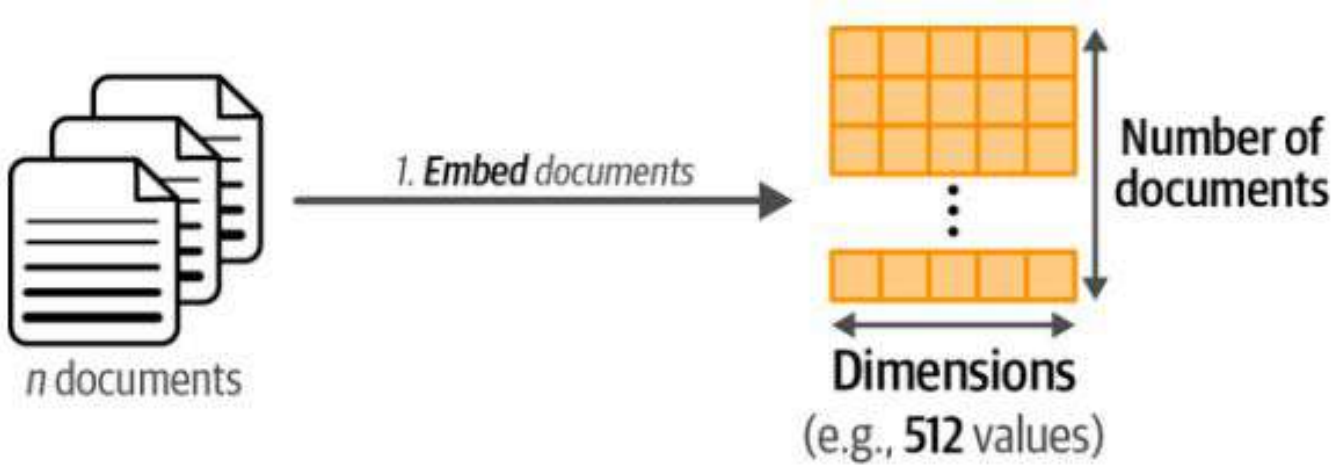


Figure 5-3. Step 1: We convert documents to embeddings using an embedding model.

the “**thenlper/gte-small**” model instead. It is a more recent model that outperforms the previous model on clustering tasks and due to its small size is even faster for inference.

Reducing the Dimensionality of Embeddings

As the number of dimensions increases, there is an exponential growth in the number of possible values within each dimension. Finding all subspaces within each dimension becomes increasingly complex.

we can make use of dimensionality reduction. As illustrated in Figure 5-4, this technique allows us to reduce the size of the dimensional space and represent the same data with fewer dimensions.

Dimensionality reduction techniques aim to preserve the global structure of high-dimensional data by finding low-dimensional representations.

Step 2 - Reducing the Dimensionality of Embeddings

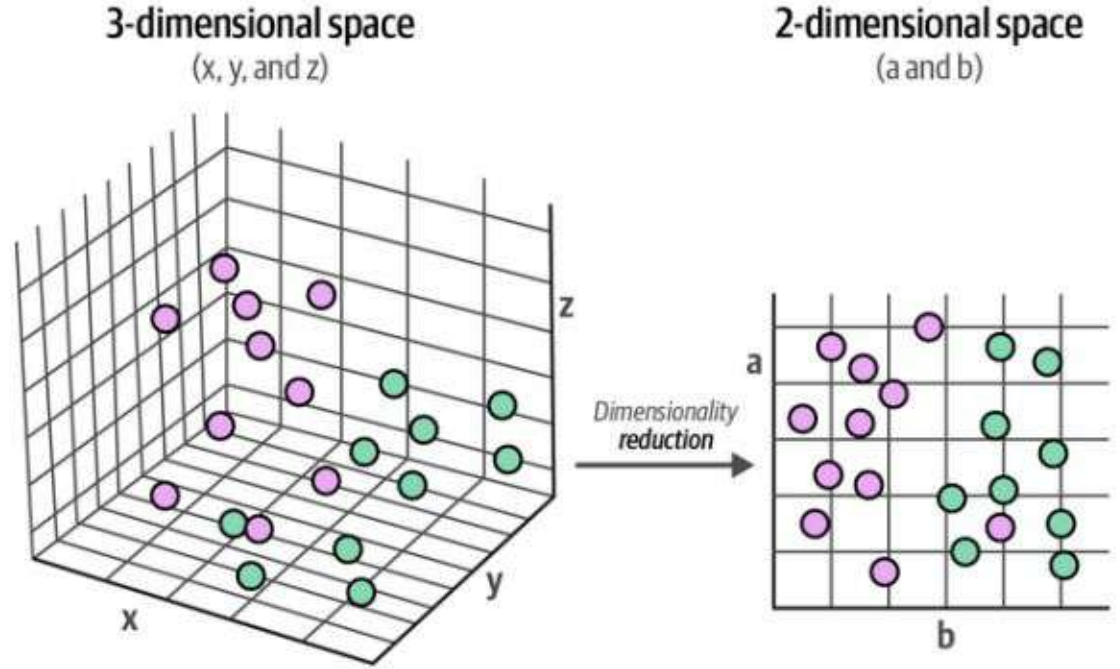


Figure 5-4. Dimensionality reduction allows data in high-dimensional space to be compressed to a lower-dimensional representation.

Well-known methods for dimensionality reduction are Principal Component Analysis (PCA) and Uniform Manifold Approximation and Projection (UMAP).

Step 3 - Cluster the Reduced Embeddings

The third step is to cluster the reduced embeddings, as illustrated in Figure 5-6.

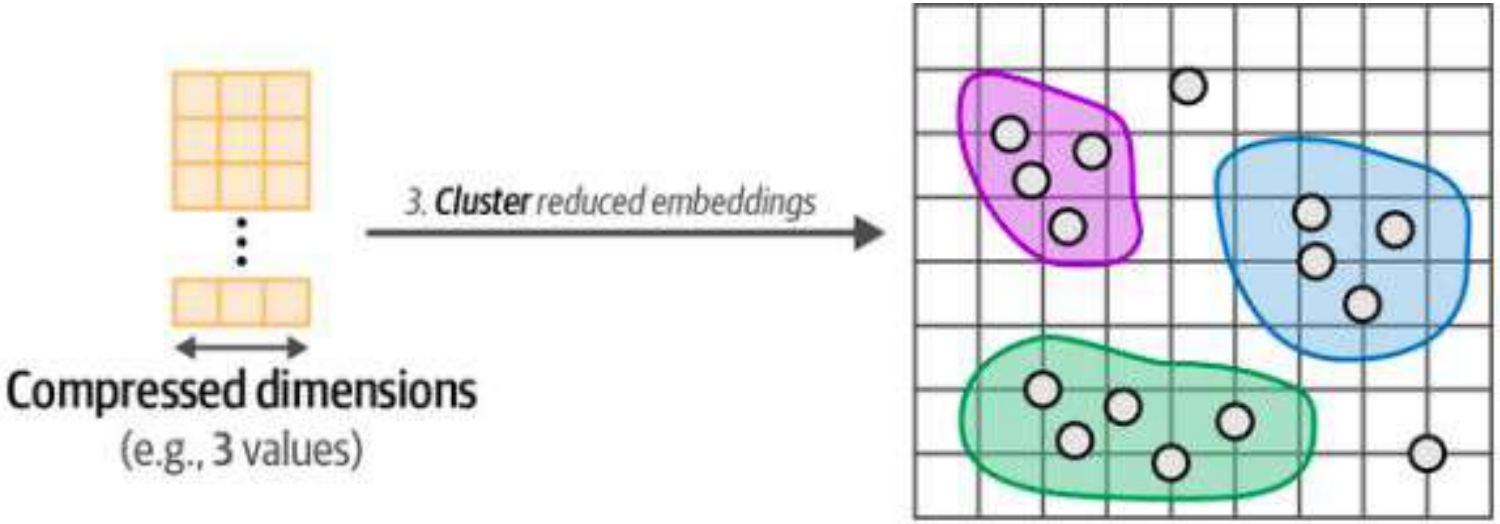


Figure 5-6. Step 3: We cluster the documents using the embeddings with reduced dimensionality.

Step 3 - Cluster the Reduced Embeddings

Although a common choice is a centroid-based algorithm like k-means, which requires a set of clusters to be generated, we do not know the number of clusters beforehand. Instead, a density-based algorithm freely calculates the number of clusters and does not force all data points to be part of a cluster, as illustrated in Figure 5-7.

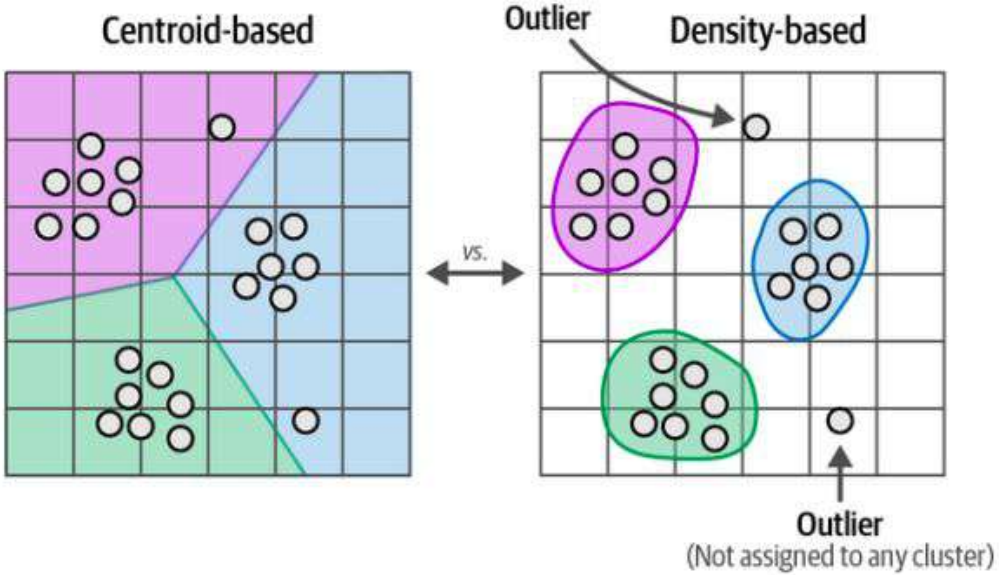


Figure 5-7. The clustering algorithm not only impacts how clusters are generated but also how they are viewed.

As a density-based method, HDBSCAN can also detect outliers in the data, which are data points that do not belong to any cluster. These outliers will not be assigned or forced to belong to any cluster.

From Text Clustering to Topic Modeling

This idea of finding themes or latent topics in a collection of textual data is often referred to as topic modeling. Traditionally, it involves finding a set of keywords or phrases that best represent and capture the meaning of the topic, as we illustrate in Figure 5-9.

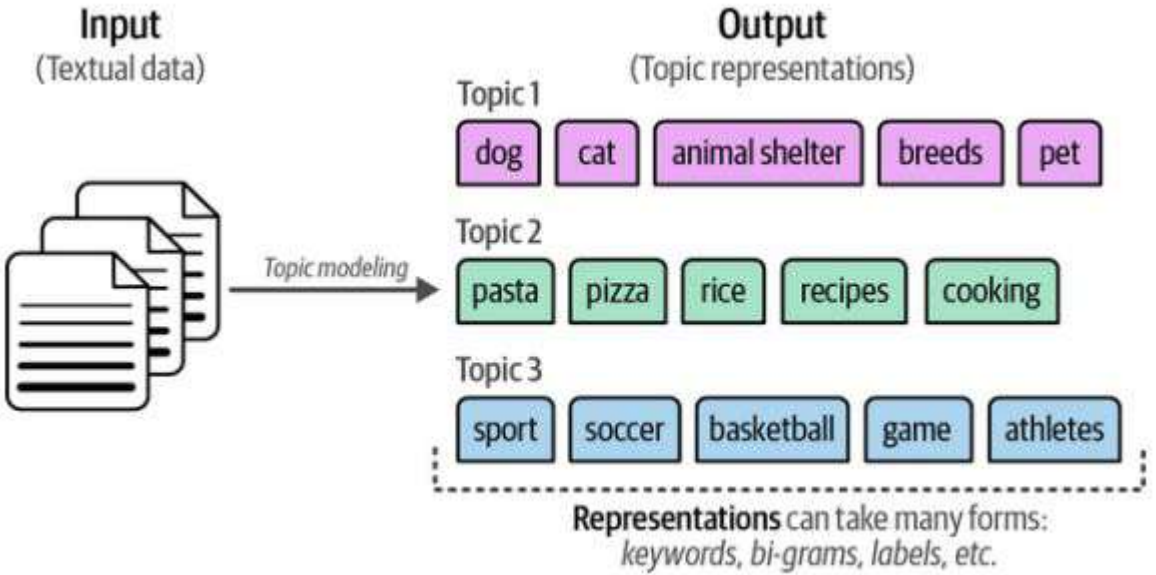


Figure 5-9. Traditionally, topics are represented by a number of keywords but can take other forms.

Traditional drichlet technology

Classic approaches, like latent Dirichlet allocation, assume that each topic is characterized by a probability distribution of words in a corpus's Vocabulary. 5 Figure 5-10 demonstrates how each word in a vocabulary is scored against its relevance to each topic.

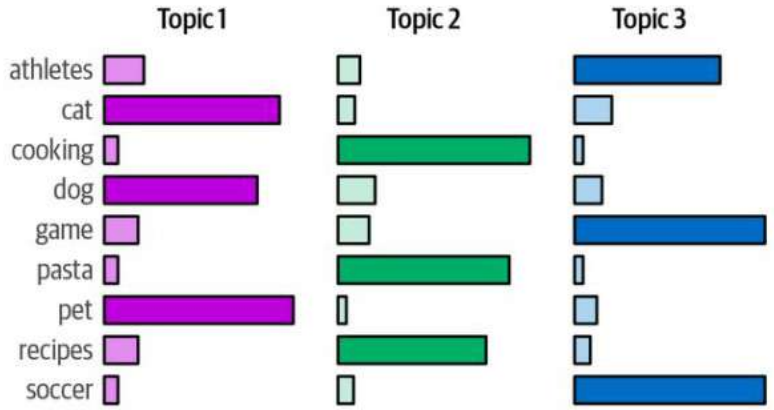


Figure 5-10. Keywords are extracted based on their distribution over a single topic.

BERTopic: A Modular Topic Modeling Framework

There are two steps in the Pipeline for BERTopic. First Step, as illustrated in Figure 5-11, we follow the same procedure as we did before in our text clustering example. We embed documents, reduce their dimensionality, and finally cluster the reduced embedding to create groups
of semantically similar documents.

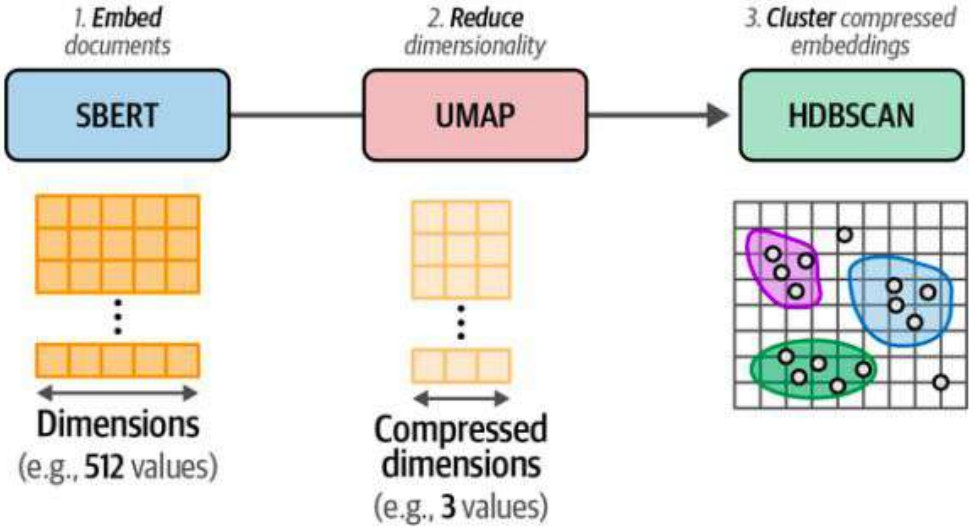


Figure 5-11. The first part of BERTopic's pipeline is to create clusters of semantically similar documents.

BERTopic: A Modular Topic Modeling Framework

Second, it models a distribution over words in the corpus's vocabulary by leveraging a classic method, namely bag-of-words. The bag-of-words, as we discussed briefly in Chapter 1 and illustrate in Figure 5-12, does exactly what its name implies, counting the number of times each word appears in a document. The resulting representation could be used to extract the most frequent words inside a document.

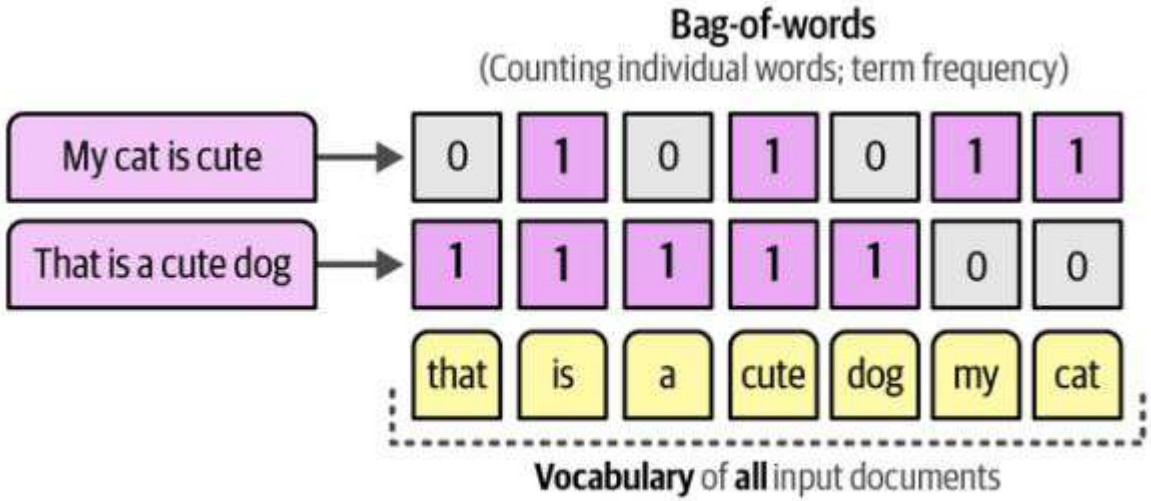


Figure 5-12. A bag-of-words counts the number of times each word appears inside a document.

From TF to c-TF

First, this is a representation on a document level and we are interested in a cluster-level perspective. To address this, the frequency of words is calculated within the entire cluster instead of only the document, as illustrated in Figure 5-13.

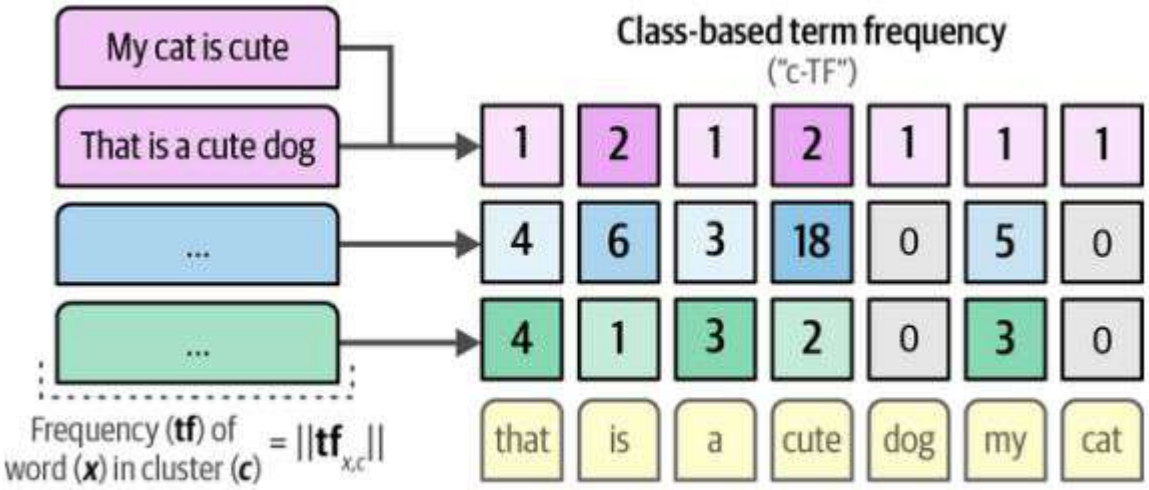


Figure 5-13. Generating c-TF by counting the frequency of words per cluster instead of per document.

Inverse Document Frequency – IDF

Second, stop words like “the” and “I” tend to appear often in documentsand provide little meaning to the actual documents. BERTopic uses a class-based variant of term frequency–inverse document frequency (c-TF-IDF) to put more weight on words that are more meaningful to a cluster and put less weight on words that are used across all clusters.

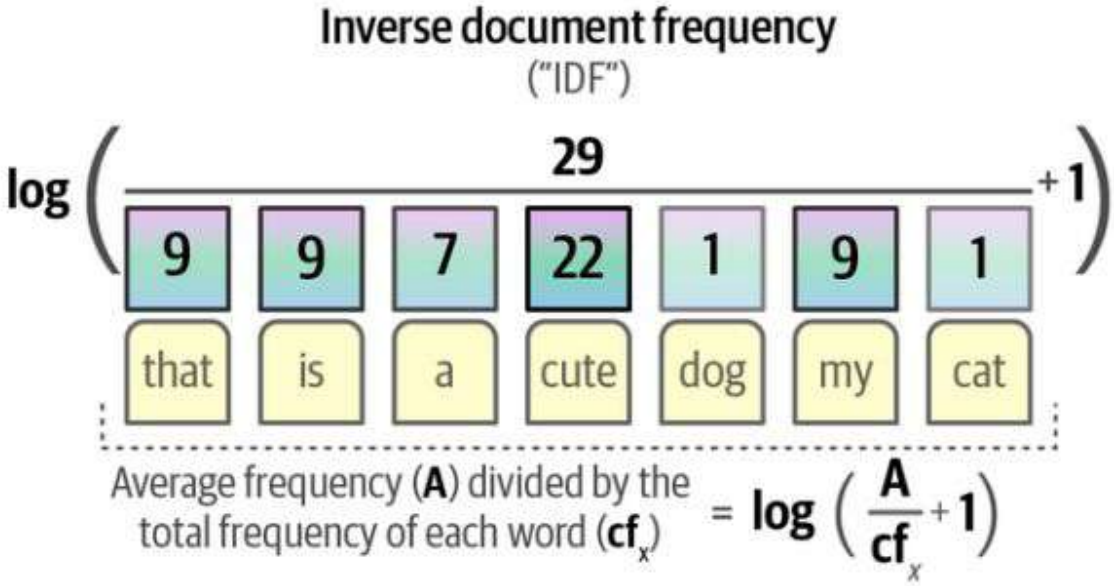


Figure 5-14. Creating a weighting scheme.

The calculation of the weight of term *x* in a class *c*.

This second part of the procedure, as shown in Figure 5-15, allows for generating a distribution over words as we have seen before. We can use
scikit-learn's CountVectorizer to generate the bag-of-words (or term frequency) representation. Here, each cluster is considered a topic that has a
specific ranking of the corpus's vocabulary.

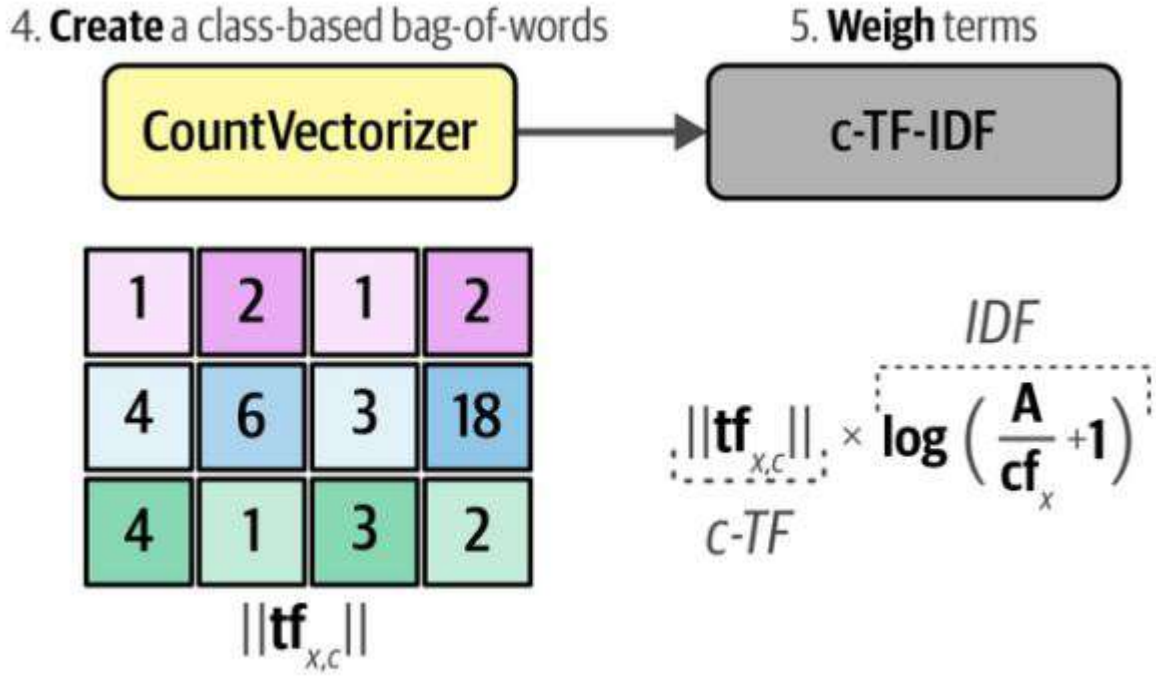


*Figure 5-15. The second part of BERTopic's pipeline is representing the topics: the calculation of the weight of term *x* in a class *c*.*

Final Pipeline for BERTopic

A major advantage of this pipeline is that the two steps, clustering and topic representation, are largely independent of one another. For instance, with c-TF-IDF, we are not dependent on the models used in clustering the documents. This allows for significant modularity throughout every component of the pipeline.

Putting the two steps together, clustering and representing topics, results in the full pipeline of BERTopic, as illustrated in Figure 5-16. With this pipeline, we can cluster semantically similar documents and from the clusters generate topics represented by several keywords. The higher a word's weight in a topic, the more representative it is of that topic.

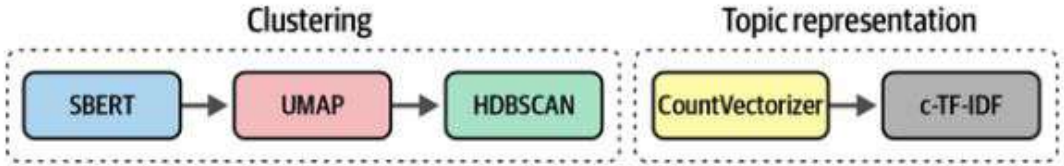


Figure 5-16. The full pipeline of BERTopic, roughly, consists of two steps, clustering and topic representation.

Using Generative Models for Text Clustering

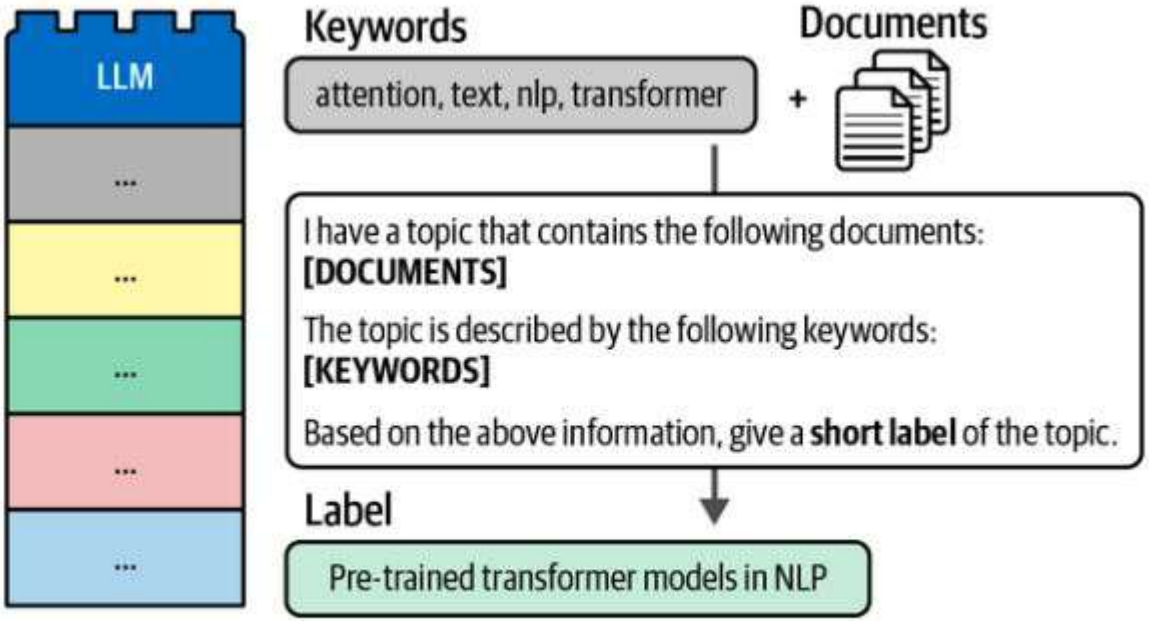


Figure 5-23. Use text generative LLMs and prompt engineering to create labels for topics from keywords and documents related to each topic.

Chapter 4. Text Classification

Text Classification

A common task in natural language processing is classification. The goal of the task is to train a model to assign a label or class to some input text (see Figure 4-1). Classifying text is used across the world for a wide range of applications, from sentiment analysis and intent detection to extracting entities and detecting language.

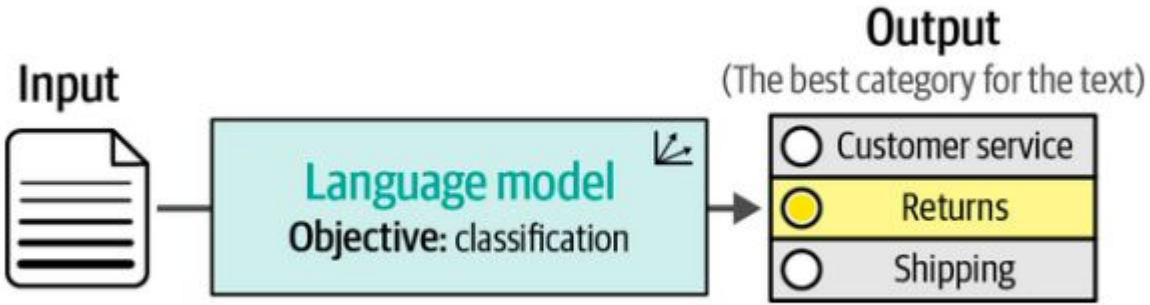


Figure 4-1. Using a language model to classify text.

used for classifying text. As illustrated in Figure 4-2, we will examine both representation and language models and explore their differences.

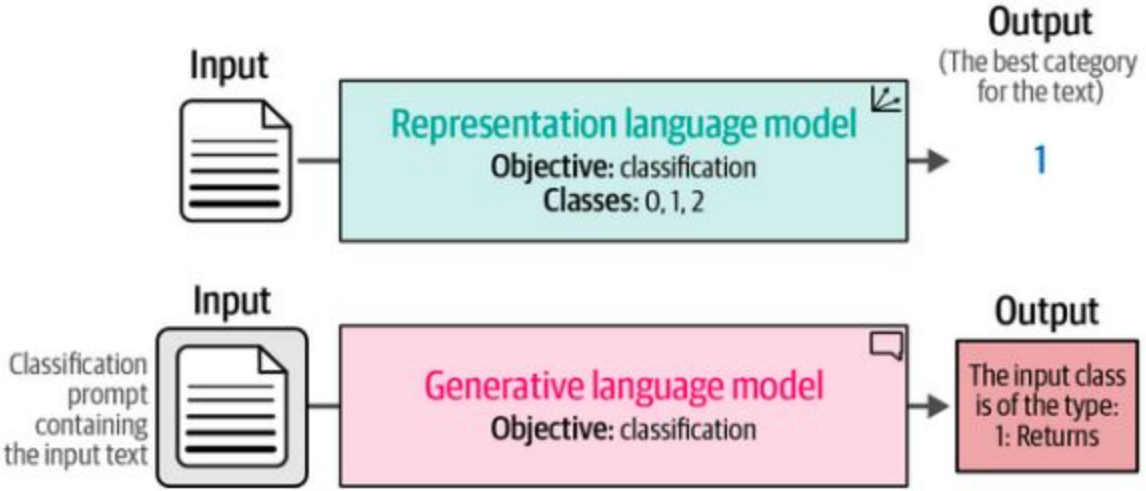


Figure 4-2. Although both representation and generative models can be used for classification, their approaches differ

A major benefit of these tokens is that they can be combined to generate representations even if they were not in the training data, as shown in Figure 4-7.

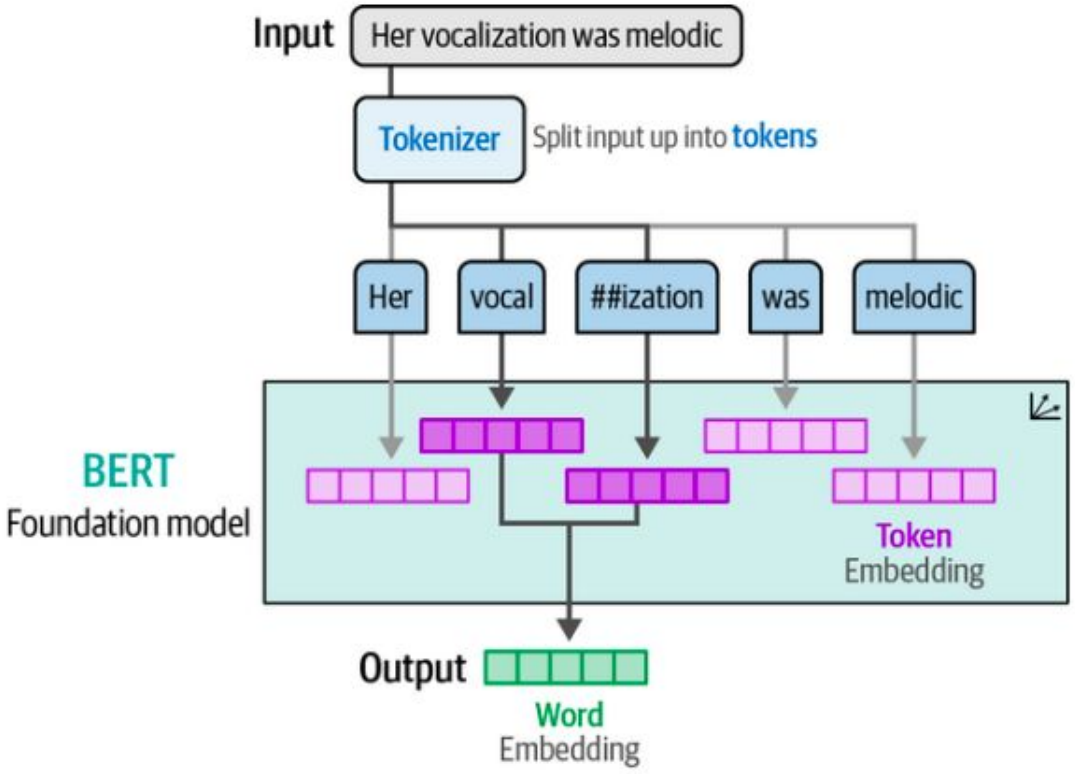


Figure 4-7. By breaking down an unknown word into tokens, word embeddings can still be generated.

Metrics

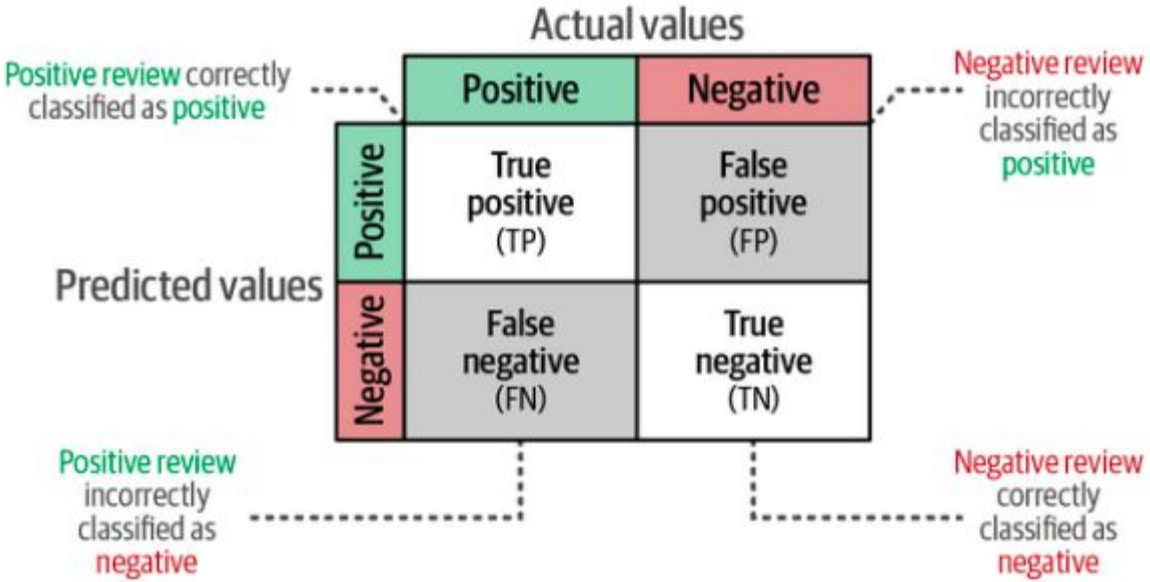


Figure 4-8. The confusion matrix describes four types of predictions we can make.

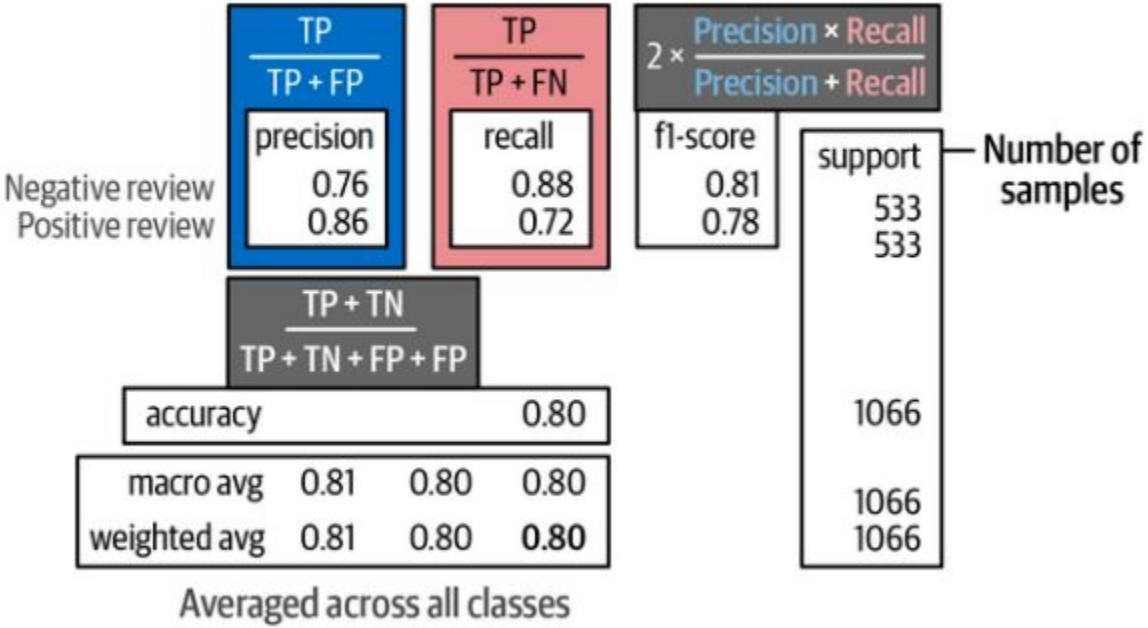


Figure 4-9. The classification report describes several metrics for evaluating a model's performance.

Using the confusion matrix, we can derive several formulas to describe the quality of the model. In the previously generated classification report we can see four such methods, namely precision, recall, accuracy, and the F1 score:

- Precision measures how many of the items found are relevant, which indicates the accuracy of the relevant results.
- Recall refers to how many relevant classes were found, which indicates its ability to find all relevant results.
- Accuracy refers to how many correct predictions the model makes out of all predictions, which indicates the overall correctness of the model.
- The F1 score balances both precision and recall to create a model's overall performance.

Text Classification with Representation Models

Classification with pretrained representation models generally comes in two flavors, either using a task-specific model or an embedding model. As we explored in the previous chapter, these models are created by fine-tuning a foundation model, like BERT, on a specific downstream task as illustrated in Figure 4-3.

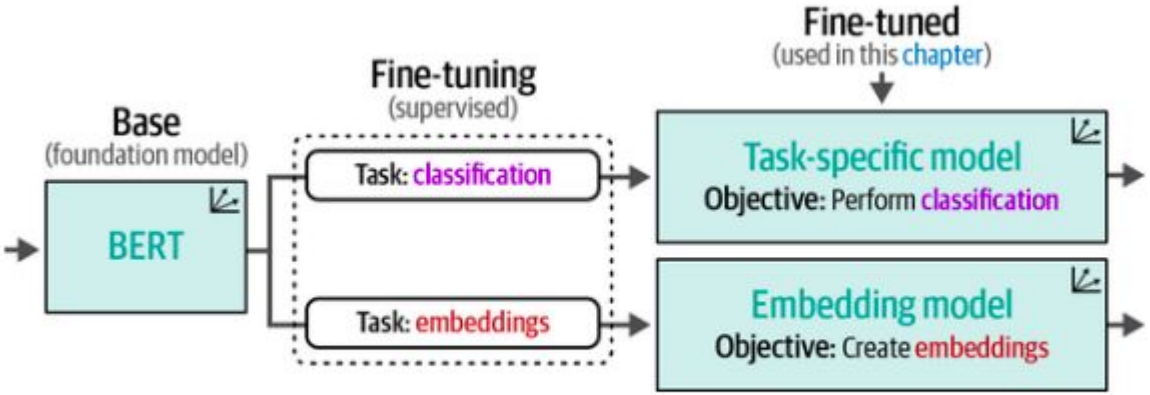


Figure 4-3. A foundation model is fine-tuned for specific tasks; for instance, to perform classification or generate general-purpose embeddings.

As we load our model, we also load the tokenizer, which is responsible for
converting input text into individual tokens, as illustrated in Figure 4-6.

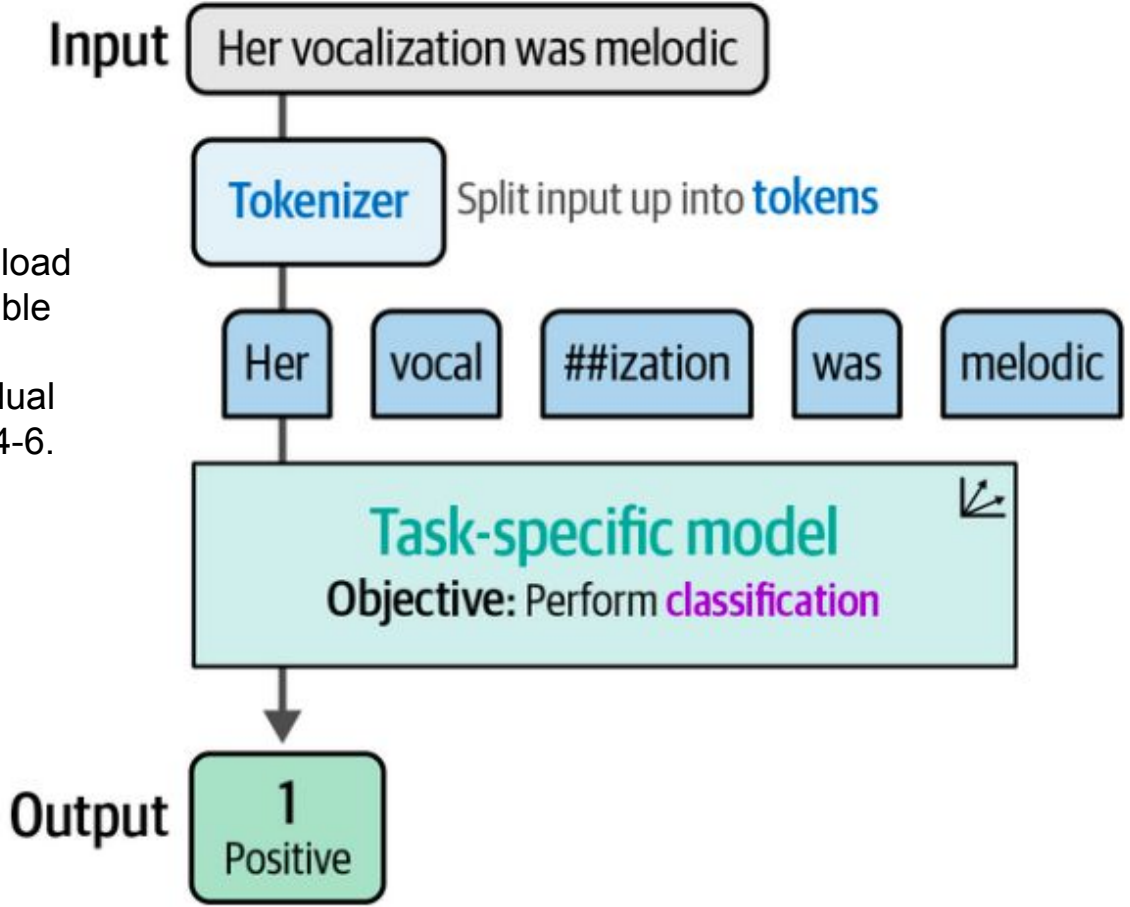


Figure 4-6. An input sentence is first fed to a tokenizer before it can be processed by the task-specific model.

Task -specific model

A **task-specific** model is typically something like "**bert-base-uncased-finetuned-SST-2**," which is BERT fine-tuned on the Stanford Sentiment Treebank (SST-2) dataset. The final layer is designed to output a probability distribution over classes (positive or negative).

How It Works

1. **Model Input**: Text is tokenized into subword units.
2. **Model Forward Pass**: The model processes these tokens.
3. **Classifier Head**: The model has a classification layer that outputs probabilities, e.g., `[0.2 (negative), 0.8 (positive)]`.
4. **Prediction**: The class with the highest probability is chosen.

Embedding Model

Now, suppose you can't find a task-specific model that perfectly suits your domain. Or maybe you want more control over your final classifier. In that case, you can take the approach of generating **embeddings** and then training a lightweight classifier, such as **Logistic Regression** or **Random Forest**, on top of those embeddings.

Step-by-Step

1. **Load an Embedding Model**: For example, `sentence-transformers/all-mpnet-base-v2`
2. **Convert Text to Embeddings**: The model generates a vector for each piece of text.
3. **Train a Classifier**: Feed these vectors into a standard classifier (e.g., Logistic Regression).
4. **Predict**: For new input, create the embedding and then run it through the classifier.

Zero-Shot Classification with Embeddings

Imagine you have a dataset but **no labels** at all. You just know the general categories you'd like to assign (e.g., "positive" vs. "negative"). **Zero-shot classification** steps in to save the day.

The "Trick" with Embeddings

1. **Create a text description of each label** (e.g., "This is a positive movie review").
2. **Embed the label descriptions** and your documents using the same embedding model.
3. **Compute the Cosine Similarity** between each document's embedding and each label's embedding.
4. **Pick the Label** with the highest similarity score.

Example

Label descriptions:

- “A very positive movie review”
- “A very negative movie review”
- Text to classify: “This movie was an absolute masterpiece!”
- Compute similarity with each label. Whichever is higher gets assigned.

This approach can yield surprisingly strong performance. Even though no labeled training data is used, the **richness** of the embedding space can accurately capture sentiment differences.

Generative Models for Text Classification

Generative models, such as **GPT-3.5**, **ChatGPT**, **Flan-T5**, and the entire GPT family, are **sequence-to-sequence** models. They take text in and produce text out.

Using Flan-T5 for Classification

1. **Load the Model**: For example, `google/flan-t5-small`.
2. **Create a Prompt**: “Is the following sentence positive or negative? ”
3. **Generate Output**: The model’s textual output might be the word “positive” or “negative.”
4. **Convert Output to a Label**: Map “positive” → `1`, “negative” → `0`.

ChatGPT: Using a Closed Source Model via API

ChatGPT, powered by the GPT-3.5 or GPT-4 family, is accessed through OpenAI's API. You can't download it to your own servers. Instead, you send text to the API, and it sends back a response.

Setting Up

1. **Create an OpenAI Account** (the free tier often covers many requests).
2. **Retrieve Your API Key**.
3. **Send Requests**: Typically, in a chat style format with roles: "system", "user", "assistant."

Example: Sentiment Classification

Prompt:

```
Predict whether the following document is a positive or negative movie review:

[DOCUMENT]

If it is positive return 1 and if it is negative return 0. Do not give any other answers.
```

- **Document**: “The acting was superb, and I laughed the whole time.”
- **Expected output**: “1” (because it’s positive).

Conclusion

Text classification is an incredibly powerful tool in NLP — and modern language models make it more accessible and accurate than ever before.

- **Representation models** (like BERT or RoBERTa) do a great job converting text into structured embeddings that capture semantic meaning.
- **Task-specific models**: Already fine-tuned for, say, sentiment analysis. Easy to use out of the box.
- **Embedding models**: Generate general-purpose embeddings. You can train a simple classifier on top or do **zero-shot** labeling by comparing embedding similarities.
- **Generative models** (like Flan-T5 or ChatGPT) output text but can be coaxed into classification by prompting.
- They offer **flexibility** and **strong performance** even without fine-tuning.
- They do, however, require mindful **prompt engineering** and can be costlier for large-scale inference.