

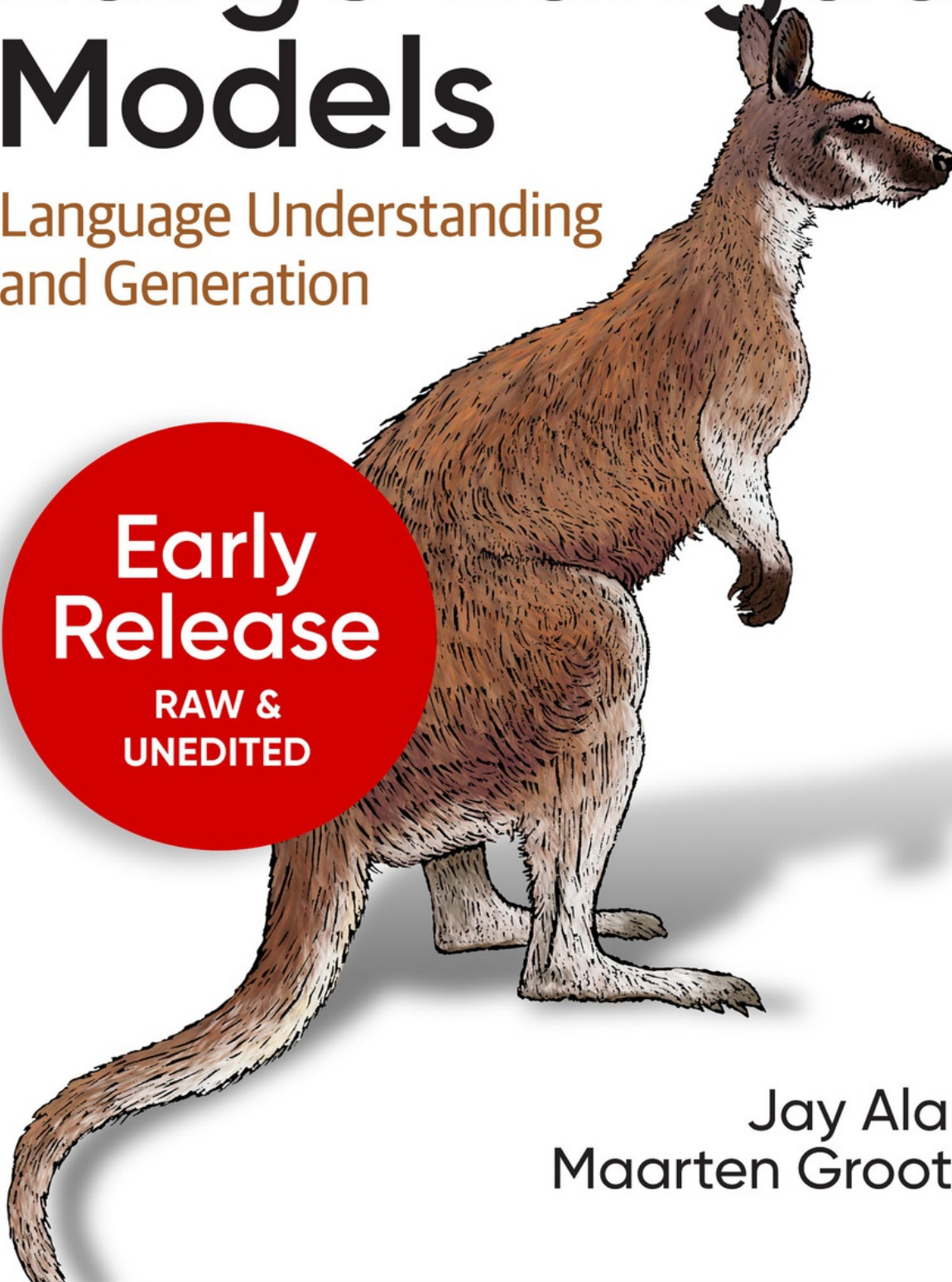
O'REILLY®

Hands-On Large Language Models

Language Understanding
and Generation

**Early
Release**

**RAW &
UNEDITED**



Jay Alammar &
Maarten Grootendorst

Hands-On Large Language Models

Language Understanding and Generation

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Jay Alamar and Maarten Grootendorst



Beijing • Boston • Farnham • Sebastopol • Tokyo

Hands-On Large Language Models

by Jay Alammar and Maarten Grootendorst

Copyright © 2025 Jay Alammar and Maarten Grootendorst. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Nicole Butterfield
- Development Editor: Michele Cronin
- Production Editor: Clare Laylock
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- December 2024: First Edition

Revision History for the Early Release

- 2023-06-09: First Release
- 2023-08-25: Second Release
- 2023-09-19: Third Release
- 2023-11-10: Fourth Release
- 2024-01-31: Fifth Release
- 2024-03-21: Sixth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098150969> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Hands-On Large Language Models*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all

responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-15090-7

[TO COME]

Brief Table of Contents (*Not Yet Final*)

Part 1: Preview of Language AI Use Cases (unavailable)

Chapter 1: Preview of Language AI Use Cases (unavailable)

Chapter 2: Categorizing Text

Chapter 3: Neural Search

Chapter 4: Text Clustering and Topic Modeling

Chapter 5: Text Generation with GPT Models

Chapter 6: Recognizing and Extracting Entities (unavailable)

Chapter 7: Multimodal Large Language Models

Part 2: Creating Language AI Models and Systems (unavailable)

Chapter 8: Tokens and Token Embeddings

Chapter 9: Looking Inside Large Language Models (unavailable)

Chapter 10: Fine-Tuning (unavailable)

Chapter 11: Building a GPT Model from Scratch (unavailable)

Chapter 12: Fine-Tuning Generation Models (unavailable)

Chapter 13: Creating Text Embedding Models

Appendix A: Essential Python Data Tools (unavailable)

Appendix B: Building Neural Networks with PyTorch (unavailable)

About the Authors

Chapter 1. Categorizing Text

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. *In particular, some of the formatting may not match the description in the text: this will be resolved when the book is finalized.*

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

One of the most common tasks in natural language processing, and machine learning in general, is classification. The goal of the task is to train a model to assign a label or class to some input text. Categorizing text is used across the world for a wide range of applications, from sentiment analysis and intent detection to

extracting entities and detecting language.

The impact of Large Language Models on categorization cannot be understated. The addition of these models has quickly settled as the default for these kinds of tasks.

In this chapter, we will discuss a variety of ways to use Large Language Modeling for categorizing text. Due to the broad field of text categorization, a variety of techniques, as well as use cases, will be discussed. This chapter also serves as a nice introduction to LLMs as most of them can be used for classification.

We will focus on leveraging pre-trained LLMs, models that already have been trained on large amounts of data and that can be used for categorizing text. Fine-tuning these models for categorizing text and domain adaptation will be discussed in more detail in Chapter 10.

Let's start by looking at the most basic application and technique, fully-supervised text classification.

Supervised Text Classification

Classification comes in many flavors, such as few-shot and zero-

shot classification which we will discuss later in this chapter, but the most frequently used method is a fully supervised classification. This means that during training, every input has a target category from which the model can learn.

For supervised classification using textual data as our input, there is a common procedure that is typically followed. As illustrated in [Figure 1-1](#), we first convert our textual input to numerical representations using a feature extraction model. Traditionally, such a model would represent text as a bag of words, simply counting the number of times a word appears in a document. In this book, however, we will be focusing on LLMs as our feature extraction model.

Movie review	Class
What a horrible movie...	0 negative
Despite some flaws, great experience.	1 positive
Best movie ever!	1 positive

Never want to see this movie again.	?
-------------------------------------	---

Figure 1-1. An example of supervised classification. Can we predict whether a movie review is either positive or negative?

Then, we train a classifier on the numerical representations, such as embeddings (remember from Chapter X?), to classify the textual data. The classifier can be a number of things, such as a neural network or logistic regression. It can even be the classifier used in many Kaggle competitions, namely XGBoost!

In this pipeline, we always need to train the classifier but we can choose to fine-tune either the entire LLM, certain parts of it, or keep it as is. If we choose not to fine-tune it all, we refer to this procedure as *freezing its layers*. This means that the layers

cannot be updated during the training process. However, it may be beneficial to *unfreeze* at least some of its layers such that the Large Language Models can be *fine-tuned* for the specific classification task. This process is illustrated in [Figure 1-2](#).



Figure 1-2. A common procedure for supervised text classification. We convert our textual input data to numerical representations through feature extraction. Then, a classifier is trained to predict labels.

Model Selection

We can use an LLM to represent the text to be fed into our classifier. The choice of this model, however, may not be as straightforward as you might think. Models differ in the language they can handle, their architecture, size, inference speed, accuracy for certain tasks, and many more differences exist.

BERT is a great underlying architecture for representing tasks that can be fine-tuned for a number of tasks, including classification. Although there are generative models that we can use, like the well-known Generated Pretrained Transformers (GPT) such as ChatGPT, BERT models often excel at being fine-

tuned for specific tasks. In contrast, GPT-like models typically excel at a broad and wide variety of tasks. In a sense, it is specialization versus generalization.

Now that we know to choose a BERT-like model for our supervised classification task, which are we going to use? BERT has a number of variations, including BERT, RoBERTa, DistilBERT, ALBERT, DeBERTa, and each architecture has been pre-trained in numerous forms, from training in certain domains to training for multi-lingual data. You can find an overview of some well-known Large Language Models in [Figure 1-3](#).

Selecting the right model for the job can be a form of art in itself. Trying thousands of pre-trained models that can be found on HuggingFace's Hub is not feasible so we need to be efficient with the models that we choose. Having said that, there are a number of models that are a great starting point and give you an idea of the base performance of these kinds of models. Consider them solid baselines:

- [BERT-base-uncased](#)
- [Roberta-base](#)
- [Distilbert-base-uncased](#)
- [Deberta-base](#)
- [BERT-tiny](#)

- [Albert-base-v2](#)

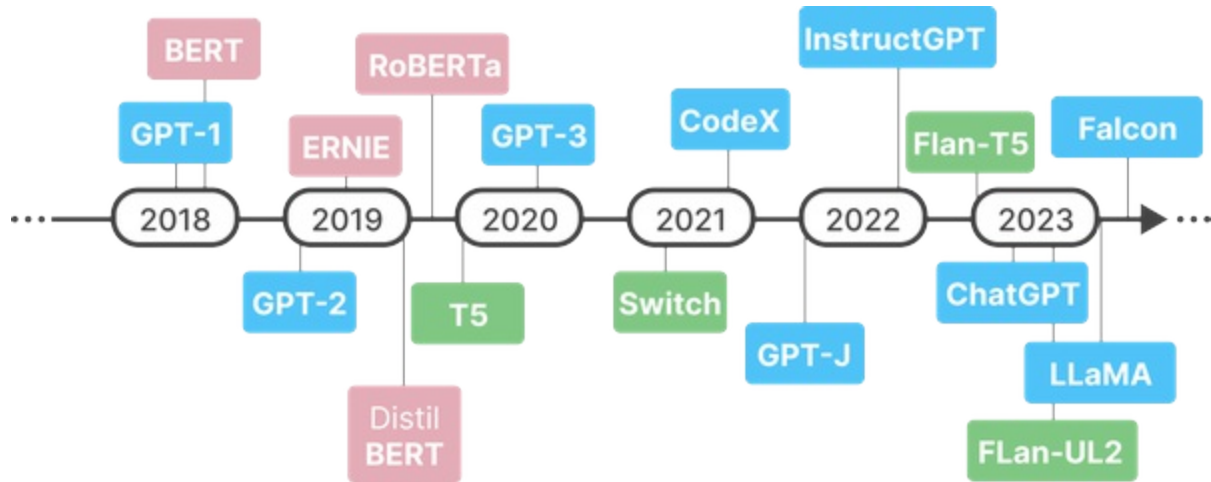


Figure 1-3. A timeline of common Large Language Model releases.

In this section, we will be using “bert-base-cased” for some of our examples. Feel free to replace “bert-base-cased” with any of the models above. Play around with different models to get a feeling for the trade-off in performance/training speed.

Data

Throughout this chapter, we will be demonstrating many techniques for categorizing text. The dataset that we will be using to train and evaluate the models is the [“rotten tomatoes”](#); pang2005seeing) dataset. It contains roughly 5000 positive and 5000 negative movie reviews from [Rotten Tomatoes](#).

We load the data and convert it to a `pandas dataframe` for easier control:

```
import pandas as pd
from datasets import load_dataset
tomatoes = load_dataset("rotten_tomatoes")

# Pandas for easier control
train_df = pd.DataFrame(tomatoes["train"])
eval_df = pd.DataFrame(tomatoes["test"])
```

TIP

Although this book focuses on LLMs, it is highly advised to compare these examples against classic, but strong baselines such as representing text with TF-IDF and training a LogisticRegression classifier on top of that.

Classification Head

Using the Rotten Tomatoes dataset, we can start with the most straightforward example of a predictive task, namely binary classification. This is often applied in sentiment analysis, detecting whether a certain document is positive or negative. This can be customer reviews with a label indicating whether that review is positive or negative (binary). In our case, we are

going to predict whether a movie review is negative (0) or positive (1).

Training a classifier with transformer-based models generally follows a two-step approach:

First, as we show in [Figure 1-4](#), we take an existing transformer model and use it to convert our textual data to numerical representations.

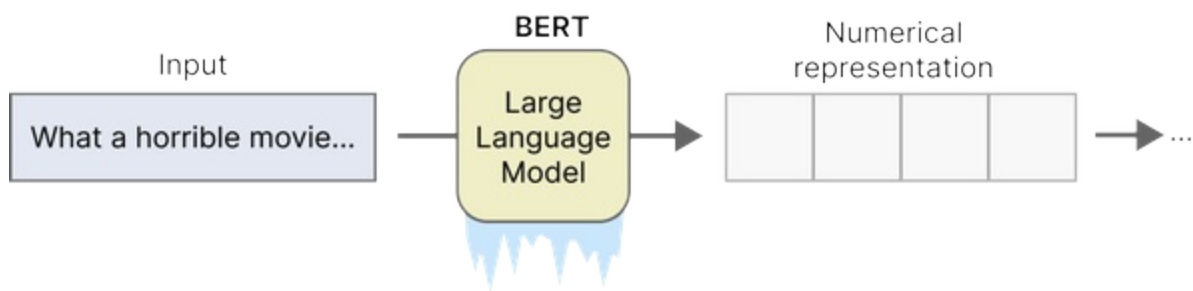


Figure 1-4. First, we start by using a generic pre-trained LLM (e.g., BERT) to convert our textual data into more numerical representations. During training, we will “freeze” the model such that its weights will not be updated. This speeds up training significantly but is generally less accurate.

Second, as shown in [Figure 1-5](#), we put a classification head on top of the pre-trained model. This classification head is generally a single linear layer that we can fine-tune.

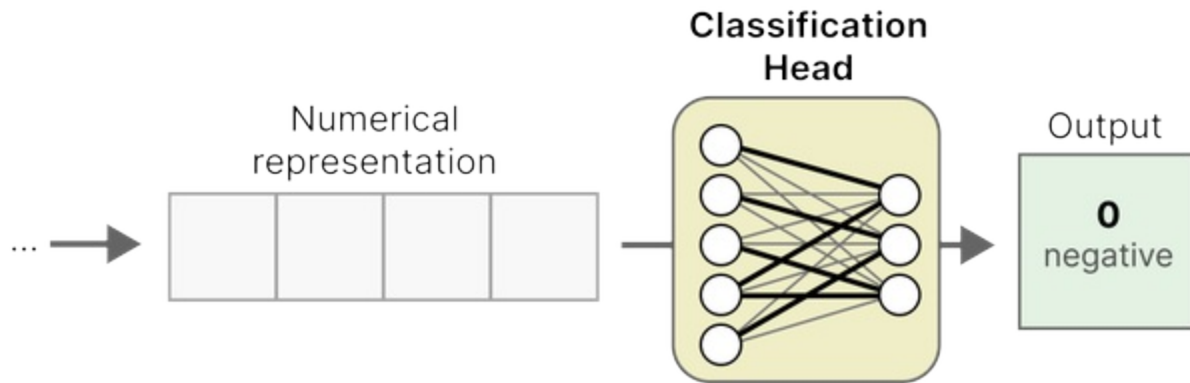


Figure 1-5. After fine-tuning our LLM, we train a classifier on the numerical representations and labels. Typically, a Feed Forward Neural Network is chosen as the classifier.

These two steps each describe the same model since the classification head is added directly to the BERT model. As illustrated in [Figure 1-6](#), our classifier is nothing more than a pre-trained LLM with a linear layer attached to it. It is feature extraction and classification in one.

Feature Extractor + Classifier

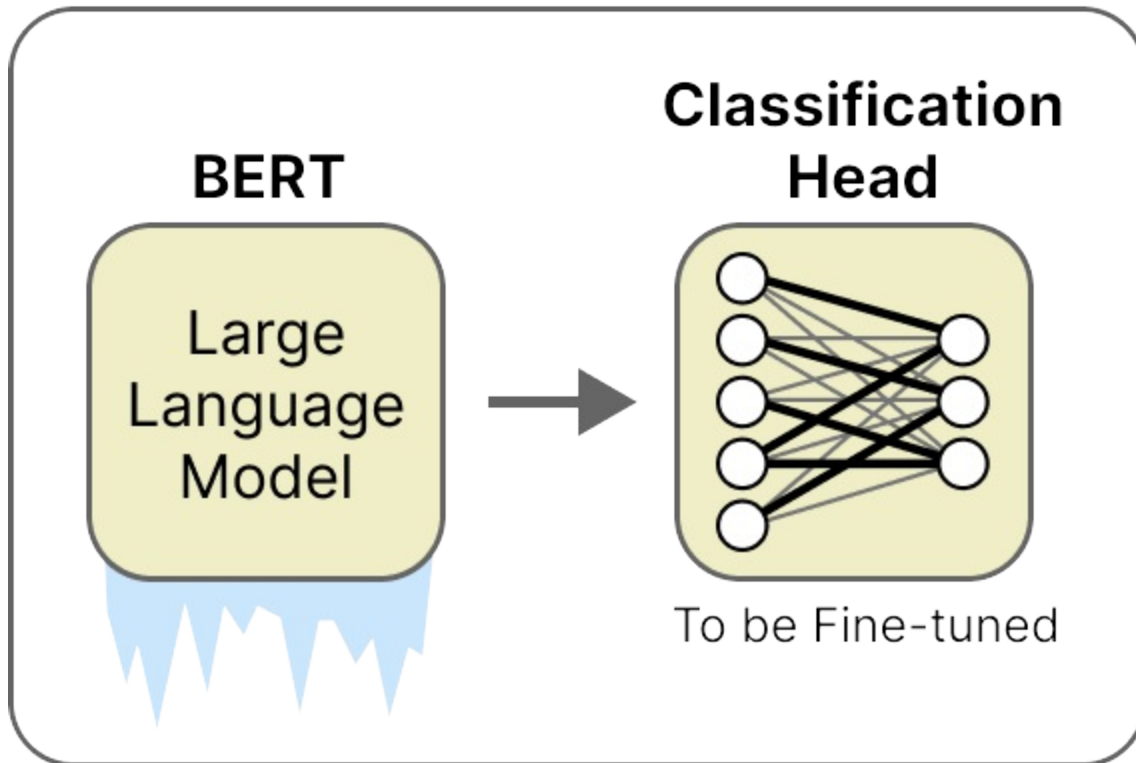


Figure 1-6. We adopt the BERT model such that its output embeddings are fed into a classification head. This head generally consists of a linear layer but might include dropout beforehand.

NOTE

In Chapter 10, we will use the same pipeline shown in Figures 2-4 and 2-5 but will instead fine-tune the Large Language Model. There, we will go more in-depth into how fine-tuning works and why it improves upon the pipeline as shown here. For now, it is essential to know that fine-tuning this model together with the classification head improves the accuracy during the classification task. The reason for this is that it allows the Large Language Model to better represent the text for classification purposes. It is fine-tuned toward the domain-specific texts.

Example

To train our model, we are going to be using the [simpletransformers package](#). It abstracts most of the technical difficulty away so that we can focus on the classification task at hand. We start by initializing our model:

```
from simpletransformers.classification import Classifi

# Train only the classifier layers
model_args = ClassificationArgs()
model_args.train_custom_parameters_only = True
model_args.custom_parameter_groups = [
    {
        "params": ["classifier.weight"],
        "lr": 1e-3,
    },
    {
        "params": ["classifier.bias"],
        "lr": 1e-3,
        "weight_decay": 0.0,
    },
]

# Initializing pre-trained BERT model
model = ClassificationModel("bert", "bert-base-cased",
```

We have chosen the popular “bert-base-cased” but as mentioned

before, there are many other models that we could have chosen instead. Feel free to play around with models to see how it influences performance.

Next, we can train the model on our training dataset and predict the labels of our evaluation dataset:

```
import numpy as np
from sklearn.metrics import f1_score

# Train the model
model.train_model(train_df)

# Predict unseen instances
result, model_outputs, wrong_predictions = model.eval_
y_pred = np.argmax(model_outputs, axis=1)
```

Now that we have trained our model, all that is left is evaluation:

```
>>> from sklearn.metrics import classification_report
>>> print(classification_report(eval_df.label, y_pred))
```

	precision	recall	f1-score	support
0	0.84	0.86	0.85	533
1	0.86	0.83	0.84	533

accuracy			0.85	1066
macro avg	0.85	0.85	0.85	1066
weighted avg	0.85	0.85	0.85	1066

Using a pre-trained BERT model for classification gives us an F-1 score of 0.85. We can use this score as a baseline throughout the examples in this section.

TIP

The `simpletransformers` package has a number of easy-to-use features for different tasks. For example, you could also use it to create a custom Named Entity Recognition model with only a few lines of code.

Pre-Trained Embeddings

Unlike the example shown before, we can approach supervised classification in a more classical form. Instead of freezing layers before training and using a feed-forward neural network on top of it, we can completely separate feature extraction and classification training.

This two-step approach completely separates feature extraction from classification:

First, as we can see in [Figure 1-7](#), we perform our feature

extraction with an LLM, SBERT (<https://www.sbert.net/>), which is trained specifically to create embeddings.

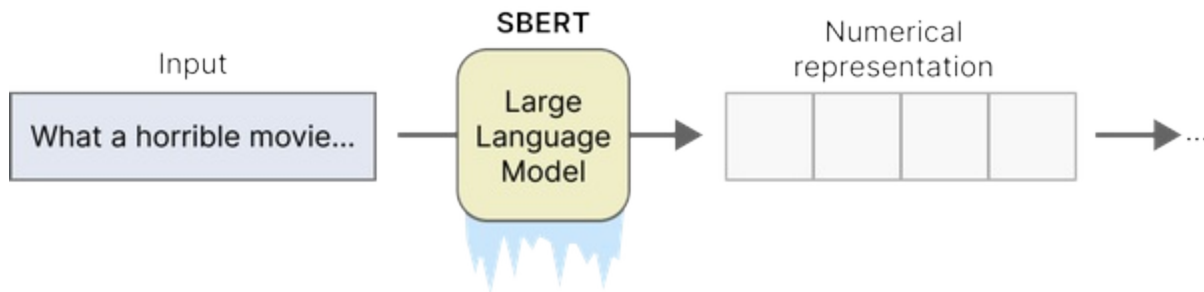


Figure 1-7. First, we use an LLM that was trained specifically to generate accurate numerical representations. These tend to be better representative vectors than we receive from a general Transformer-based model like BERT.

Second, as shown in [Figure 1-8](#), we use the embeddings as input for a logistic regression model. We are completely separating the feature extraction model from the classification model.

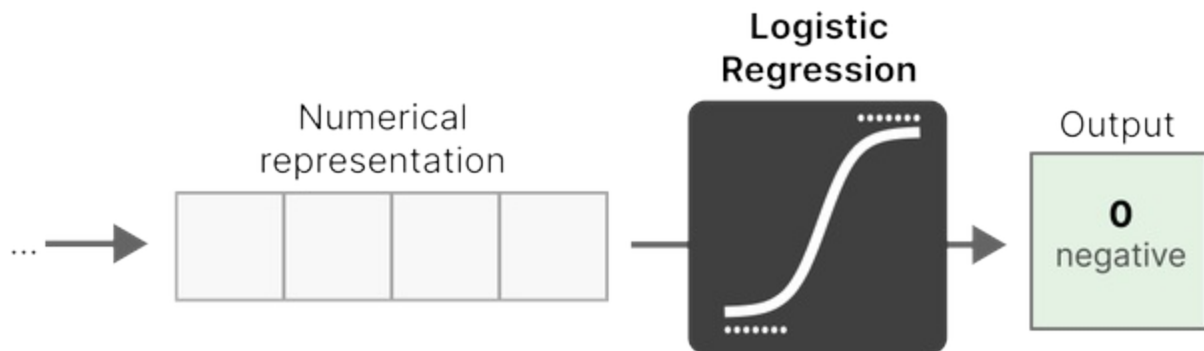


Figure 1-8. Using the embeddings as our features, we train a logistic regression model on our training data.

In contrast to our previous example, these two steps each describe a different model. SBERT for generating features, namely embeddings, and a Logistic Regression as the classifier.

As illustrated in Figure 2-9, our classifier is nothing more than a pre-trained LLM with a linear layer attached to it.

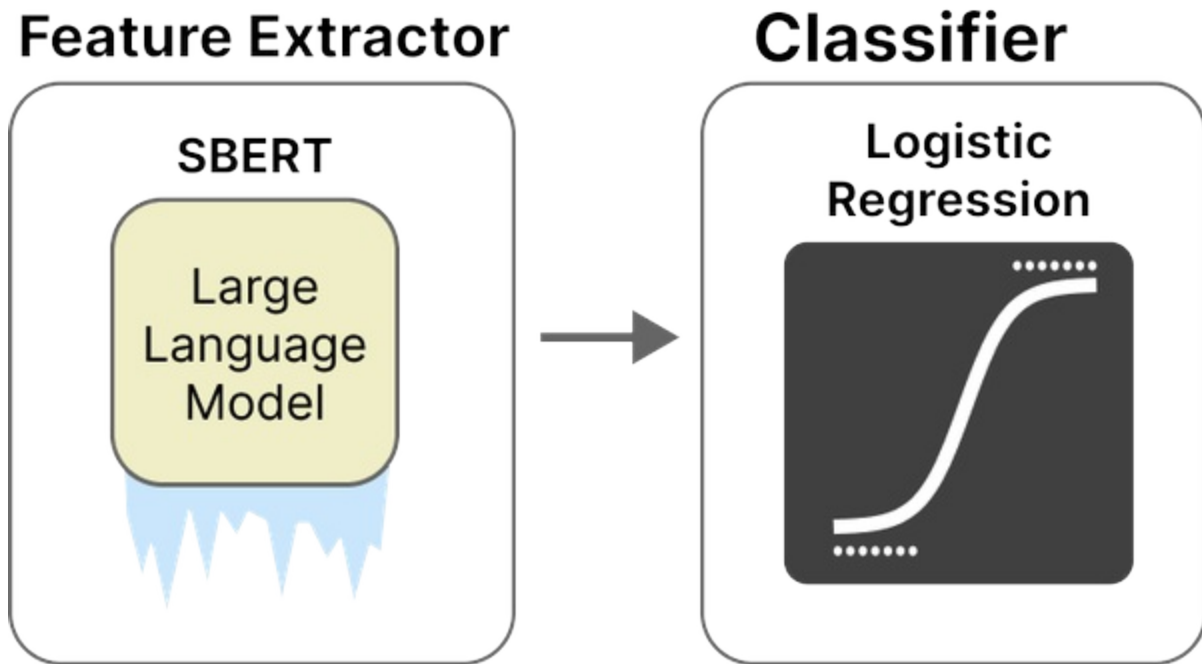


Figure 1-9. The classifier is a separate model that leverages the embeddings from SBERT to learn from.

Example

Using sentence-transformer, we can create our features before training our classification model:

```
from sentence_transformers import SentenceTransformer,
model = SentenceTransformer('all-mpnet-base-v2')
train_embeddings = model.encode(train_df.text)
eval_embeddings = model.encode(eval_df.text)
```

We created the embeddings for our training (train_df) and evaluation (eval_df) data. Each instance in the resulting embeddings is represented by 768 values. We consider these values the features on which we can train our model.

Selecting the model can be straightforward. Instead of using a feed-forward neural network, we can go back to the basics and use a Logistic Regression instead:

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=42).fit(train_em
```

In practice, you can use any classifier on top of our generated embeddings, like Decision Trees or Neural Networks.

Next, let's evaluate our model:

```
>>> from sklearn.metrics import classification_report
>>> y_pred = clf.predict(eval_embeddings)
>>> print(classification_report(eval_df.label, y_pred))
```

	precision	recall	f1-score	support
0	0.84	0.86	0.85	151

1	0.86	0.83	0.84	149
accuracy			0.85	300
macro avg	0.85	0.85	0.85	300
weighted avg	0.85	0.85	0.85	300

Without needing to fine-tune our LLM, we managed to achieve an F1-score of 0.85. This is especially impressive since it is a much smaller model compared to our previous example.

Zero-shot Classification

We started this chapter with examples where all of our training data has labels. In practice, however, this might not always be the case. Getting labeled data is a resource-intensive task that can require significant human labor. Instead, we can use zero-shot classification models. This method is a nice example of transfer learning where a model trained for one task is used for a task different than what it was originally trained for. An overview of zero-shot classification is given in Figure 2-11. Note that this pipeline also demonstrates the capabilities of performing multi-label classification if the probabilities of multiple labels exceed a given threshold.

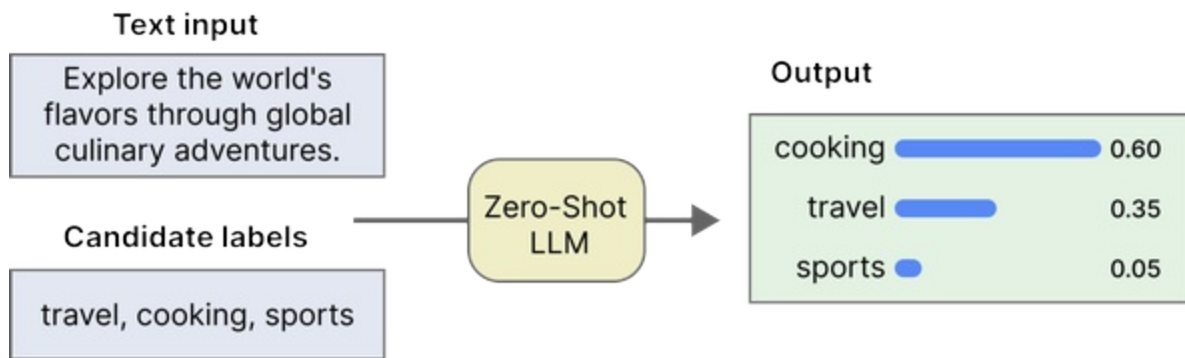


Figure 1-10. Figure 2-11. In zero-shot classification, the LLM is not trained on any of the candidate labels. It learned from different labels and generalized that information to the candidate labels.

Often, zero-shot classification tasks are used with pre-trained LLMs that use natural language to describe what we want our model to do. It is often referred to as an emergent feature of LLMs as the models increase in size (wei2022emergent). As we will see later in this chapter on classification with generative models, GPT-like models can often do these kinds of tasks quite well.

Pre-Trained Embeddings

As we have seen in our supervised classification examples, embeddings are a great and often accurate way of representing textual data. When dealing with no labeled documents, we have to be a bit creative in how we are going to be using pre-trained embeddings. A classifier cannot be trained since we have no labeled data to work with.

Fortunately, there is a trick that we can use. We can describe our labels based on what they should represent. For example, a negative label for movie reviews can be described as “This is a negative movie review”. By describing and embedding the labels and documents, we have data that we can work with. This process, as illustrated in [Figure 1-11](#), allows us to generate our own target labels without the need to actually have any labeled data.

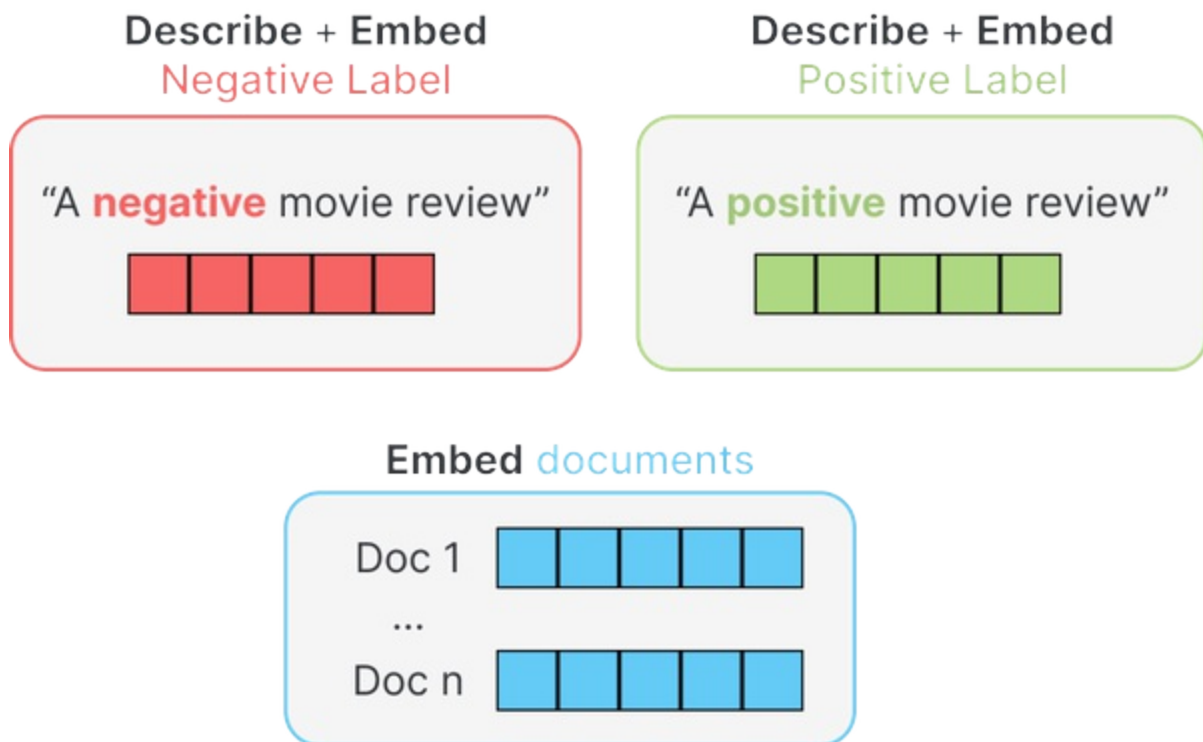


Figure 1-11. To embed the labels, we first need to give them a description. For example, the description of a negative label could be “A negative movie review”. This description can then be embedded through sentence-transformers. In the end, both labels as well as all the documents are embedded.

To assign labels to documents, we can apply cosine similarity to

the document label pairs. Cosine similarity, which will often be used throughout this book, is a similarity measure that checks how similar two vectors are to each other.

It is the cosine of the angle between vectors which is calculated through the dot product of the embeddings and divided by the product of their lengths. It definitely sounds more complicated than it is and, hopefully, the illustration in [Figure 1-12](#) should provide additional intuition.

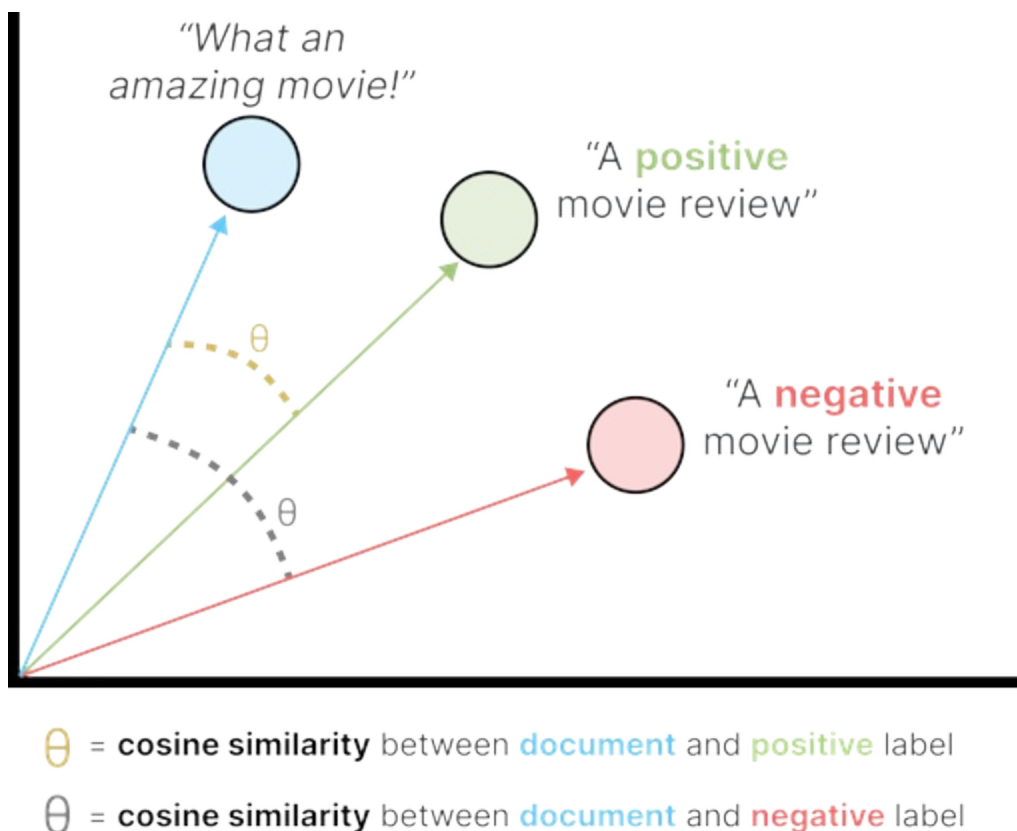


Figure 1-12. The cosine similarity is the angle between two vectors or embeddings. In this example, we calculate the similarity between a document and the two possible labels, positive and negative.

For each document, its embedding is compared to that of each label. The label with the highest similarity to the document is chosen. [Figure 1-13](#) gives a nice example of how a document is assigned a label.

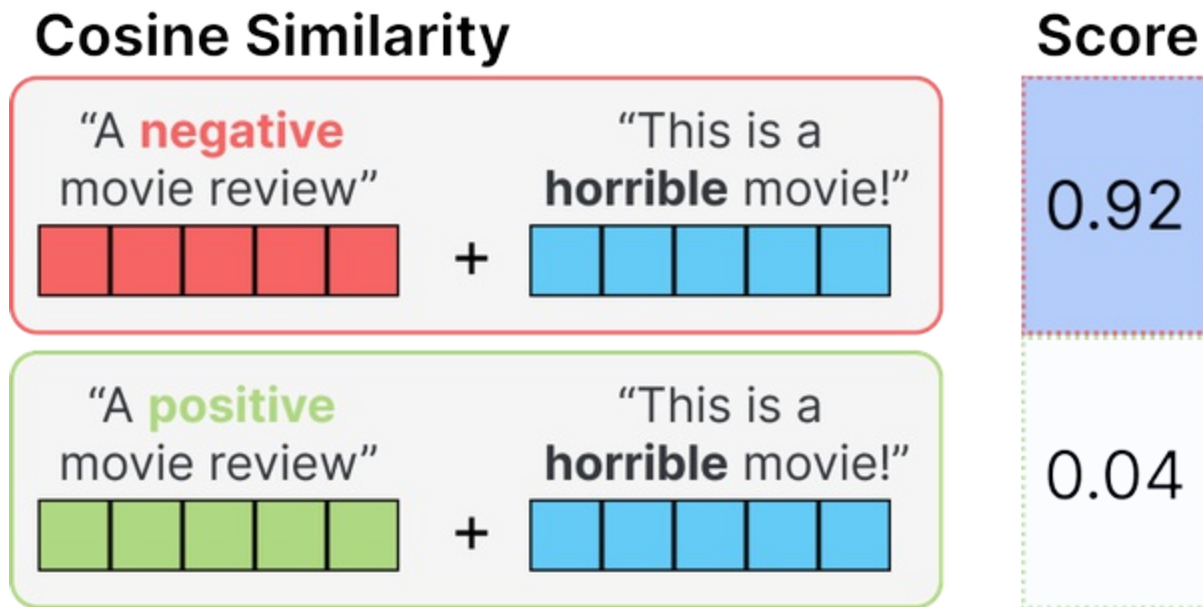


Figure 1-13. After embedding the label descriptions and the documents, we can use cosine similarity for each label document pair. For each document, the label with the highest similarity to the document is chosen.

Example

We start by generating the embeddings for our evaluation dataset. These embeddings are generated with sentence-transformers as they are quite accurate and are computationally quite fast.

```
from sentence_transformers import SentenceTransformer,
```

```
# Create embeddings for the input documents
model = SentenceTransformer('all-mpnet-base-v2')
eval_embeddings = model.encode(eval_df.text)
```

Next, embeddings of the labels need to be generated. The labels, however, do not have a textual representation that we can leverage so we will instead have to name the labels ourselves.

Since we are dealing with positive and negative movie reviews, let's name the labels "A positive review" and "A negative review". This allows us to embed those labels:

```
# Create embeddings for our labels
label_embeddings = model.encode(["A negative review",
```

Now that we have embeddings for our reviews and the labels, we can apply cosine similarity between them to see which label fits best with which review. Doing so requires only a few lines of code:

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Find the best matching label for each document
```

```
sim_matrix = cosine_similarity(eval_embeddings, label_embeddings)
y_pred = np.argmax(sim_matrix, axis=1)
```

And that is it! We only needed to come up with names for our labels to perform our classification tasks. Let's see how well this method works:

```
>>> print(classification_report(eval_df.label, y_pred))
```

	precision	recall	f1-score	support
0	0.83	0.77	0.80	151
1	0.79	0.84	0.81	149
accuracy			0.81	300
macro avg	0.81	0.81	0.81	300
weighted avg	0.81	0.81	0.81	300

An F-1 score of 0.81 is quite impressive considering we did not use any labeled data at all! This just shows how versatile and useful embeddings are especially if you are a bit creative with how they are used.

Let's put that creativity to the test. We decided upon "A negative/positive review" as the names of our labels but that can

be improved. Instead, we can make them a bit more concrete and specific towards our data by using “A very negative/positive movie review” instead. This way, the embedding will capture that it is a movie review and will focus a bit more on the extremes of the two labels.

We use the code we used before to see whether this actually works:

```
>>> # Create embeddings for our labels
>>> label_embeddings = model.encode(["A very negative
>>>
>>> # Find the best matching label for each document
>>> sim_matrix = cosine_similarity(eval_embeddings, la
>>> y_pred = np.argmax(sim_matrix, axis=1)
>>>
>>> # Report results
>>> print(classification_report(eval_df.label, y_pred))
```

	precision	recall	f1-score	support
0	0.90	0.74	0.81	151
1	0.78	0.91	0.84	149
accuracy			0.83	300
macro avg	0.84	0.83	0.83	300
weighted avg	0.84	0.83	0.83	300

By only changing the phrasing of the labels, we increased our F-1 score quite a bit!

TIP

In the example, we applied zero-shot classification by naming the labels and embedding them. When we have a few labeled examples, embedding them and adding them to the pipeline could help increase the performance. For example, we could average the embeddings of the labeled examples together with the label embeddings. We could even do a voting procedure by creating different types of representations (label embeddings, document embeddings, averaged embeddings, etc.) and see which label is most often found. This would make our zero-shot classification example a few-shot approach.

Natural Language Inference

Zero-shot classification can also be done using natural language inference (NLI), which refers to the task of investigating whether, for a given premise, a hypothesis is true (entailment) or false (contradiction). [Figure 1-14](#) shows a nice example how they relate to one another.

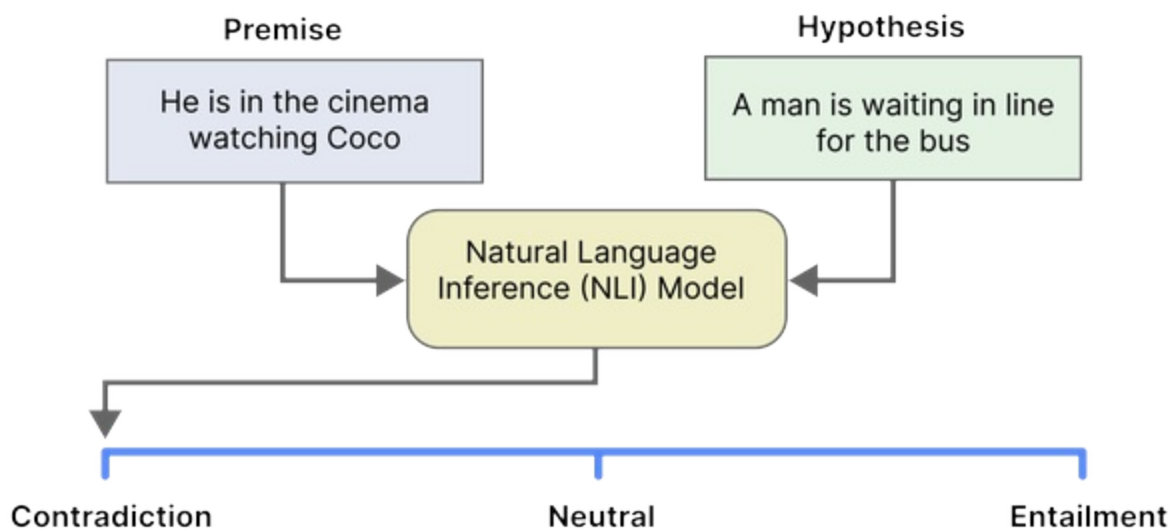


Figure 1-14. An example of natural language inference (NLI). The hypothesis is contradicted by the premise and is not relevant to one another.

NLI can be used for zero-shot classification by being a bit creative with how the premise/hypothesis pair is used, as demonstrated in [Figure 1-15](#). We use the input document, the review that we want to extract sentiment from and use that as our premise (yin2019benchmarking). Then, we create a hypothesis asking whether the premise is about our target label. In our movie reviews example, the hypothesis could be: “This example is a positive movie review”. When the model finds it to be an entailment, we can label the review as positive and negative when it is a contradiction. Using NLI for zero-shot classification is illustrated with an example in [Figure 1-15](#).

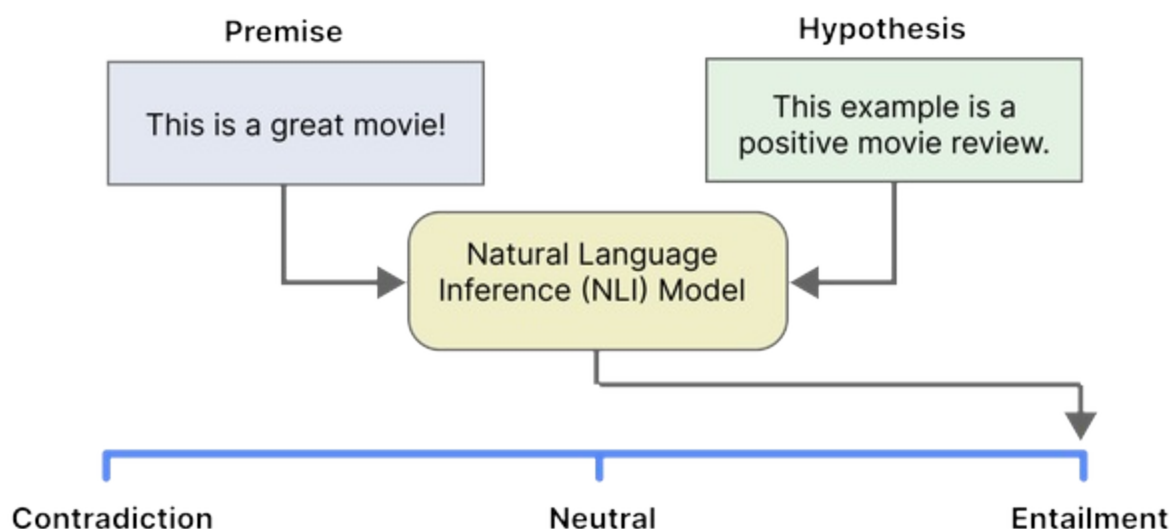


Figure 1-15. An example of zero-shot classification with natural language inference (NLI). The hypothesis is supported by the premise and the model will return that the review is indeed a positive movie review.

Example

With transformers, loading and running a pre-trained NLI model is straightforward. Let's select "facebook /bart-large-mnli" as our pre-trained model. The model was trained on more than 400k premise/hypothesis pairs and should serve well for our use case.

NOTE

Over the course of the last few years, Hugging Face has strived to become the Github of Machine Learning by hosting pretty much everything related to Machine Learning. As a result, there is a large amount of pre-trained models available on their hub. For zero-shot classification tasks, you can follow this link: https://huggingface.co/models?pipeline_tag=zero-shot-classification.

We load in our transformers pipeline and run it on our evaluation dataset:

```
from transformers import pipeline

# Pre-trained MNLI model
pipe = pipeline(model="facebook/bart-large-mnli")

# Candidate labels
candidate_labels_dict = {"negative movie review": 0, "
candidate_labels = ["negative movie review", "positive

# Create predictions
predictions = pipe(eval_df.text.values.tolist(), candi
```

Since this is a zero-shot classification task, no training is necessary for us to get the predictions that we are interested in. The predictions variable contains not only the prediction but also a score indicating the probability of a candidate label (hypothesis) to entail the input document (premise).

```
>>> from sklearn.metrics import classification_report
>>> y_pred = [candidate_labels_dict[prediction["labels
>>> print(classification_report(eval_df.label, y_pred)
```

precision	recall	f1-score	support
-----------	--------	----------	---------

	0	0.77	0.89	0.83	151
	1	0.87	0.74	0.80	149
accuracy				0.81	300
macro avg		0.82	0.81	0.81	300
weighted avg		0.82	0.81	0.81	300

Without any fine-tuning whatsoever, it received an F1-score of 0.81. We might be able to increase this value depending on how we phrase the candidate labels. For example, see what happens if the candidate labels were simply “negative” and “positive” instead.

TIP

Another great pre-trained model for zero-shot classification is sentence-transformers' cross-encoder, namely 'cross-encoder/ nli -deberta-base'. Since training a sentence-transformers model focuses on pairs of sentences, it naturally lends itself to zero-shot classification tasks that leverage premise/hypothesis pairs.

Classification with Generative Models

Classification with generative large language models, such as

OpenAI's GPT models, works a bit differently from what we have done thus far. Instead of fine-tuning a model to our data, we use the model and try to guide it toward the type of answers that we are looking for.

This guiding process is done mainly through the prompts that you give such as a model. Optimizing the prompts such that the model understands what kind of answer you are looking for is called **prompt engineering**. This section will demonstrate how we can leverage generative models to perform a wide variety of classification tasks.

This is especially true for extremely large language models, such as GPT-3. An excellent paper and read on this subject, "Language Models are Few-Shot Learners", describes that these models are competitive on downstream tasks whilst needing less task-specific data (brown2020language).

In-Context Learning

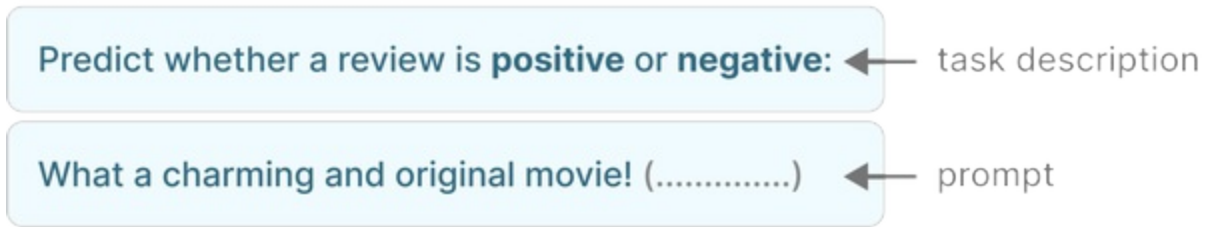
What makes generative models so interesting is their ability to follow the prompts they are given. A generative model can even do something entirely new by merely being shown a few examples of this new task. This process is also called in-context learning and refers to the process of having the model learn or

do something new without actually fine-tuning it.

For example, if we ask a generative model to write a haiku (a traditional Japanese poetic form), it might not be able to if it has not seen a haiku before. However, if the prompt contains a few examples of what a haiku is, then the model “learns” from that and is able to create haikus.

We purposely put “learning” in quotation marks since the model is not actually learning but following examples. After successfully having generated the haikus, we would still need to continuously provide it with examples as the internal model was not updated. These examples of in-context learning are shown in [Figure 1-16](#) and demonstrate the creativity needed to create successful and performant prompts.

Zero-shot Classification



Few-shot Classification



Figure 1-16. Zero-shot and few-shot classification through prompt engineering with generative models.

In-context learning is especially helpful in few-shot classification tasks where we have a small number of examples that the generative model can follow.

Not needing to fine-tune the internal model is a major advantage of in-context learning. These generative models are often quite large in size and are difficult to run on consumer hardware let alone fine-tune them. Optimizing your prompts to guide the generative model is relatively low-effort and often does not need

somebody well-versed in generative AI.

Example

Before we go into the examples of in-context learning, we first create a function that allows us to perform prediction with OpenAI's GPT models.

```
from tenacity import retry, stop_after_attempt, wait_

@retry(wait=wait_random_exponential(min=1, max=60), st
def gpt_prediction(prompt, document, model="gpt-3.5-tu
    messages=[
        {"role": "system", "content": "You are a helpful a
        {"role": "user", "content": prompt.replace("[DOC
    ]
    response = openai.ChatCompletion.create(model=model,
    return response["choices"][0]["message"]["content"]
```

This function allows us to pass a specific `prompt` and `document` for which we want to create a prediction. The `tenacity` module that you also see here allows us to deal with rate limit errors, which happen when you call the API too often. OpenAI, and other external APIs, often want to limit the rate at which you call their API so as not to overload their servers.

This `tenacity` module is essentially a “retrying module” that allows us to retry API calls in specific ways. Here, we implemented something called **exponential backoff** to our `gpt_prediction` function. Exponential backoff performs a short sleep when we hit a rate limit error and then retries the unsuccessful request. Every time the request is unsuccessful, the sleep length is increased until the request is successful or we hit a maximum number of retries.

One easy way to avoid rate limit errors is to automatically retry requests with a random exponential backoff. Retrying with exponential backoff means performing a short sleep when a rate limit error is hit, then retrying the unsuccessful request. If the request is still unsuccessful, the sleep length is increased and the process is repeated. This continues until the request is successful or until a maximum number of retries is reached.

Lastly, we need to sign in to OpenAI’s API with an API-key that you can get from your account:

```
import openai
openai.api_key = "sk-..."
```

WARNING

When using external APIs, always keep track of your usage. External APIs, such as

OpenAI or Cohere, can quickly become costly if you request too often from their APIs.

Zero-shot Classification

Zero-shot classification with generative models is essentially what we typically do when interacting with these types of models, simply ask them if they can do something. In our examples, we ask the model whether a specific document is a positive or negative movie review.

To do so, we create a base template for our zero-shot classification prompt and ask the model if it can predict whether a review is positive or negative:

```
# Define a zero-shot prompt as a base
zeroshot_prompt = """Predict whether the following doc

[DOCUMENT]

If it is positive say 1 and if it is negative say 0. D
"""
```

You might have noticed that we explicitly say to not give any other answers. These generative models tend to have a mind of their own and return large explanations as to why something is

or isn't negative. Since we are evaluating its results, we want either a 0 or a 1 to be returned.

Next, let's see if it can correctly predict that the review "unpretentious, charming, quickie, original" is positive:

```
# Define a zero-shot prompt as a base
zeroshot_prompt = """Predict whether the following doc

[DOCUMENT]

If it is positive say 1 and if it is negative say 0. D
"""

# Predict the target using GPT
document = "unpretentious , charming , quirky , origin
gpt_prediction(zeroshot_prompt, document)
```

The output indeed shows that the review was labeled by OpenAI's model as positive! Using this prompt template, we can insert any document at the "[DOCUMENT]" tag. These models have token limits which means that we might not be able to insert an entire book into the prompt. Fortunately, reviews tend not to be the sizes of books but are often quite short.

Next, we can run this for all reviews in the evaluation dataset

and look at its performance. Do note though that this requires 300 requests to OpenAI's API:

```
> from sklearn.metrics import classification_report
> from tqdm import tqdm
>
> y_pred = [int(gpt_prediction(zeroshot_prompt, doc))
> print(classification_report(eval_df.label, y_pred))
```

	precision	recall	f1-score	support
0	0.86	0.96	0.91	151
1	0.95	0.86	0.91	149
accuracy			0.91	300
macro avg	0.91	0.91	0.91	300
weighted avg	0.91	0.91	0.91	300

An F-1 score of 0.91! That is the highest we have seen thus far and is quite impressive considering we did not fine-tune the model at all.

NOTE

Although this zero-shot classification with GPT has shown high performance, it should be noted that fine-tuning generally outperforms in-context learning as presented in this section. This is especially true if domain-specific data is involved which the model

during pre-training is unlikely to have seen. A model's adaptability to task-specific nuances might be limited when its parameters are not updated for the task at hand. Preferably, we would want to fine-tune this GPT model on this data to improve its performance even further!

Few-shot Classification

In-context learning works especially well when we perform few-shot classification. Compared to zero-shot classification, we simply add a few examples of movie reviews as a way to guide the generative model. By doing so, it has a better understanding of the task that we want to accomplish.

We start by updating our prompt template to include a few hand-picked examples:

```
# Define a few-shot prompt as a base
fewshot_prompt = """Predict whether the following docu
```

```
[DOCUMENT]
```

```
Examples of negative reviews are:
```

- a film really has to be exceptional to justify a thr
- the film , like jimmy's routines , could use a few g

```
Examples of positive reviews are:
```

- very predictable but still entertaining
- a solid examination of the male midlife crisis .

If it is positive say 1 and if it is negative say 0. D
""

We picked two examples per class as a quick way to guide the model toward assigning sentiment to movie reviews.

NOTE

Since we added a few examples to the prompt, the generative model consumes more tokens and as a result could increase the costs of requesting the API. However, that is relatively little compared to fine-tuning and updating the entire model.

Prediction is the same as before but replacing the zero-shot prompt with the few-shot prompt:

```
# Predict the target using GPT
document = "unpretentious , charming , quirky , origin
gpt_prediction(fewshot_prompt, document)
```

Unsurprisingly, it correctly assigned sentiment to the review. The more difficult or complex the task is, the bigger the effect of providing examples, especially if they are high-quality.

As before, let's run the improved prompt against the entire evaluation dataset:

```
>>> predictions = [gpt_prediction(fewshot_prompt, doc)
```

	precision	recall	f1-score	support
0	0.88	0.97	0.92	151
1	0.96	0.87	0.92	149
accuracy			0.92	300
macro avg	0.92	0.92	0.92	300
weighted avg	0.92	0.92	0.92	300

The F1-score is now 0.92 which is a very slight increase compared to what we had before. This is not unexpected since its score was already quite high and the task at hand was not particularly complex.

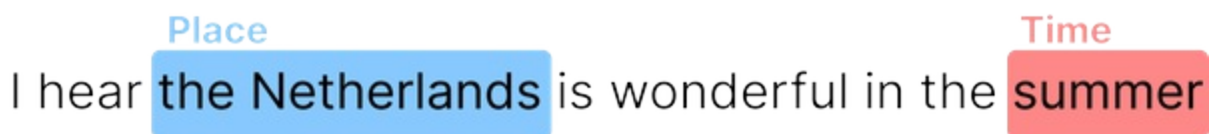
NOTE

We can extend the examples of in-context learning to multi-label classification by engineering the prompt. For example, we can ask the model to choose one or multiple labels and return them separated by commas.

Named Entity Recognition

In the previous examples, we have tried to classify entire texts, such as reviews. There are many cases though where we are more interested in specific information inside those texts. We may want to extract certain medications from textual electronic health records or find out which organizations are mentioned in news posts.

These tasks are typically referred to as token classification or Named Entity Recognition (NER) which involves detecting these entities in text. As illustrated in [Figure 1-17](#), instead of classifying an entire text, we are now going to classify certain tokens or token sets.



I hear **the Netherlands** is wonderful in the **summer**

Figure 1-17. An example of named entity recognition that detects the entities “place” and “time”.

When we think about token classification, one major framework comes into mind, namely SpaCy (<https://spacy.io/>). It is an incredible package for performing many industrial-strength NLP applications and has been the go-to framework for NER tasks. So, let’s use it!

Example

To use OpenAI's models with SpaCy, we will first need to save the API key as an environment variable. This makes it easier for SpaCy to access it without the need to save it locally:

```
import os
os.environ['OPENAI_API_KEY'] = "sk-..."
```

Next, we need to configure our SpaCy pipeline. A “task” and a “backend” will need to be defined. The “task” is what we want the SpaCy pipeline to do, which is Named Entity Recognition. The “backend” is the underlying LLM that is used to perform the “task” which is OpenAI's GPT-3.5-turbo model. In the task, we can create any labels that we would like to extract from our text. Let's assume that we have information about patients and we would like to extract some personal information but also the disease and symptoms they developed. We create the entities date, age, location, disease, and symptom:

```
import spacy

nlp = spacy.blank("en")

# Create a Named Entity Recognition Task and define la
```

```
task = {"task": {
    "@llm_tasks": "spacy.NER.v1",
    "labels": "DATE,AGE,LOCATION, DISEASE, SYM

# Choose which backend to use
backend = {"backend": {
    "@llm_backends": "spacy.REST.v1",
    "api": "OpenAI",
    "config": {"model": "gpt-3.5-turbo"}}}}

# Combine configurations and create SpaCy pipeline
config = task | backend
nlp.add_pipe("llm", config=config)
```

Next, we only need two lines of code to automatically extract the entities that we are interested in:

```
> doc = nlp("On February 11, 2020, a 73-year-old woman
> print([(ent.text, ent.label_) for ent in doc.ents])

[('February 11', 'DATE'), ('2020', 'DATE'), ('73-year-
```

It seems to correctly extract the entities but it is difficult to immediately see if everything worked out correctly. Fortunately, SpaCy has a display function that allows us to visualize the

entities found in the document ([Figure 1-18](#)):

```
from spacy import displacy
from IPython.core.display import display, HTML

# Display entities
html = displacy.render(doc, style="ent")
display(HTML(html))
```

On February 11 DATE , 2020 DATE , a 73-year-old AGE woman came to the hospital LOCATION and was diagnosed with COVID-19 DISEASE and has a cough SYMPTOM .

Figure 1-18. The output of SpaCy using OpenAI's GPT-3.5 model. Without any training, it correctly identifies our custom entities.

That is much better! Figure 2-X shows that we can clearly see that the model has correctly identified our custom entities. Without any fine-tuning or training of the model, we can easily detect entities that we are interested in.

TIP

Training a NER model from scratch with SpaCy is not possible with only a few lines of code but it is also by no means difficult! Their [documentation and tutorials](#) are, in our opinions, state-of-the-art and do an excellent job of explaining how to create a custom model.

Summary

In this chapter, we saw many different techniques for performing a wide variety of classification tasks. From fine-tuning your entire model to no tuning at all! Classifying textual data is not as straightforward as it may seem on the surface and there is an incredible amount of creative techniques for doing so.

In the next chapter, we will continue with classification but focus instead on unsupervised classification. What can we do if we have textual data without any labels? What information can we extract? We will focus on clustering our data as well as naming the clusters with topic modeling techniques.

Chapter 2. Semantic Search

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. *In particular, some of the formatting may not match the description in the text: this will be resolved when the book is finalized.*

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Search was one of the first Large Language Model (LLM) applications to see broad industry adoption. Months after the release of the seminal [BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding](#) paper, Google announced it was using it to power Google Search and that it

represented “one of the biggest leaps forward in the history of Search”. Not to be outdone, Microsoft Bing also stated that “Starting from April of this year, we used large transformer models to deliver the largest quality improvements to our Bing customers in the past year”.

This is a clear testament to the power and usefulness of these models. Their addition instantly and massively improves some of the most mature, well-maintained systems that billions of people around the planet rely on. The ability they add is called *semantic search*, which enables searching by meaning, and not simply keyword matching.

In this chapter, we’ll discuss three major ways of using language models to power search systems. We’ll go over code examples where you can use these capabilities to power your own applications. Note that this is not only useful for web search, but that search is a major component of most apps and products. So our focus will not be just on building a web search engine, but rather on your own dataset. This capability powers lots of other exciting LLM applications that build on top of search (e.g., retrieval-augmented generation, or document question answering). Let’s start by looking at these three ways of using LLMs for semantic search.

Three Major Categories of Language-Model-based Search Systems

There's a lot of research on how to best use LLMs for search. Three broad categories of these models are:

1- Dense Retrieval

Say that a user types a search query into a search engine.

Dense retrieval systems rely on the concept of embeddings, the same concept we've encountered in the previous chapters, and turn the search problem into retrieving the nearest neighbors of the search query (after both the query and the documents are converted into embeddings). [Figure 2-1](#) shows how dense retrieval takes a search query, consults its archive of texts, and outputs a set of relevant results.

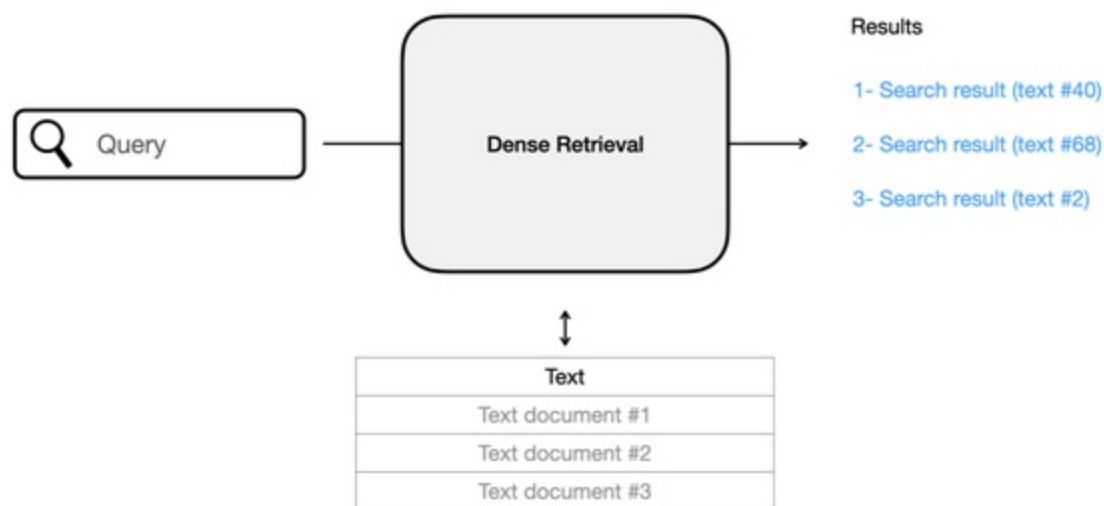


Figure 2-1. Dense retrieval is one of the key types of semantic search, relying on the similarity of text embeddings to retrieve relevant results

2- Reranking

These systems are pipelines of multiple steps. A Reranking LLM is one of these steps and is tasked with scoring the relevance of a subset of results against the query, and then the order of results is changed based on these scores. [Figure 2-2](#) shows how rerankers are different from dense retrieval in that they take an additional input: a set of search results from a previous step in the search pipeline.

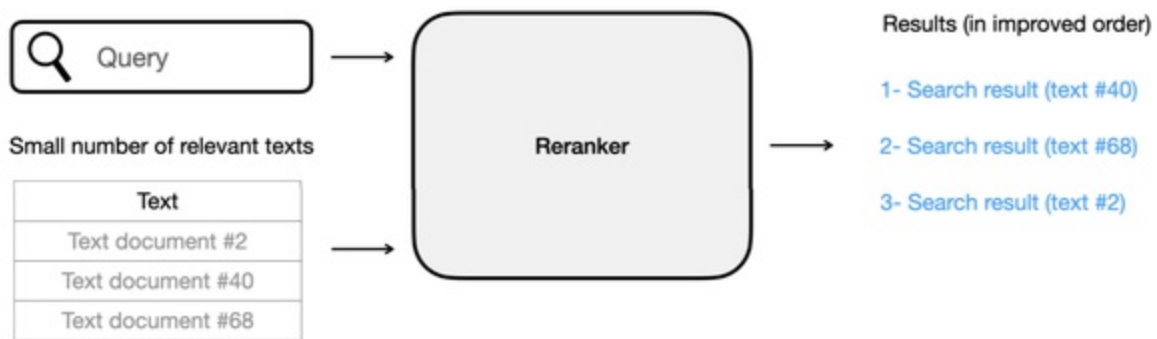


Figure 2-2. Rerankers, the second key type of semantic search, take a search query and a collection of results, and re-order them by relevance, often resulting in vastly improved results.

3- Generative Search

The growing LLM capability of text generation led to a new batch of search systems that include a generation model that simply generates an answer in response to a query. [Figure 2-3](#) shows a generative search example.

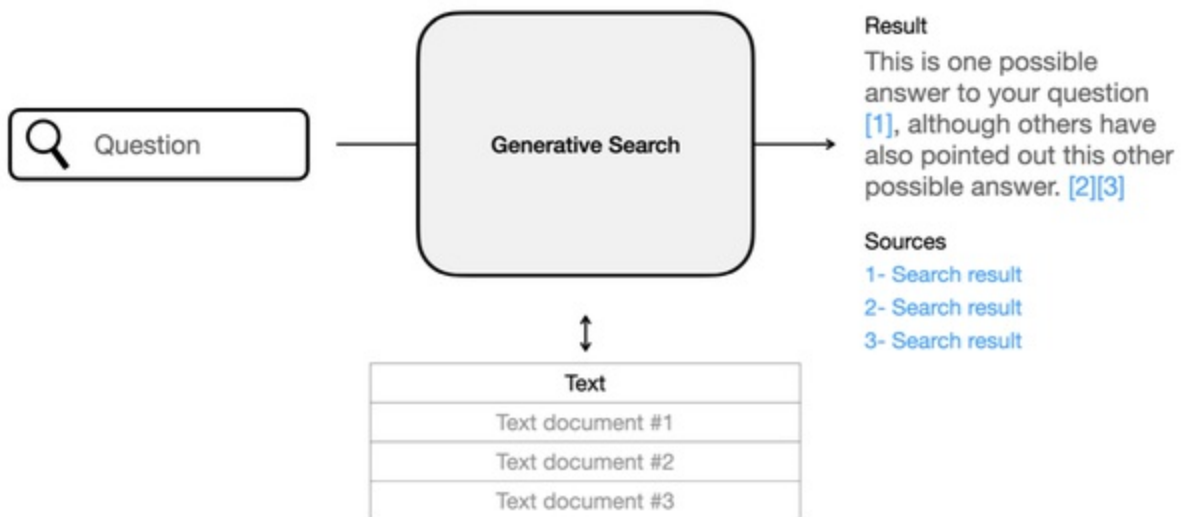


Figure 2-3. Generative search formulates an answer to a question and cites its information sources.

All three concepts are powerful and can be used together in the same pipeline. The rest of the chapter covers these three types of systems in more detail. While these are the major categories, they are not the only LLM applications in the domain of search.

Dense Retrieval

Recall that embeddings turn text into numeric representations. Those can be thought of as points in space as we can see in [Figure 2-4](#). Points that are close together mean that the text they represent is similar. So in this example, text 1 and text 2 are similar to each other (because they are near each other), and different from text 3 (because it's farther away).

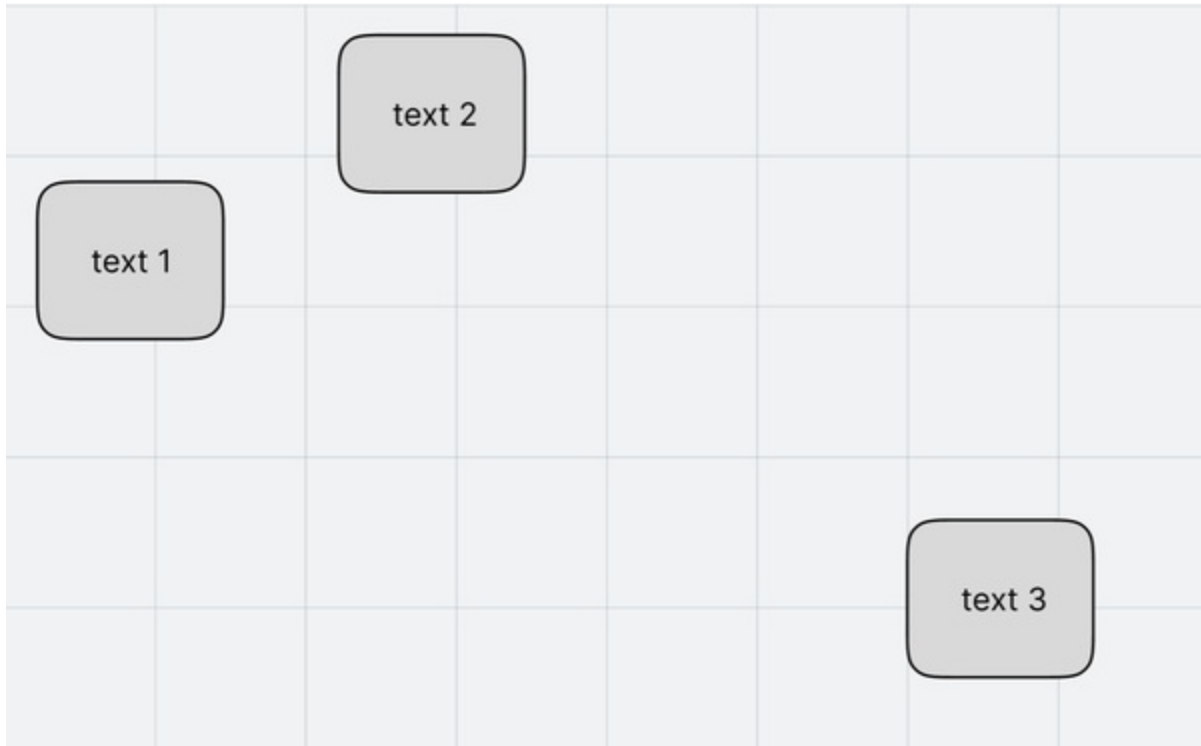


Figure 2-4. The intuition of embeddings: each text is a point, texts with similar meaning are close to each other.

This is the property that is used to build search systems. In this scenario, when a user enters a search query, we embed the query, thus projecting it into the same space as our text archive. Then we simply find the nearest documents to the query in that space, and those would be the search results.

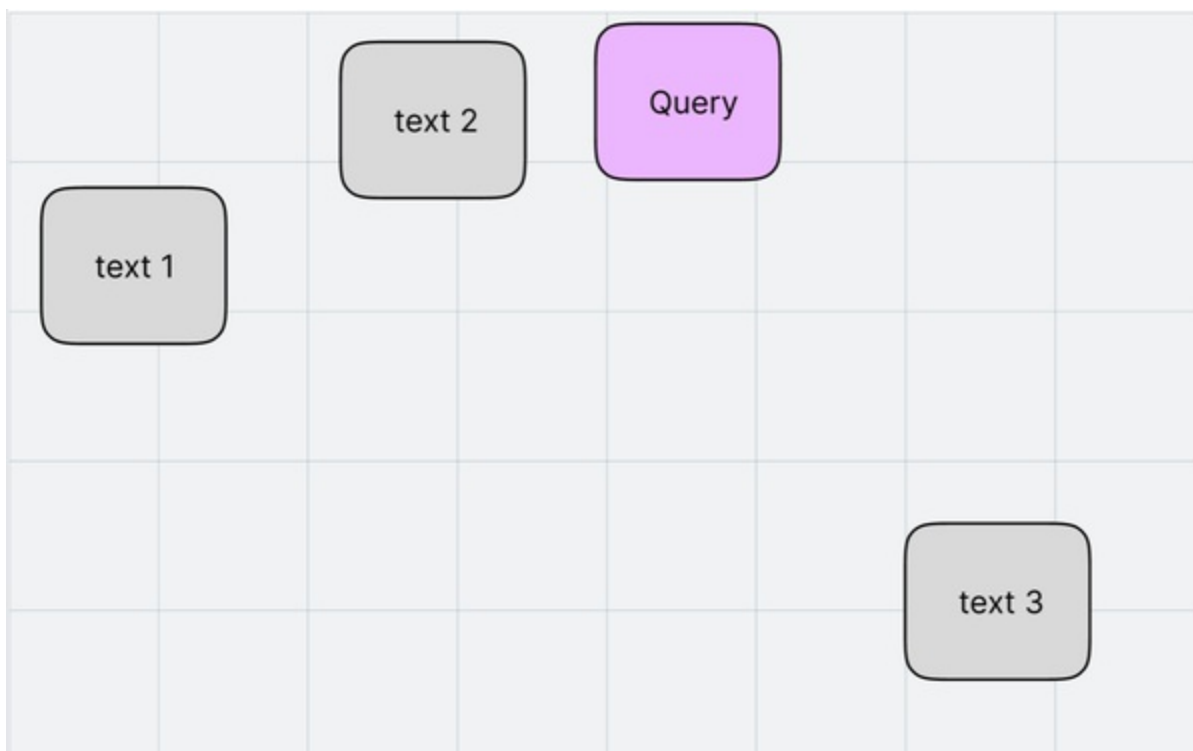


Figure 2-5. Dense retrieval relies on the property that search queries will be close to their relevant results.

Judging by the distances in [Figure 2-5](#), “text 2” is the best result for this query, followed by “text 1”. Two questions could arise here, however:

Should text 3 even be returned as a result? That’s a decision for you, the system designer. It’s sometimes desirable to have a max threshold of similarity score to filter out irrelevant results (in case the corpus has no relevant results for the query).

Are a query and its best result semantically similar? Not always. This is why language models need to be trained on question-answer pairs to become better at retrieval. This process is

explained in more detail in chapter 13.

Dense Retrieval Example

Let's take a look at a dense retrieval example by using Cohere to search the Wikipedia page for the film *Interstellar*. In this example, we will do the following:

1. Get the text we want to make searchable, apply some light processing to chunk it into sentences.
2. Embed the sentences
3. Build the search index
4. Search and see the results

To start, we'll need to install the libraries we'll need for the example:

```
# Install Cohere for embeddings, Annoy for approximate  
!pip install cohere tqdm Annoy
```

Get your Cohere API key by signing up at <https://cohere.ai/>. Paste it in the cell below. You will not have to pay anything to run through this example.

Let's import the datasets we'll need:

```
import cohere
import numpy as np
import re
import pandas as pd
from tqdm import tqdm
from sklearn.metrics.pairwise import cosine_similarity
from annoy import AnnoyIndex

# Paste your API key here. Remember to not share publicly
api_key = ''

# Create and retrieve a Cohere API key from os.cohere.
co = cohere.Client(api_key)
```

1. Getting the text Archive

Let's use the first section of the Wikipedia article on the film *Interstellar*. [https://en.wikipedia.org/wiki/Interstellar_\(film\)](https://en.wikipedia.org/wiki/Interstellar_(film)). We'll get the text, then break it into sentences.

```
text = """
Interstellar is a 2014 epic science fiction film co-
It stars Matthew McConaughey, Anne Hathaway, Jessica
Set in a dystopian future where humanity is struggli

Brothers Christopher and Jonathan Nolan wrote the sc
```

Caltech theoretical physicist and 2017 Nobel laureate
Cinematographer Hoyte van Hoytema shot it on 35 mm m
Principal photography began in late 2013 and took pl
Interstellar uses extensive practical and miniature

Interstellar premiered on October 26, 2014, in Los A
In the United States, it was first released on film
The film had a worldwide gross over \$677 million (an
It received acclaim for its performances, direction,
It has also received praise from many astronomers fo
Interstellar was nominated for five awards at the 87
Split into a list of sentences

```
texts = text.split('.')
```

```
# Clean up to remove empty spaces and new lines  
texts = np.array([t.strip(' \n') for t in texts])
```

2. Embed the texts

Let's now embed the texts. We'll send them to the Cohere API,
and get back a vector for each text.

```
# Get the embeddings  
response = co.embed(  
    texts=texts,  
)  
.embeddings  
  
embeds = np.array(response)
```

```
print(embeds.shape)
```

Which outputs:

```
(15, 4096)
```

Indicating that we have 15 vectors, each one is of size 4096.

3. Build The Search Index

Before we can search, we need to build a search index. An index stores the embeddings and is optimized to quickly retrieve the nearest neighbors even if we have a very large number of points.

```
# Create the search index, pass the size of embedding
search_index = AnnoyIndex(embeds.shape[1], 'angular')

# Add all the vectors to the search index
for index, embed in enumerate(embeds):
    search_index.add_item(index, embed)

search_index.build(10)
search_index.save('test.ann')
```

4. Search the index

We can now search the dataset using any query we want. We simply embed the query, and present its embedding to the index, which will retrieve the most similar texts.

Let's define our search function:

```
def search(query):  
  
    # 1. Get the query's embedding  
    query_embed = co.embed(texts=[query]).embeddings[0]  
  
    # 2. Retrieve the nearest neighbors  
    similar_item_ids = search_index.get_nns_by_vector(  
        query_embed, k=10, include_distances=True)  
  
    # 3. Format the results  
    results = pd.DataFrame(data={'texts': texts[similar_item_ids],  
                                'distance': similar_item_distances})  
  
    # 4. Print and return the results  
    print(f"Query: '{query}'\nNearest neighbors:")  
    return results
```

We are now ready to write a query and search the texts!

```
query = "How much did the film make?"  
search(query)
```

Which produces the output:

```
Query: 'How much did the film make?'
```

Nearest neighbors:

texts

- | | |
|---|---|
| 0 | The film had a worldwide gross over \$677 mil |
| 1 | It stars Matthew McConaughey, Anne Hathaway, |
| 2 | In the United States, it was first released |

The first result has the least distance, and so is the most similar to the query. Looking at it, it answers the question perfectly. Notice that this wouldn't have been possible if we were only doing keyword search because the top result did not include the words "much" or "make".

To further illustrate the capabilities of dense retrieval, here's a list of queries and the top result for each one:

Query: "Tell me about the \$\$\$?"

Top result: The film had a worldwide gross over \$677 million (and \$773 million with subsequent re-releases), making it the tenth-highest grossing film of 2014

Distance: 1.244138

Query: “Which actors are involved?”

Top result: It stars Matthew McConaughey, Anne Hathaway, Jessica Chastain, Bill Irwin, Ellen Burstyn, Matt Damon, and Michael Caine

Distance: 0.917728

Query: “How was the movie released?”

Top result: In the United States, it was first released on film stock, expanding to venues using digital projectors

Distance: 0.871881

Caveats of Dense Retrieval

It’s useful to be aware of some of the drawbacks of dense retrieval and how to address them. What happens, for example, if the texts don’t contain the answer? We still get results and their distances. For example:

Query: 'What is the mass of the moon?'

Nearest neighbors:

texts

0 The film had a worldwide gross over \$677 milli

1 It has also received praise from many astronom

2 Cinematographer Hoyte van Hoytema shot it on 3

In cases like this, one possible heuristic is to set a threshold level -- a maximum distance for relevance, for example. A lot of search systems present the user with the best info they can get, and leave it up to the user to decide if it's relevant or not. Tracking the information of whether the user clicked on a result (and were satisfied by it), can improve future versions of the search system.

Another caveat of dense retrieval is cases where a user wants to find an exact match to text they're looking for. That's a case that's perfect for keyword matching. That's one reason why hybrid search, which includes both semantic search and keyword search, is used.

Dense retrieval systems also find it challenging to work properly in domains other than the ones that they were trained on. So for example if you train a retrieval model on internet and Wikipedia data, and then deploy it on legal texts (without having enough

legal data as part of the training set), the model will not work as well in that legal domain.

The final thing we'd like to point out is that this is a case where each sentence contained a piece of information, and we showed queries that specifically ask those for that information. What about questions whose answers span multiple sentences? This shows one of the important design parameters of dense retrieval systems: what is the best way to chunk long texts? And why do we need to chunk them in the first place?

Chunking Long Texts

One limitation of Transformer language models is that they are limited in context sizes. Meaning we cannot feed them very long texts that go above a certain number of words or tokens that the model supports. So how do we embed long texts?

There are several possible ways, and two possible approaches shown in [Figure 2-6](#) include indexing one vector per document, and indexing multiple vectors per document.

One vector per document



Chunk document into multiple chunks



Figure 2-6. It's possible to create one vector representing an entire document, but it's better for longer documents to be split into smaller chunks that get their own embeddings.

One vector per document

In this approach, we use a single vector to represent the whole document. The possibilities here include:

- Embedding only a representative part of the document and ignoring the rest of the text. This may mean embedding only the title, or only the beginning of the document. This is useful to get quickly started with building a demo but it leaves a lot of information unindexed and so unsearchable. As an approach, it may work better for documents where the beginning captures the main points of a document (think: Wikipedia article). But it's really not the best approach for a real system.

- Embedding the document in chunks, embedding those chunks, and then aggregating those chunks into a single vector. The usual method of aggregation here is to average those vectors. A downside of this approach is that it results in a highly compressed vector that loses a lot of the information in the document.

This approach can satisfy some information needs, but not others. A lot of the time, a search is for a specific piece of information contained in an article, which is better captured if the concept had its own vector.

Multiple vectors per document

In this approach, we chunk the document into smaller pieces, and embed those chunks. Our search index then becomes that of chunk embeddings, not entire document embeddings.

The chunking approach is better because it has full coverage of the text and because the vectors tend to capture individual concepts inside the text. This leads to a more expressive search index. Figure X-3 shows a number of possible approaches.

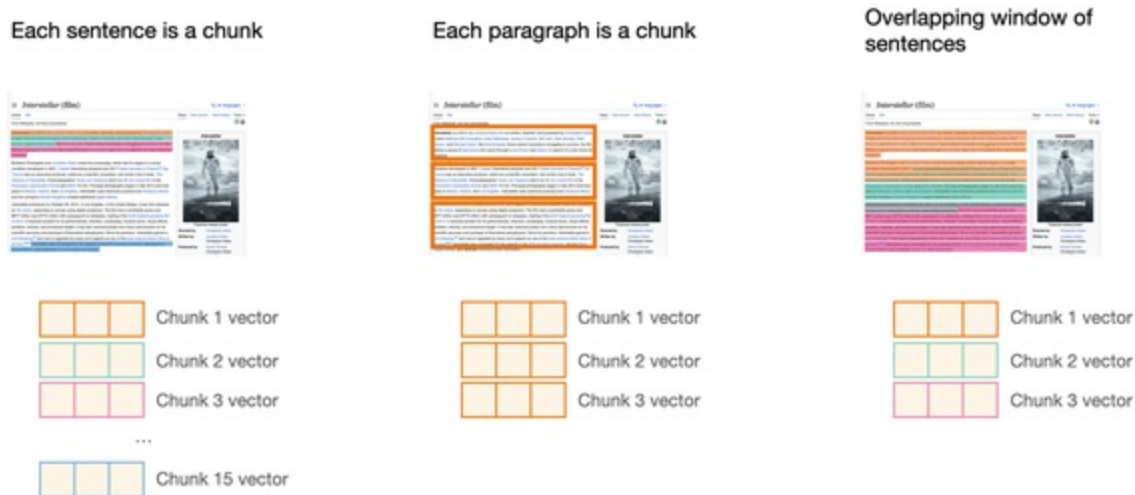


Figure 2-7. A number of possible options for chunking a document for embedding.

The best way of chunking a long text will depend on the types of texts and queries your system anticipates. Approaches include:

- Each sentence is a chunk. The issue here is this could be too granular and the vectors don't capture enough of the context.
- Each paragraph is a chunk. This is great if the text is made up of short paragraphs. Otherwise, it may be that every 4-8 sentences are a chunk.
- Some chunks derive a lot of their meaning from the text around them. So we can incorporate some context via:
 - Adding the title of the document to the chunk
 - Adding some of the text before and after them to the chunk.
 This way, the chunks can overlap so they include some surrounding text. This is what we can see in [Figure 2-8](#).



Figure 2-8. Chunking the text into overlapping segments is one strategy to retain more of the context around different segments.

Expect more chunking strategies to arise as the field develops -- some of which may even use LLMs to dynamically split a text into meaningful chunks.

Nearest Neighbor Search vs. Vector Databases

The most straightforward way to find the nearest neighbors is to calculate the distances between the query and the archive. That can easily be done with NumPy and is a reasonable approach if you have thousands or tens of thousands of vectors in your archive.

As you scale beyond to the millions of vectors, an optimized approach for the retrieval is to rely on approximate nearest neighbor search libraries like Annoy or FAISS. These allow you to retrieve results from massive indexes in milliseconds and some

of them can scale to GPUs and clusters of machines to serve very large indices.

Another class of vector retrieval systems are vector databases like Weaviate or Pinecone. A vector database allows you to add or delete vectors without having to rebuild the index. They also provide ways to filter your search or customize it in ways beyond merely vector distances.

Fine-tuning embedding models for dense retrieval

Just like we've seen in the text classification chapter, we can improve the performance of an LLM on a task using fine-tuning. Just like in that case, retrieval needs to optimize text embeddings and not simply token embeddings. The process for this finetuning is to get training data composed of queries and relevant results.

Looking at one example from our dataset, the sentence "Interstellar premiered on October 26, 2014, in Los Angeles.". Two possible queries where this is a relevant result are:

- Relevant Query 1: "Interstellar release date"
- Relevant Query 2: "When did Interstellar premier"

The fine-tuning process aims to make the embeddings of these queries close to the embedding of the resulting sentence. It also needs to see negative examples of queries that are not relevant to the sentence, for example.

- Irrelevant Query: “Interstellar cast”

Having these examples, we now have three pairs - two positive pairs and one negative pair. Let’s assume, as we can see in [Figure 2-9](#), that before fine-tuning, all three queries have the same distance from the result document. That’s not far-fetched because they all talk about Interstellar.

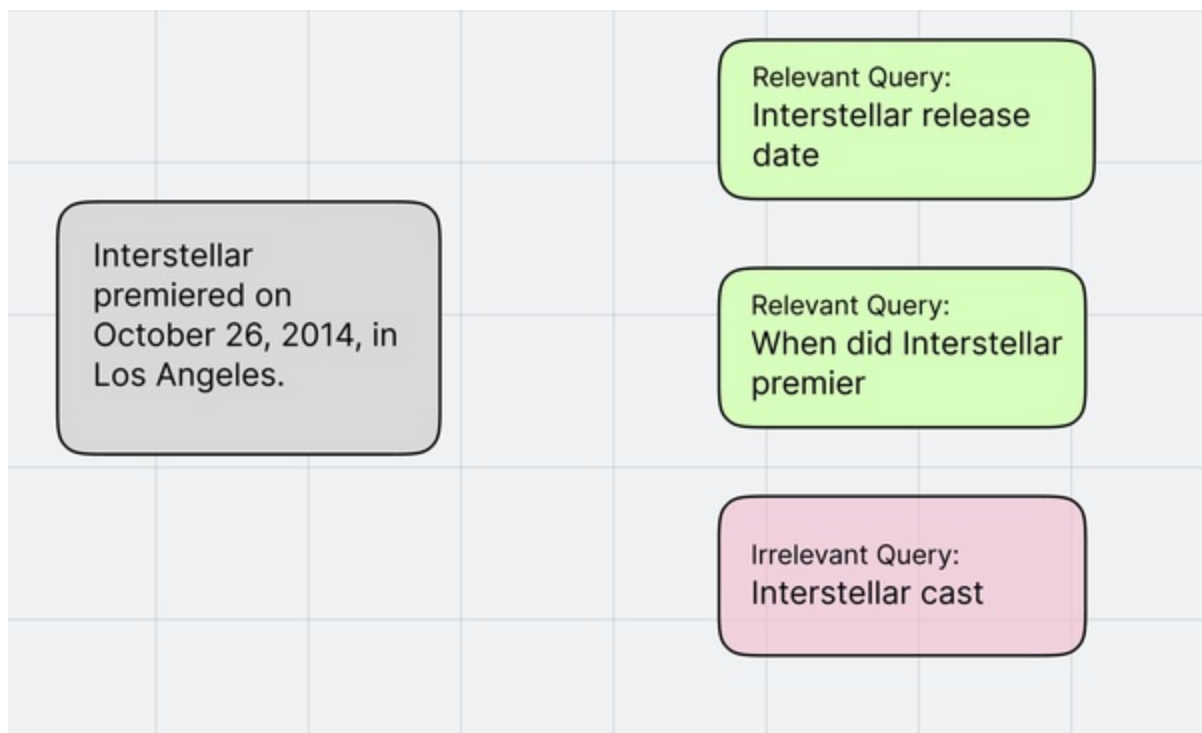


Figure 2-9. Before fine-tuning, the embeddings of both relevant and irrelevant queries may be close to a particular document.

The fine-tuning step works to make the relevant queries closer to the document and at the same time making irrelevant queries farther from the document. We can see this effect in [Figure 2-10](#).

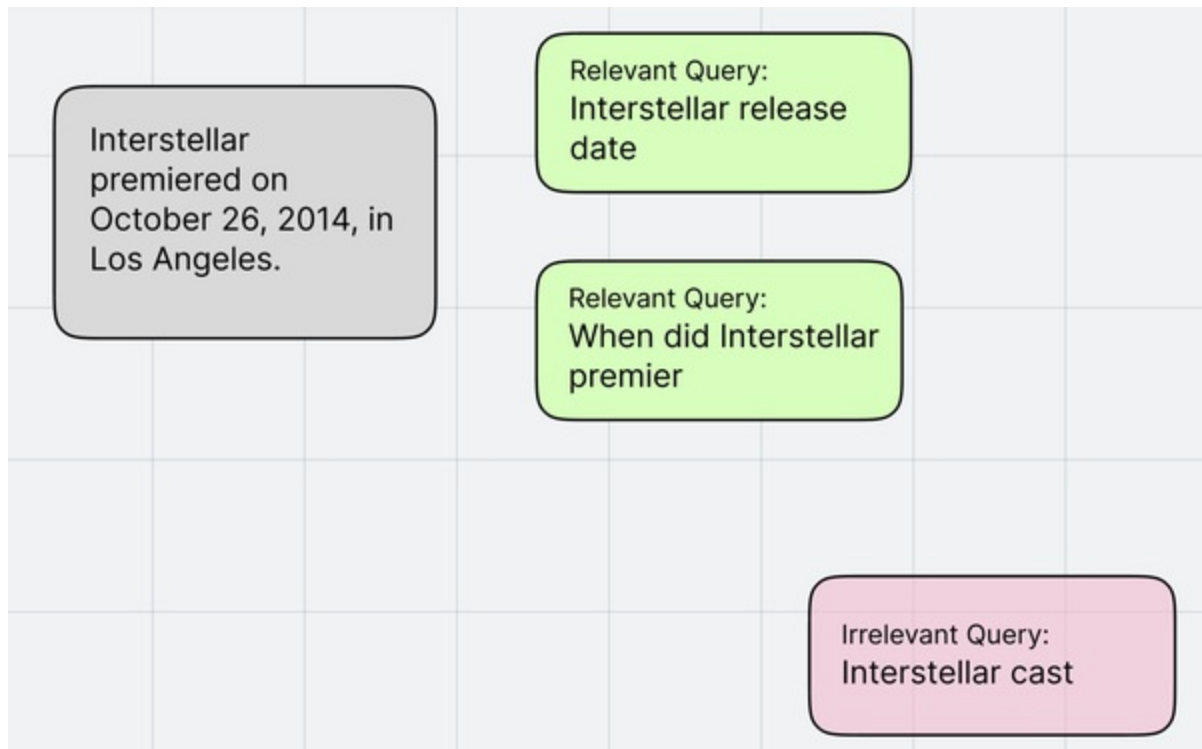


Figure 2-10. After the fine-tuning process, the text embedding model becomes better at this search task by incorporating how we define relevance on our dataset using the examples we provided of relevant and irrelevant documents.

Reranking

A lot of companies have already built search systems. For those companies, an easier way to incorporate language models is as a final step inside their search pipeline. This step is tasked with changing the order of the search results based on relevance to

the search query. This one step can vastly improve search results and it's in fact what Microsoft Bing added to achieve the improvements to the search results using BERT-like models.

[Figure 2-11](#) shows the structure of a rerank search system serving as the second stage in a two-stage search system.

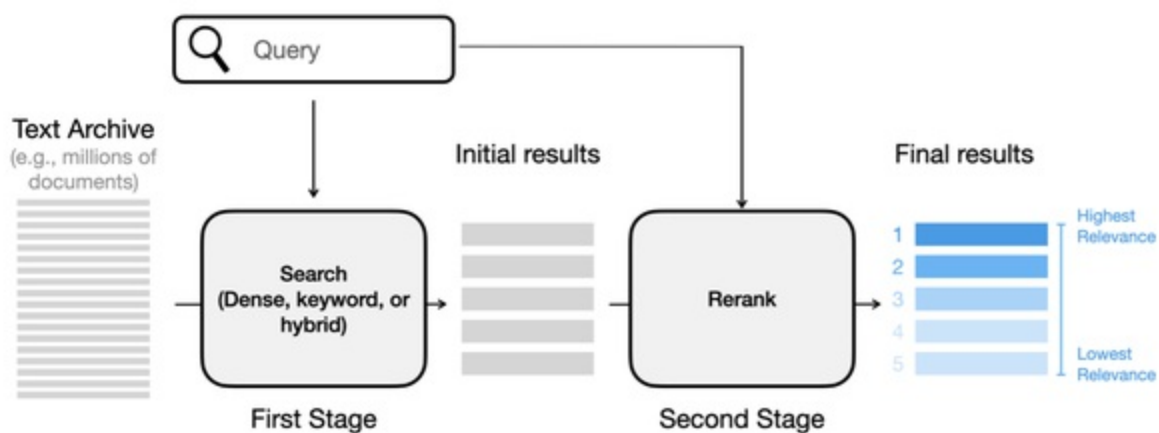


Figure 2-11. LLM Rerankers operate as a part of a search pipeline with the goal of re-ordering a number of shortlisted search results by relevance

Reranking Example

A reranker takes in the search query and a number of search results, and returns the optimal ordering of these documents so the most relevant ones to the query are higher in ranking.

```
import cohere as co
API_KEY = ""
co = cohere.Client(API_KEY)
```

```
MODEL_NAME = "rerank-english-02" # another option is r

query = "film gross"
```

Cohere's [Rerank](#) endpoint is a simple way to start using a first reranker. We simply pass it the query and texts, and get the results back. We don't need to train or tune it.

```
results = co.rerank(query=query, model=MODEL_NAME, doc
```

We can print these results:

```
results = co.rerank(query=query, model=MODEL_NAME, doc
for idx, r in enumerate(results):
    print(f"Document Rank: {idx + 1}, Document Index: {r
    print(f"Document: {r.document['text']}")
    print(f"Relevance Score: {r.relevance_score:.2f}")
    print("\n")
```

Output:

```
Document Rank: 1, Document Index: 10
Document: The film had a worldwide gross over $677 mil
Relevance Score: 0.92
```

Document Rank: 2, Document Index: 12

Document: It has also received praise from many astron
Relevance Score: 0.11

Document Rank: 3, Document Index: 2

Document: Set in a dystopian future where humanity is
Relevance Score: 0.03

This shows the reranker is much more confident about the first result, assigning it a relevance score of 0.92 while the other results are scored much lower in relevance.

More often, however, our index would have thousands or millions of entries, and we need to shortlist, say one hundred or one thousand results and then present those to the reranker. This shortlisting is called the *first stage* of the search pipeline.

The dense retriever example we looked at in the previous section is one possible first-stage retriever. In practice, the first stage can also be a search system that incorporates both keyword search as well as dense retrieval.

Open Source Retrieval and Reranking with Sentence Transformers

If you want to locally setup retrieval and reranking on your own machine, then you can use the Sentence Transformers library. Refer to the documentation in <https://www.sbert.net/> for setup. Check the [*Retrieve & Re-Rank section*](#) for instructions and code examples for how to conduct these steps in the library.

How Reranking Models Work

One popular way of building LLM search rerankers present the query and each result to an LLM working as a *cross-encoder*. Meaning that a query and possible result are presented to the model at the same time allowing the model to view the full text of both these texts before it assigns a relevance score. This method is described in more detail in a paper titled [*Multi-Stage Document Ranking with BERT*](#) and is sometimes referred to as monoBERT.

This formulation of search as relevance scoring basically boils down to being a classification problem. Given those inputs, the model outputs a score from 0-1 where 0 is irrelevant and 1 is highly relevant. This should be familiar from looking at the Classification chapter.

To learn more about the development of using LLMs for search, [*Pretrained Transformers for Text Ranking: BERT and Beyond*](#) is a highly recommended look at the developments of these models until about 2021.

Generative Search

You may have noticed that dense retrieval and reranking both use representation language models, and not generative language models. That's because they're better optimized for these tasks than generative models.

At a certain scale, however, generative LLMs started to seem more and more capable of a form of useful information retrieval. People started asking models like ChatGPT questions and sometimes got relevant answers. The media started painting this as a threat to Google which seems to have started an arms race in using language models for search. Microsoft [*launched*](#) Bing AI, powered by generative models. Google launched [*Bard*](#), its own answer in this space.

What is Generative Search?

Generative search systems include a text generation step in the search pipeline. At the moment, however, generative LLMs aren't

reliable information retrievers and are prone to generate coherent, yet often incorrect, text in response to questions they don't know the answer to.

The first batch of generative search systems is using search models as simply a summarization step at the end of the search pipeline. We can see an example in [Figure 2-12](#).

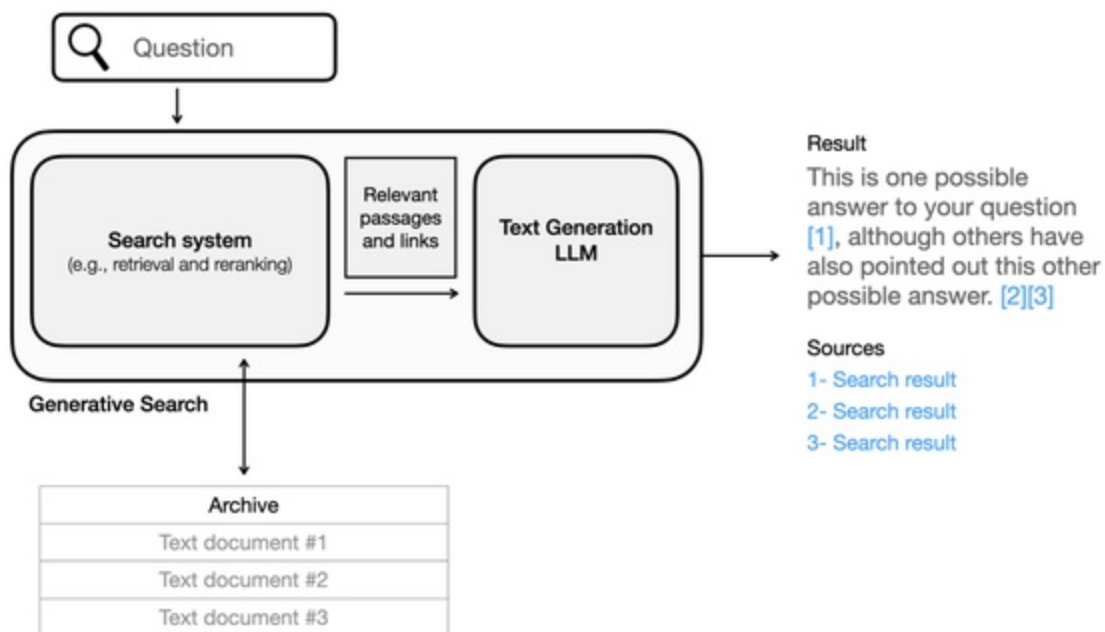


Figure 2-12. Generative search formulates answers and summaries at the end of a search pipeline while citing its sources (returned by the previous steps in the search system).

Until the time of this writing, however, language models excel at generating coherent text but they are not reliable in retrieving facts. They don't yet really know what they know or don't know, and tend to answer lots of questions with coherent text that can

be incorrect. This is often referred to as *hallucination*. Because of it, and for the fact that search is a use case that often relies on facts or referencing existing documents, generative search models are trained to cite their sources and include links to them in their answers.

Generative search is still in its infancy and is expected to improve with time. It draws from a machine learning research area called retrieval-augmented generation. Notable systems in the field include [RAG](#), [RETRO](#), [Atlas](#), amongst others.

Other LLM applications in search

In addition to these three categories, there are plenty of other ways to use LLMs to power or improve search systems. Examples include:

- Generating synthetic data to improve embedding models. This includes methods like [GenQ](#) and [InPars-v2](#) that look at documents, generate possible queries and questions about those documents, then use that generated data to fine-tune a retrieval system.
- The growing reasoning capabilities of text generation models are leading to search systems that can tackle complex

questions and queries by breaking them down into multiple sub-queries that are tackled in sequence, leading up to a final answer of the original question. One method in this category is described in *Demonstrate-Search-Predict: Composing retrieval and language models for knowledge-intensive NLP*.

Evaluation metrics

Semantic search is evaluated using metrics from the Information Retrieval (IR) field. Let's discuss two of these popular metrics: Mean Average Precision (MAP), and Normalized Discounted Cumulative Gain (nDCG).

Evaluating search systems needs three major components, a text archive, a set of queries, and relevance judgments indicating which documents are relevant for each query. We see these components in Figure 3-13.

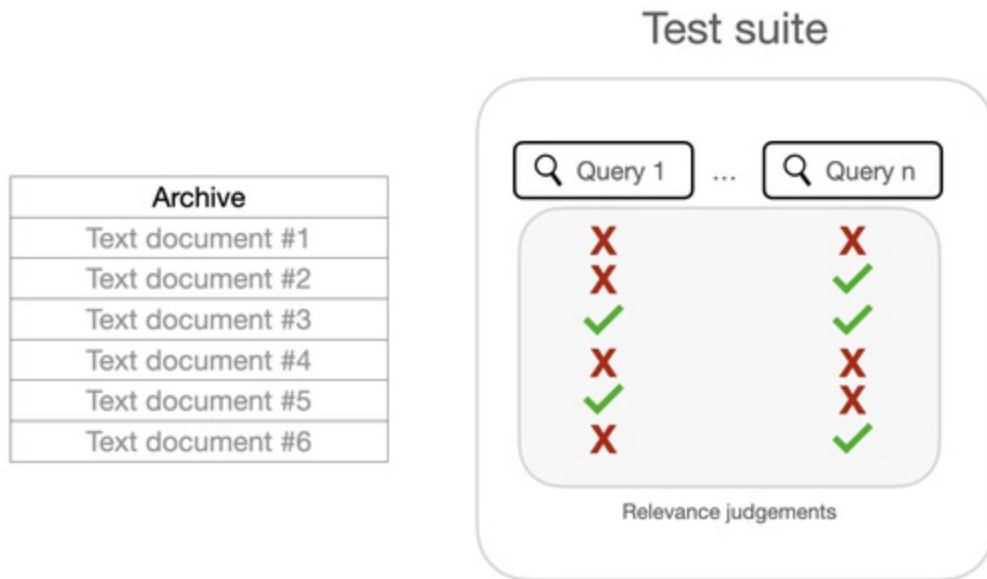


Figure 2-13. To evaluate search systems, we need a test suite including queries and relevance judgements indicating which documents in our archive are relevant for each query.

Using this test suite, we can proceed to explore evaluating search systems. Let's start with a simple example, let's assume we pass Query 1 to two different search systems. And get two sets of results. Say we limit the number of results to three results only as we can see in [Figure 2-14](#).

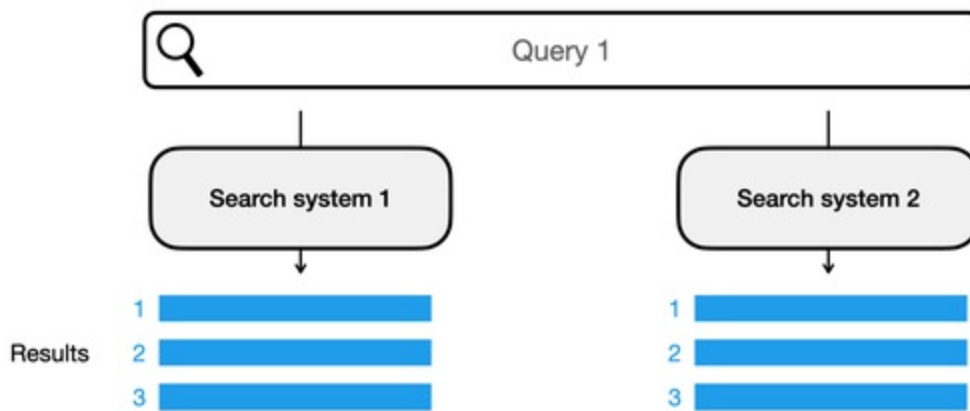


Figure 2-14. To compare two search systems, we pass the same query from our test suite to both systems and look at their top results

To tell which is a better system, we turn the relevance judgments that we have for the query. [Figure 2-15](#) shows which of the returned results are relevant.



Figure 2-15. Looking at the relevance judgements from our test suite, we can see that System 1 did a better job than System 2.

This shows us a clear case where system 1 is better than system

2. Intuitively, we may just count how many relevant results each system retrieved. System A got two out of three correctly, and System 2 got only one out of three correctly.

But what about a case like Figure 3-16 where both systems only get one relevant result out of three, but they're in different positions.



Figure 2-16. We need a scoring system that rewards system 1 for assigning a high position to a relevant result -- even though both systems retrieved only one relevant result in their top three results.

In this case, we can intuit that System 1 did a better job than system 2 because the result in the first position (the most important position) is correct. But how can we assign a number or score to how much better that result is? Mean Average Precision is a measure that is able to quantify this distinction.

One common way to assign numeric scores in this scenario is

Average Precision, which evaluates System 1's result for the query to be 0.6 and System 2's to be 0.1. So let's see how Average Precision is calculated to evaluate one set of results, and then how it's aggregated to evaluate a system across all the queries in the test suite.

Mean Average Precision (MAP)

To score system 1 on this query, we need to calculate multiple scores first. Since we are looking at only three results, we'll need to look at three scores - one associated with each position.

The first one is easy, looking at only the first result, we calculate the precision score: we divide the number of correct results by the total number of results (correct and incorrect). [Figure 2-17](#) shows that in this case, we have one correct result out of one (since we're only looking at the first position now). So precision here is $1/1 = 1$.



Figure 2-17. To calculate Mean Average Precision, we start by calculating precision at

each position, starting by position #1.

We need to continue calculating precision results for the rest of the position. The calculation at the second position looks at both the first and second position. The precision score here is 1 (one out of two results being correct) divided by 2 (two results we're evaluating) = 0.5.

[Figure 2-18](#) continues the calculation for the second and third positions. It then goes one step further -- having calculated the precision for each position, we average them to arrive at an Average Precision score of 0.61.



Figure 2-18. Caption to come

This calculation shows the average precision for a single query and its results. If we calculate the average precision for System 1 on all the queries in our test suite and get their mean, we arrive at the Mean Average Precision score that we can use to compare System 1 to other systems across all the queries in our test suite.

Summary

In this chapter, we looked at different ways of using language models to improve existing search systems and even be the core of new, more powerful search systems. These include:

- Dense retrieval, which relies on the similarity of text embeddings. These are systems that embed a search query and retrieve the documents with the nearest embeddings to the query's embedding.
- Rerankers, systems (like monoBERT) that look at a query and candidate results, and scores the relevance of each document to that query. These relevance scores are then used to order the shortlisted results according to their relevance to the query often producing an improved results ranking.
- Generative search, where search systems that have a generative LLM at the end of the pipeline to formulate an answer based on retrieved documents while citing its sources.

We also looked at one of the possible methods of evaluating search systems. Mean Average Precision allows us to score search systems to be able to compare across a test suite of queries and their known relevance to the test queries.

Chapter 3. Text Clustering and Topic Modeling

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. *In particular, some of the formatting may not match the description in the text: this will be resolved when the book is finalized.*

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Although supervised techniques, such as classification, have reigned supreme over the last few years in the industry, the potential of unsupervised techniques such as text clustering cannot be understated.

Text clustering aims to group similar texts based on their semantic content, meaning, and relationships, as illustrated in [Figure 3-1](#). Just like how we've used distances between text embeddings in dense retrieval in chapter XXX, clustering embeddings allow us to group the documents in our archive by similarity.

The resulting clusters of semantically similar documents not only facilitate efficient categorization of large volumes of unstructured text but also allows for quick exploratory data analysis. With the advent of Large Language Models (LLMs) allowing for contextual and semantic representations of text, the power of text clustering has grown significantly over the last years. Language is not a bag of words, and Large Language Models have proved to be quite capable of capturing that notion.

An underestimated aspect of text clustering is its potential for creative solutions and implementations. In a way, unsupervised means that we are not constrained by a certain task or thing that we want to optimize. As a result, there is much freedom in text clustering that allows us to steer from the well-trodden paths. Although text clustering would naturally be used for grouping and classifying documents, it can be used to algorithmically and visually find improper labels, perform topic modeling, speed up labeling, and many more interesting use cases.

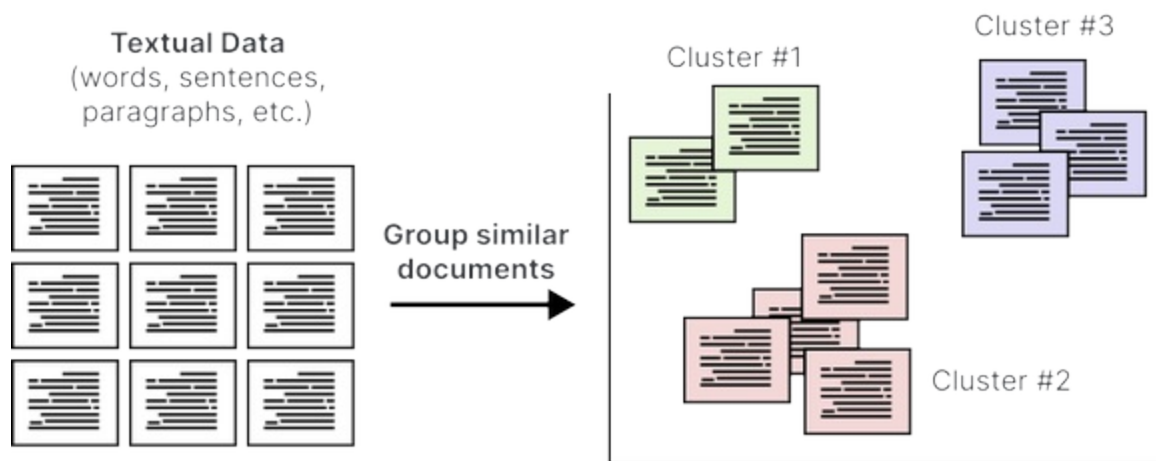


Figure 3-1. Clustering unstructured textual data.

This freedom also comes with its challenges. Since we are not guided by a specific task, then how do we evaluate our unsupervised clustering output? How do we optimize our algorithm? Without labels, what are we optimizing the algorithm for? When do we know our algorithm is correct? What does it mean for the algorithm to be “correct”? Although these challenges can be quite complex, they are not insurmountable but often require some creativity and a good understanding of the use case.

Striking a balance between the freedom of text clustering and the challenges it brings can be quite difficult. This becomes even more pronounced if we step into the world of topic modeling, which has started to adopt the “text clustering” way of thinking.

With topic modeling, we want to discover abstract topics that

appear in large collections of textual data. We can describe a topic in many ways, but it has traditionally been described by a set of keywords or key phrases. A topic about natural language processing (NLP) could be described with terms such as “deep learning”, “transformers”, and “self-attention”. Traditionally, we expect a document about a specific topic to contain terms appearing more frequently than others. This expectation, however, ignores contextual information that a document might contain. Instead, we can leverage Large Language Models, together with text clustering, to model contextualized textual information and extract semantically-informed topics. [Figure 3-2](#) demonstrates this idea of describing clusters through textual representations.

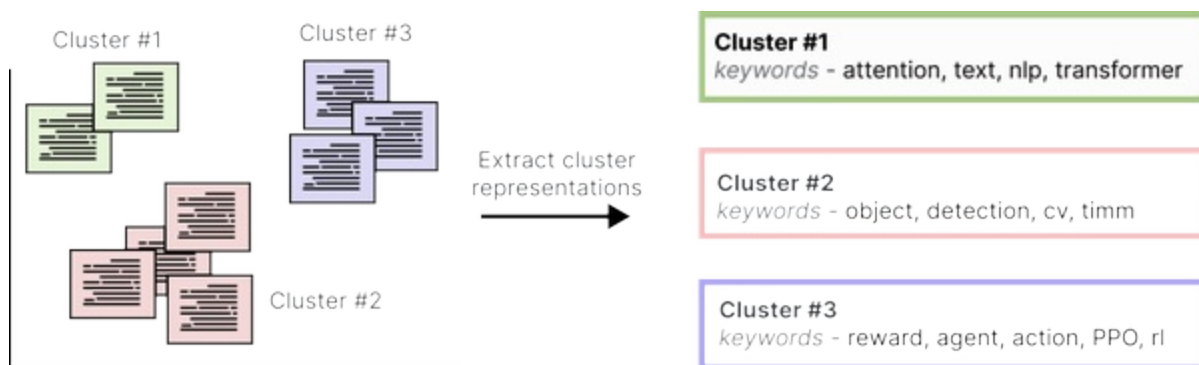


Figure 3-2. Topic modeling is a way to give meaning to clusters of textual documents.

In this chapter, we will provide a guide on how text clustering can be done with Large Language Models. Then, we will

transition into a text-clustering-inspired method of topic modeling, namely BERTopic.

Text Clustering

One major component of exploratory data analysis in NLP is text clustering. This unsupervised technique aims to group similar texts or documents together as a way to easily discover patterns among large collections of textual data. Before diving into a classification task, text clustering allows for getting an intuitive understanding of the task but also of its complexity.

The patterns that are discovered from text clustering can be used across a variety of business use cases. From identifying recurring support issues and discovering new content to drive SEO practices, to detecting topic trends in social media and discovering duplicate content. The possibilities are diverse and with such a technique, creativity becomes a key component. As a result, text clustering can become more than just a quick method for exploratory data analysis.

Data

Before we describe how to perform text clustering, we will first introduce the data that we are going to be using throughout this

chapter. To keep up with the theme of this book, we will be clustering a variety of ArXiv articles in the domain of machine learning and natural language processing. The dataset contains roughly **XXX** articles between **XXX** and **XXX**.

We start by importing our dataset using [HuggingFace's dataset package](#) and extracting metadata that we are going to use later on, like the abstracts, years, and categories of the articles.

```
# Load data from huggingface
from datasets import load_dataset
dataset = load_dataset("maartengr/arxiv_nlp")["train"]

# Extract specific metadata
abstracts = dataset["Abstracts"]
years = dataset["Years"]
categories = dataset["Categories"]
titles = dataset["Titles"]
```

How do we perform Text Clustering?

Now that we have our data, we can perform text clustering. To perform text clustering, a number of techniques can be employed, from graph-based neural networks to centroid-based clustering techniques. In this section, we will go through a well-

known pipeline for text clustering that consists of three major steps:

1. Embed documents
2. Reduce dimensionality
3. Cluster embeddings

1. Embed documents

The first step in clustering textual data is converting our textual data to text embeddings. Recall from previous chapters that embeddings are numerical representations of text that capture its meaning. Producing embeddings optimized for semantic similarity tasks is especially important for clustering. By mapping each document to a numerical representation such that semantically similar documents are close, clustering will become much more powerful. A set of popular Large Language Models optimized for these kinds of tasks can be found in the well-known sentence-transformers framework (reimers2019sentence). [Figure 3-3](#) shows this first step of converting documents to numerical representations.

Embed documents

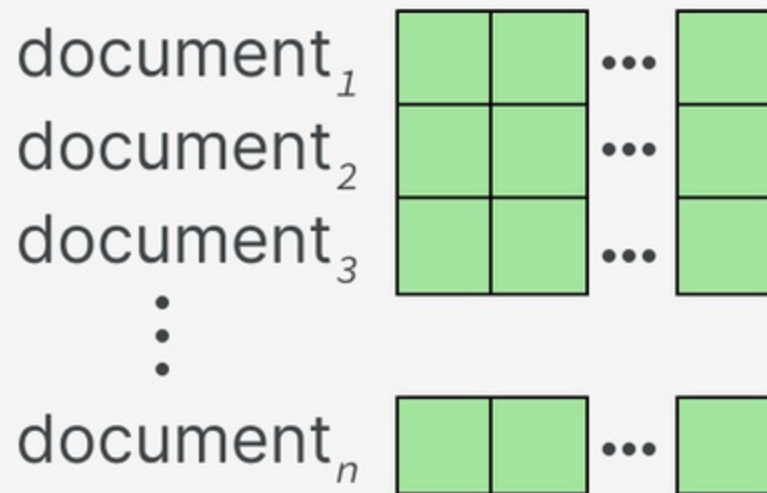


Figure 3-3. Step 1: We convert documents to numerical representations, namely embeddings.

Sentence-transformers has a clear API and can be used as follows to generate embeddings from pieces of text:

```
from sentence_transformers import SentenceTransformer

# We load our model
embedding_model = SentenceTransformer('all-MiniLM-L6-v2')

# The abstracts are converted to vector representation
embeddings = embedding_model.encode(abstracts)
```

The sizes of these embeddings differ depending on the model but typically contain at least 384 values for each sentence or paragraph. The number of values an embedding contains is referred to as the dimensionality of the embedding.

2. Reduce dimensionality

Before we cluster the embeddings we generated from the ArXiv abstracts, we need to take care of the curse of dimensionality first. This curse is a phenomenon that occurs when dealing with high-dimensional data. As the number of dimensions increases, there is an exponential growth of the number of possible values within each dimension. Finding all subspaces within each dimension becomes increasingly complex. Moreover, as the number of dimensions grows, the concept of distance between points becomes increasingly less precise.

As a result, high-dimensional data can be troublesome for many clustering techniques as it gets more difficult to identify meaningful clusters. Clusters are more diffuse and less distinguishable, making it difficult to accurately identify and separate them.

The previously generated embeddings are high in their dimensionality and often trigger the curse of dimensionality. To

prevent their dimensionality from becoming an issue, the second step in our clustering pipeline is dimensionality reduction, as shown in [Figure 3-4](#).

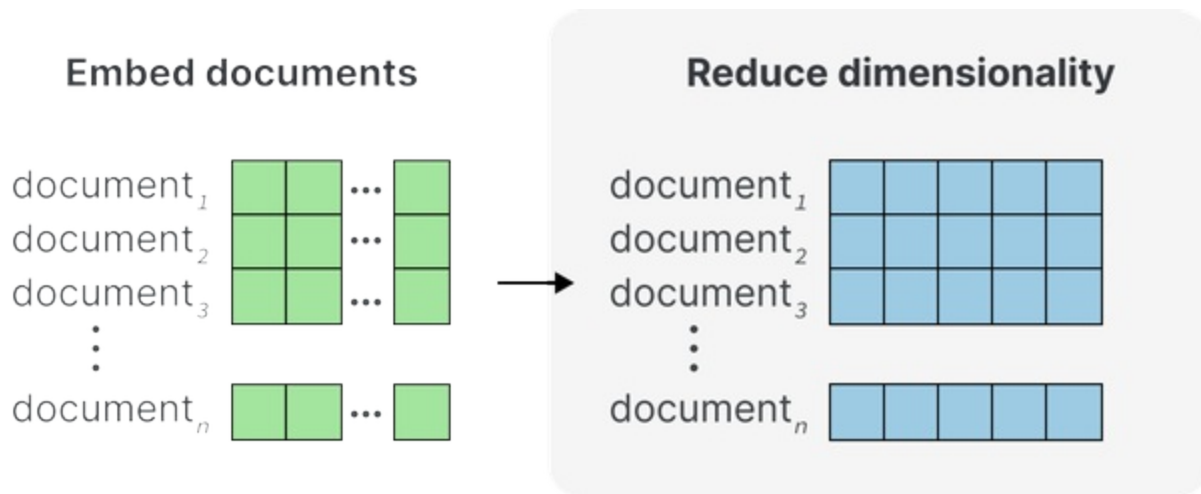


Figure 3-4. Step 2: The embeddings are reduced to a lower dimensional space using dimensionality reduction.

Dimensionality reduction techniques aim to preserve the global structure of high-dimensional data by finding low-dimensional representations. Well-known methods are Principal Component Analysis (PCA) and Uniform Manifold Approximation and Projection (UMAP; mcinnes2018umap). For this pipeline, we are going with UMAP as it tends to handle non-linear relationships and structures a bit better than PCA.

NOTE

Dimensionality reduction techniques, however, are not flawless. They cannot perfectly capture high-dimensional data in a lower-dimensional representation. Information

will always be lost with this procedure. There is a balance between reducing dimensionality and keeping as much information as possible.

To perform dimensionality reduction, we need to instantiate our UMAP class and pass the generated embeddings to it:

```
from umap import UMAP

# We instantiate our UMAP model
umap_model = UMAP(n_neighbors=15, n_components=5, min_

# We fit and transform our embeddings to reduce them
reduced_embeddings = umap_model.fit_transform(embeddin
```

We can use the `n_components` parameter to decide the shape of the lower-dimensional space. Here, we used `n_components=5` as we want to retain as much information as possible without running into the curse of dimensionality. No one value does this better than another, so feel free to experiment!

3. Cluster embeddings

As shown in [Figure 3-5](#), the final step in our pipeline is to cluster the previously reduced embeddings. Many algorithms out there

handle clustering tasks quite well, from centroid-based methods like k-Means to hierarchical methods like Agglomerative Clustering. The choice is up to the user and is highly influenced by the respective use case. Our data might contain some noise, so a clustering algorithm that detects outliers would be preferred. If our data comes in daily, we might want to look for an online or incremental approach instead to model if new clusters were created.

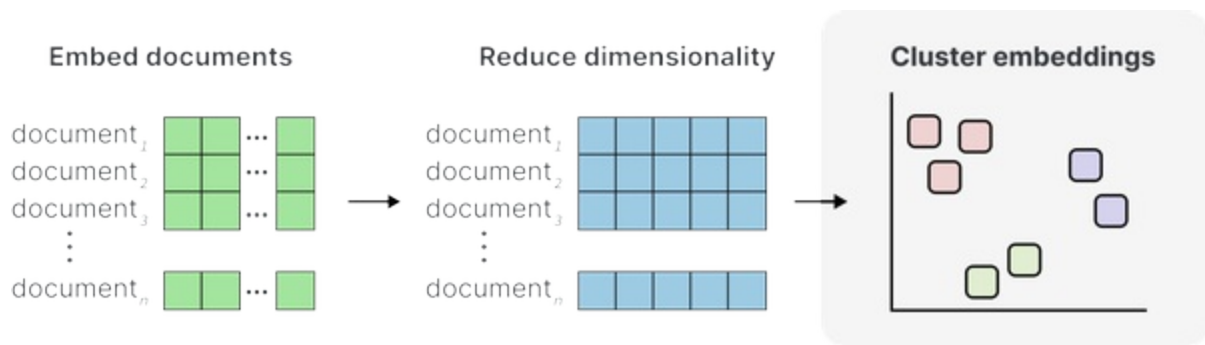


Figure 3-5. Step 3: We cluster the documents using the embeddings that were reduced in their dimensionality.

A good default model is Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN; mcinnes2017hdbscan). HDBSCAN is a hierarchical variation of a clustering algorithm called DBSCAN which allows for dense (micro)-clusters to be found without us having to explicitly specify the number of clusters. As a density-based method, it can also detect outliers in the data. Data points that do not belong to any cluster. This is important as forcing data into clusters might

create noisy aggregations.

As with the previous packages, using HDBSCAN is straightforward. We only need to instantiate the model and pass our reduced embeddings to it:

```
from hdbscan import HDBSCAN

# We instantiate our HDBSCAN model
hdbscan_model = HDBSCAN(min_cluster_size=15, metric='e

# We fit our model and extract the cluster labels
hdbscan_model.fit(reduced_embeddings)
labels = hdbscan_model.labels_
```

Then, using our previously generated 2D-embeddings, we can visualize how HDBSCAN has clustered our data:

```
import seaborn as sns

# Reduce 384-dimensional embeddings to 2 dimensions fo
reduced_embeddings = UMAP(n_neighbors=15, n_components
min_dist=0.0, metric='cosine').fit_transform(embedding
df = pd.DataFrame(np.hstack([reduced_embeddings, clust
                        columns=["x", "y", "cluster"])).sort_values("clust
```

```
# Visualize clusters
df.cluster = df.cluster.astype(int).astype(str)
sns.scatterplot(data=df, x='x', y='y', hue='cluster',
               linewidth=0, legend=False, s=3, alpha=0.3)
```

As we can see in [Figure 3-6](#), it tends to capture major clusters quite well. Note how clusters of points are colored in the same color, indicating that HDBSCAN put them in a group together. Since we have a large number of clusters, the plotting library cycles the colors between clusters, so don't think that all blue points are one cluster, for example.

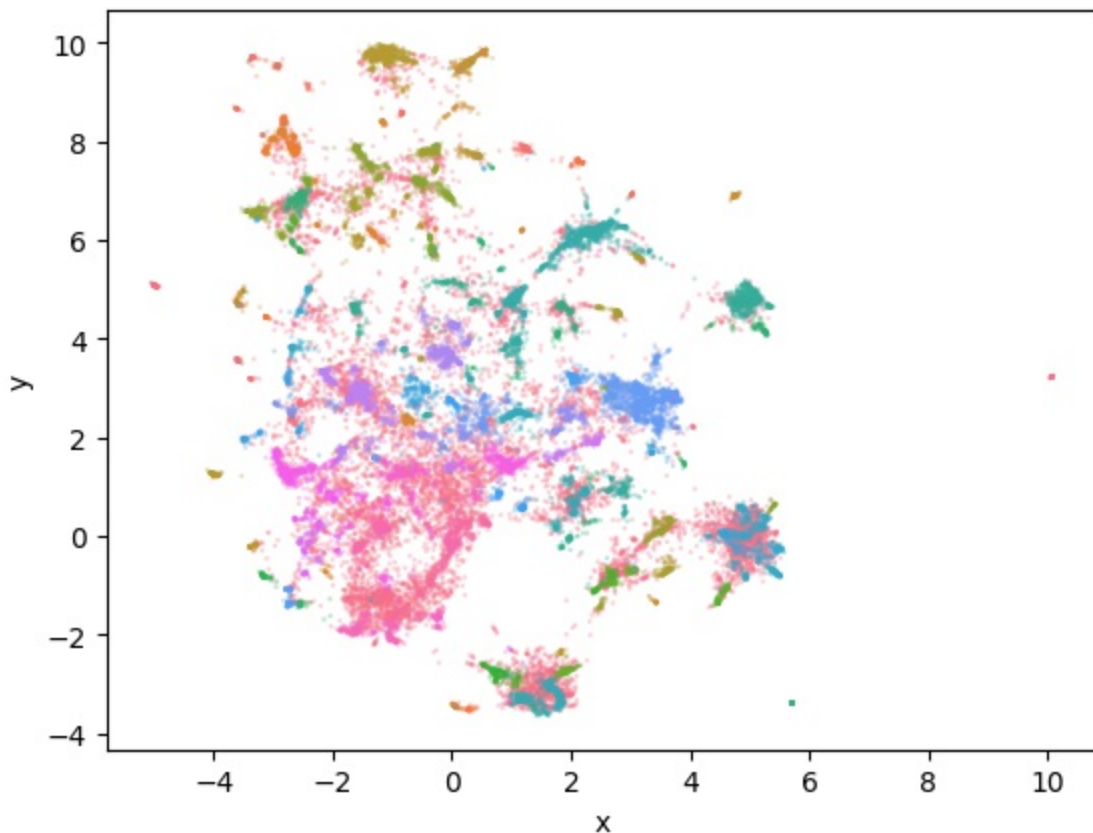


Figure 3-6. The generated clusters (colored) and outliers (grey) are represented as a 2D visualization.

NOTE

Using any dimensionality reduction technique for visualization purposes creates information loss. It is merely an approximation of what our original embeddings look like. Although it is informative, it might push clusters together and drive them further apart than they actually are. Human evaluation, inspecting the clusters ourselves, is, therefore, a key component of cluster analysis!

We can inspect each cluster manually to see which documents are semantically similar enough to be clustered together. For example, let us take a few random documents from cluster **XXX**:

```
>>> for index in np.where(labels==1)[0][:3]:
>>>     print(abstracts[index])
```

```
Sarcasm is considered one of the most difficult probl
analysis. In our ob-servation on Indonesian social med
people tend to criticize something using sarcasm. Here
additional features to detect sarcasm after a common s
con...
```

```
Automatic sarcasm detection is the task of predictin
is a crucial step to sentiment analysis, considering p
of sarcasm in sentiment-bearing text. Beginning with a
speech-based features, sarcasm detection has witnessed
```

We introduce a deep neural network for automated sarcasm detection. Previous work has emphasized the need for models to capitalize on semantic cues beyond lexical and syntactic cues present in utterance context. Speakers will tend to employ sarcasm regarding differences between expected and actual outcomes.

These printed documents tell us that the cluster likely contains documents that talk about **XXX**. We can do this for every created cluster out there but that can be quite a lot of work, especially if we want to experiment with our hyperparameters. Instead, we would like to create a method for automatically extracting representations from these clusters without us having to go through all documents.

This is where topic modeling comes in. It allows us to model these clusters and give singular meaning to them. Although there are many techniques out there, we choose a method that builds upon this clustering philosophy as it allows for significant flexibility.

Topic Modeling

Traditionally, topic modeling is a technique that aims to find latent topics or themes in a collection of textual data. For each topic, a set of keywords or phrases are identified that best

represent and capture the meaning of the topic. This technique is ideal for finding common themes in large corpora as it gives meaning to sets of similar content. An illustrated overview of topic modeling in practice can be found in [Figure 3-7](#).

Latent Dirichlet Allocation (LDA; blei2003latent) is a classical and popular approach to topic modeling that assumes that each topic is characterized by a probability distribution over words in a corpus vocabulary. Each document is to be considered a mixture of topics. For example, a document about Large Language Models might have a high probability of containing words like “BERT”, “self-attention”, and “transformers”, while a document about reinforcement learning might have a high probability of containing words like “PPO”, “reward”, “rlhf”.

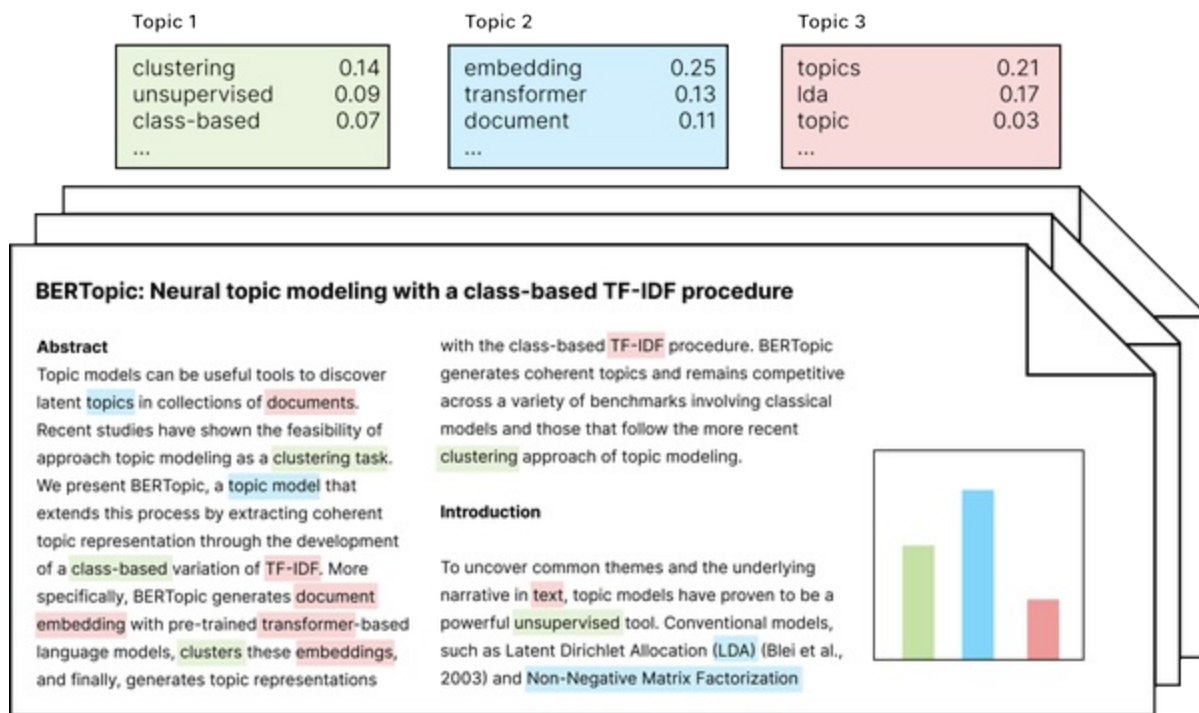


Figure 3-7. An overview of traditional topic modeling.

To this day, the technique is still a staple in many topic modeling use cases, and with its strong theoretical background and practical applications, it is unlikely to go away soon. However, with the seemingly exponential growth of Large Language Models, we start to wonder if we can leverage these Large Language Models in the domain of topic modeling.

There have been several models adopting Large Language Models for topic modeling, like the embedded topic model and the contextualized topic model. However, with the rapid developments in natural language processing, these models have a hard time keeping up.

A solution to this problem is BERTopic, a topic modeling technique that leverages a highly-flexible and modular architecture. Through this modularity, many newly released models can be integrated within its architecture. As the field of Large Language Modeling grows, so does BERTopic. This allows for some interesting and unexpected ways in which these models can be applied in topic modeling.

BERTopic

BERTopic is a topic modeling technique that assumes that clusters of semantically similar documents are a powerful way of generating and describing clusters. The documents in each cluster are expected to describe a major theme and combined they might represent a topic.

As we have seen with text clustering, a collection of documents in a cluster might represent a common theme but the theme itself is not yet described. With text clustering, we would have to go through every single document in a cluster to understand what the cluster is about. To get to the point where we can call a cluster a topic, we need a method for describing that cluster in a condensed and human-readable way.

Although there are quite a few methods for doing so, there is a

trick in BERTopic that allows it to quickly describe a cluster, and therefore make it a topic, whilst generating a highly modular pipeline. The underlying algorithm of BERTopic contains, roughly, two major steps.

First, as we did in our text clustering example, we embed our documents to create numerical representations, then reduce their dimensionality and finally cluster the reduced embeddings. The result is clusters of semantically similar documents.

[Figure 3-8](#) describes the same steps as before, namely using sentence-transformers for embedding the documents, UMAP for dimensionality reduction, and HDBSCAN for clustering.

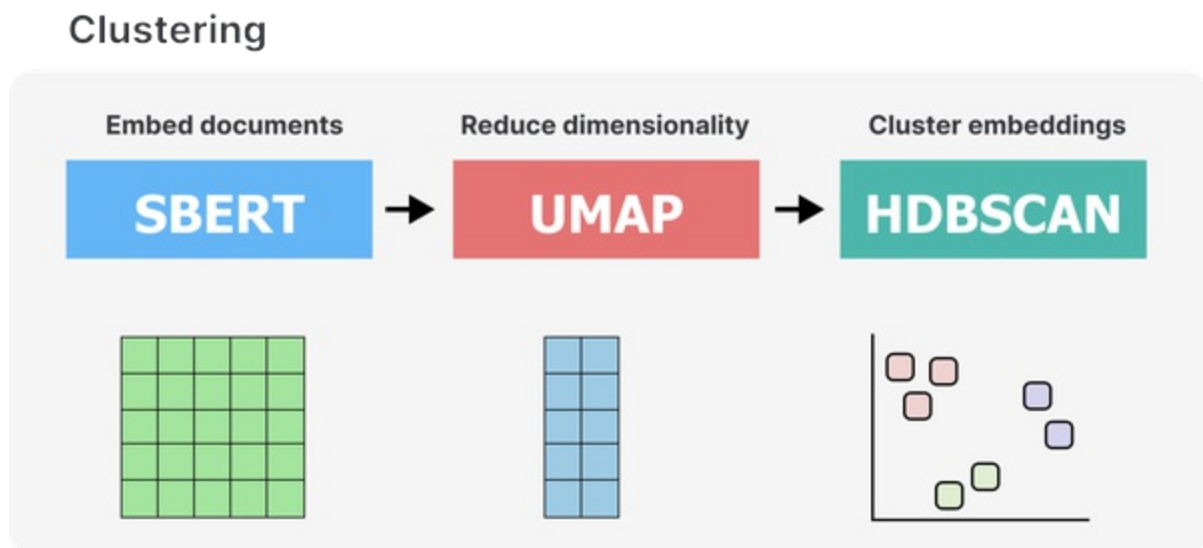


Figure 3-8. The first part of BERTopic's pipeline is clustering textual data.

Second, we find the best-matching keywords or phrases for each cluster. Most often, we would take the centroid of a cluster and find words, phrases, or even sentences that might represent it best. There is a disadvantage to this however: we would have to continuously keep track of our embeddings, and if we were to have millions of documents storing and keeping track becomes computationally difficult. Instead, BERTopic uses the classic bag-of-words method to represent the clusters. A bag of words is exactly what the name implies, for each document we simply count how often a certain word appears and use that as our textual representation.

However, words like “the”, “and”, and “I” appear quite frequently in most English texts and are likely to be overrepresented. To give proper weight to these words, BERTopic uses a technique called c-TF-IDF, which stands for class-based term-frequency inverse-document frequency. c-TF-IDF is a class-based adaptation of the classic TF-IDF procedure. Instead of considering the importance of words within documents, c-TF-IDF considers the importance of words between clusters of documents.

To use c-TF-IDF, we first concatenate each document in a cluster to generate one long document. Then, we extract the frequency of the term f_x in class c , where c refers to one of the

clusters we created before. Now we have, per cluster, how many and which words they contain, a mere count.

To weight this count, we take the logarithm of one plus the average number of words per cluster A divided by the frequency of term x across all clusters. Plus one is added within the logarithm to guarantee positive values which is also often done within TF-IDF.

As shown in [Figure 3-9](#), the c-TF-IDF calculation allows us to generate, for each word in a cluster, a weight corresponding to that cluster. As a result, we generate a topic-term matrix for each topic that describes the most important words they contain. It is essentially a ranking of a corpus' vocabulary in each topic.

Topic Representation

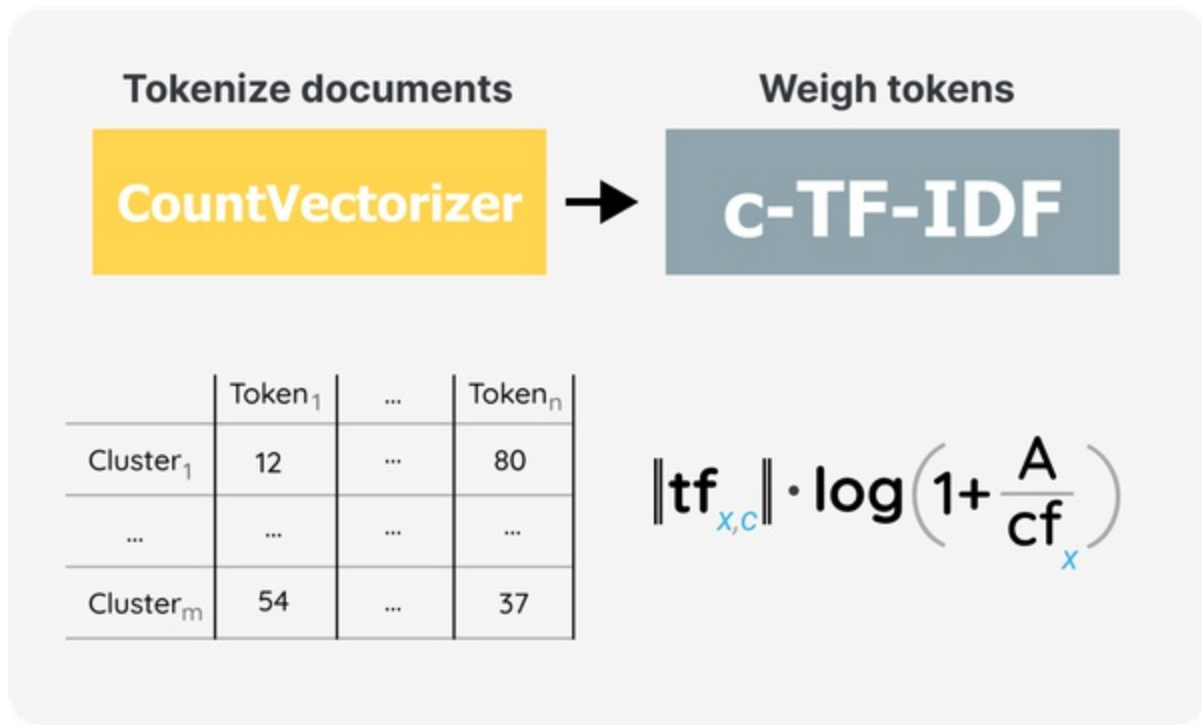


Figure 3-9. The second part of BERTopic's pipeline is representing the topics. The calculation of the weight of term x in a class c .

Putting the two steps together, clustering and representing topics, results in the full pipeline of BERTopic, as illustrated in [Figure 3-10](#). With this pipeline, we can cluster semantically similar documents and from the clusters generate topics represented by several keywords. The higher the weight of a keyword for a topic, the more representative it is of that topic.



Figure 3-10. The full pipeline of BERTopic, roughly, consists of two steps, clustering and topic representation.

NOTE

Interestingly, the c-TF-IDF trick does not use a Large Language Model and therefore does not take the context and semantic nature of words into account. However, like with neural search, it allows for an efficient starting point after which we can use the more compute-heavy techniques, such as GPT-like models.

One major advantage of this pipeline is that the two steps, clustering and topic representation, are relatively independent of one another. When we generate our topics using c-TF-IDF, we do not use the models from the clustering step, and, for example, do not need to track the embeddings of every single document. As a result, this allows for significant modularity not only with respect to the topic generation process but the entire pipeline.

NOTE

With clustering, each document is assigned to only a single cluster or topic. In practice, documents might contain multiple topics, and assigning a multi-topic document to a single topic would not always be the most accurate method. We will go into this later, as BERTopic has a few ways of handling this, but it is important to understand that at its core, topic modeling with BERTopic is a clustering task.

The modular nature of BERTopic's pipeline is extensible to every component. Although sentence-transformers are used as a default embedding model for transforming documents to numerical representations, nothing is stopping us from using any

other embedding technique. The same applies to the dimensionality reduction, clustering, and topic generation process. Whether a use case calls for k-Means instead of HDBSCAN, and PCA instead of UMAP, anything is possible.

You can think of this modularity as building with lego blocks, each part of the pipeline is completely replaceable with another, similar algorithm. This “lego block” way of thinking is illustrated in [Figure 3-11](#). The figure also shows an additional algorithmic lego block that we can use. Although we use c-TF-IDF to create our initial topic representations, there are a number of interesting ways we can use LLMs to fine-tune these representations. In the “**Representation Models**” section below, we will go into extensive detail on how this algorithmic lego block works.

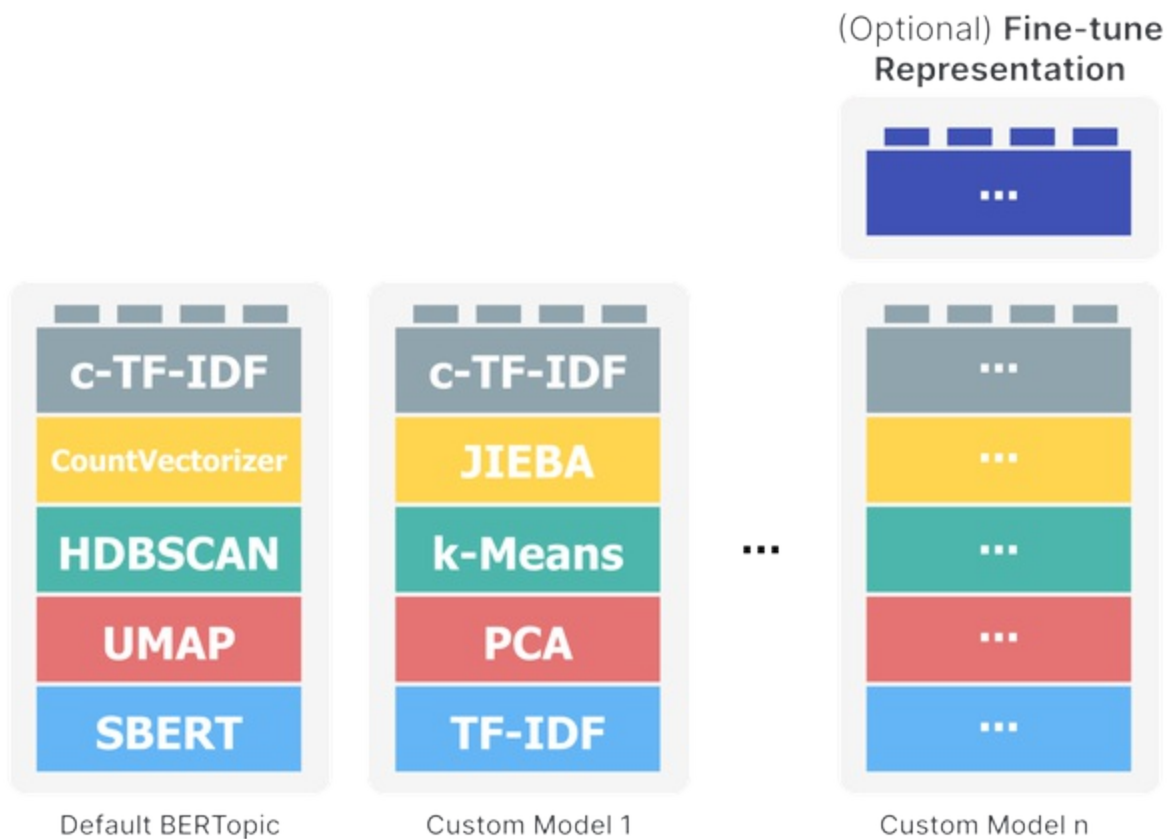


Figure 3-11. The modularity of BERTopic is a key component and allows you to build your own topic model whoever you want.

Code Overview

Enough talk! This is a hands-on book, so it is finally time for some hands-on coding. The default pipeline, as illustrated previously in [Figure 3-10](#), only requires a few lines of code:

```
from bertopic import BERTopic

# Instantiate our topic model
topic_model = BERTopic()
```

```
# Fit our topic model on a list of documents
topic_model.fit(documents)
```

However, the modularity that BERTopic is known for and that we have visualized thus far can also be visualized through a coding example. First, let us import some relevant packages:

```
from umap import UMAP
from hdbscan import HDBSCAN
from sentence_transformers import SentenceTransformer
from sklearn.feature_extraction.text import CountVector

from bertopic import BERTopic
from bertopic.representation import KeyBERTInspired
from bertopic.vectorizers import ClassTfidfTransformer
```

As you might have noticed, most of the imports, like UMAP and HDBSCAN, are part of the default BERTopic pipeline. Next, let us build the default pipeline of BERTopic a bit more explicitly and go through each individual step:

```
# Step 1 - Extract embeddings (blue block)
embedding_model = SentenceTransformer("all-MiniLM-L6-v2")

# Step 2 - Reduce dimensionality (red block)
```

```

umap_model = UMAP(n_neighbors=15, n_components=5, min_

# Step 3 - Cluster reduced embeddings (green block)
hdbscan_model = HDBSCAN(min_cluster_size=15, metric='e

# Step 4 - Tokenize topics (yellow block)
vectorizer_model = CountVectorizer(stop_words="english

# Step 5 - Create topic representation (grey block)
ctfidf_model = ClassTfidfTransformer()

# Step 6 - (Optional) Fine-tune topic representations
# a `bertopic.representation` model (purple block)
representation_model = KeyBERTInspired()
# Combine the steps and build our own topic model
topic_model = BERTopic(
    embedding_model=embedding_model,          # Step 1
    umap_model=umap_model,                   # Step 2
    hdbscan_model=hdbscan_model,             # Step 3
    vectorizer_model=vectorizer_model,       # Step 4
    ctfidf_model=ctfidf_model,               # Step 5
    representation_model=representation_model # Step 6
)

```

This code allows us to go through all steps of the algorithm explicitly and essentially let us build the topic model however we want. The resulting topic model, as defined in the variable

`topic_model`, now represents the base pipeline of BERTopic as illustrated back in [Figure 3-10](#).

Example

We are going to keep using the abstracts of ArXiv articles throughout this use case. To recap what we did with text clustering, we start by importing our dataset using HuggingFace's dataset package and extracting metadata that we are going to use later on, like the abstracts, years, and categories of the articles.

```
# Load data from huggingface
from datasets import load_dataset
dataset = load_dataset("maartengr/arxiv_nlp")

# Extract specific metadata
abstracts = dataset["Abstracts"]
years = dataset["Years"]
categories = dataset["Categories"]
titles = dataset["Titles"]
```

Using BERTopic is quite straightforward, and it can be used in just three lines:

```
# Train our topic model in only three lines of code
from bertopic import BERTopic
```

```
topic_model = BERTopic()  
topics, probs = topic_model.fit_transform(abstracts)
```

With this pipeline, you will have 3 variables returned, namely `topic_model`, `topics`, and `probs`:

- `topic_model` is the model that we have just trained before and contains information about the model and the topics that we created.
- `topics` are the topics for each abstract.
- `probs` are the probabilities that a topic belongs to a certain abstract.

Before we start to explore our topic model, there is one change that we will need to make the results reproducible. As mentioned before, one of the underlying models of BERTopic is UMAP. This model is stochastic in nature which means that every time we run BERTopic, we will get different results. We can prevent this by passing a ``random_state`` to the UMAP model.

```
from umap import UMAP  
from bertopic import BERTopic  
  
# Using a custom UMAP model
```

```

umap_model = UMAP(n_neighbors=15, n_components=5, min_

# Train our model
topic_model = BERTopic(umap_model=umap_model)
topics, probs = topic_model.fit_transform(abstracts)

```

Now, let's start by exploring the topics that were created. The `get_topic_info()` method is useful to get a quick description of the topics that we found:

```

>>> topic_model.get_topic_info()

```

Topic	Count	Name	
0	-1	11648	-1_of_the_and_to
1	0	1554	0_question_answer_questions_qa
2	1	620	1_hate_offensive_toxic_detection
3	2	578	2_summarization_summaries_summary_abs
4	3	568	3_parsing_parser_dependency_amr
...	...	...	...
317	316	10	316_prf_search_conversational_spok
318	317	10	317_crowdsourcing_workers_annotato
319	318	10	318_curriculum_nmt_translation_dcl
320	319	10	319_botsim_menu_user_dialogue
321	320	10	320_color_colors_ib_naming

There are many topics generated from our model, **XXX**! Each of

these topics is represented by several keywords, which are concatenated with a “_” in the Name column. This Name column allows us to quickly get a feeling of what the topic is about as it shows the four keywords that best represent it.

NOTE

You might also have noticed that the very first topic is labeled -1. That topic contains all documents that could not be fitted within a topic and are considered to be outliers. This is a result of the clustering algorithm, HDBSCAN, that does not force all points to be clustered. To remove outliers, we could either use a non-outlier algorithm like k-Means or use BERTopic’s `reduce_outliers()` function to remove some of the outliers and assign them to topics.

For example, topic 2 contains the keywords “summarization”, “summaries”, “summary”, and “abstractive”. Based on these keywords, it seems that the topic is summarization tasks. To get the top 10 keywords per topic as well as their c-TF-IDF weights, we can use the `get_topic()` function:

```
>>> topic_model.get_topic(2)
[('summarization', 0.029974019692323675),
 ('summaries', 0.018938088406361412),
 ('summary', 0.018019112468622436),
 ('abstractive', 0.015758156442697138),
 ('document', 0.011038627359130419),
 ('extractive', 0.010607624721836042),
```

```
('rouge', 0.00936377058925341),  
( 'factual', 0.005651676100789188),  
( 'sentences', 0.005262910357048789),  
( 'mds', 0.005050565343932314)]
```

This gives us a bit more context about the topic and helps us understand what the topic is about. For example, it is interesting to see the word “rogue” appear since that is a common metric for evaluating summarization models.

We can use the `find_topics()` function to search for specific topics based on a search term. Let’s search for a topic about topic modeling:

```
>>> topic_model.find_topics("topic modeling")  
([17, 128, 116, 6, 235],  
 [0.6753638370140129,  
  0.40951682679389345,  
  0.3985390076544335,  
  0.37922002441932795,  
  0.3769700288091359])
```

It returns that topic 17 has a relatively high similarity (0.675) with our search term. If we then inspect the topic, we can see that it is indeed a topic about topic modeling:

```
>>> topic_model.get_topic(17)
[('topic', 0.0503756681079549),
 ('topics', 0.02834246786579726),
 ('lda', 0.015441277604137684),
 ('latent', 0.011458141214781893),
 ('documents', 0.01013764950401255),
 ('document', 0.009854201885298964),
 ('dirichlet', 0.009521114618288628),
 ('modeling', 0.008775384549157435),
 ('allocation', 0.0077508974418589605),
 ('clustering', 0.005909325849593925)]
```

Although we know that this topic is about topic modeling, let us see if the BERTopic abstract is also assigned to this topic:

```
>>> topics[titles.index('BERTopic: Neural topic modeli
17
```

It is! It seems that the topic is not just about LDA-based methods but also cluster-based techniques, like BERTopic.

Lastly, we mentioned before that many topic modeling techniques assume that there can be multiple topics within a single document or even a sentence. Although BERTopic leverages clustering, which assumes a single assignment to each

data point, it can approximate the topic distribution.

We can use this technique to see what the topic distribution is of the first sentence in the BERTopic paper:

```
index = titles.index('BERTopic: Neural topic modeling')

# Calculate the topic distributions on a token-level
topic_distr, topic_token_distr = topic_model.approximate_distribution(
    df = topic_model.visualize_approximate_distribution(
        df


```

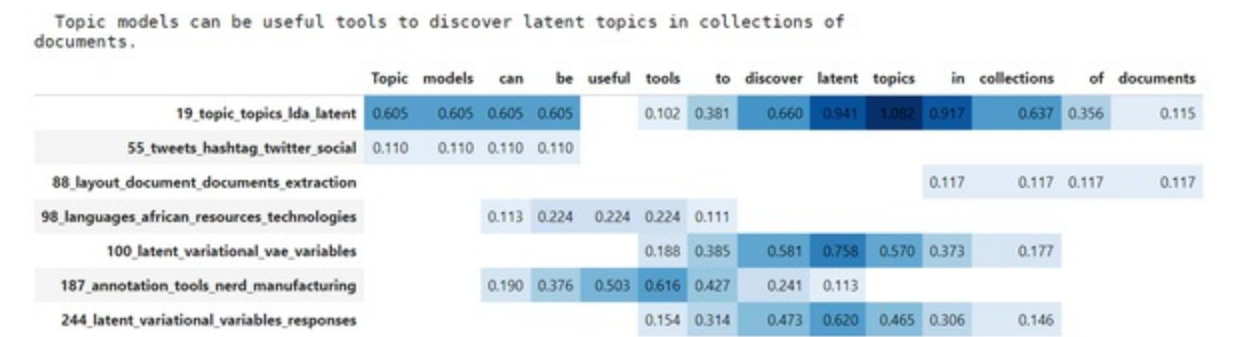


Figure 3-12. A wide range of visualization options are available in BERTopic.

The output, as shown in [Figure 3-12](#), demonstrates that the document, to a certain extent, contains multiple topics. This assignment is even done on a token level!

(Interactive) Visualizations

Going through **XXX** topics manually can be quite a task. Instead, several helpful visualization functions allow us to get a broad overview of the topics that were generated. Many of which are interactive by using the Plotly visualization framework.

[Figure 3-13](#) shows all possible visualization options in BERTopic, from 2D document representations and topic bar charts to topic hierarchy and similarity. Although we are not going through all visualizations, there are some worth looking into.



Figure 3-13. A wide range of visualization options are available in BERTopic.

To start, we can create a 2D representation of our topics by using UMAP to reduce the c-TF-IDF representations of each topic.

```
topic_model.visualize_topics()
```

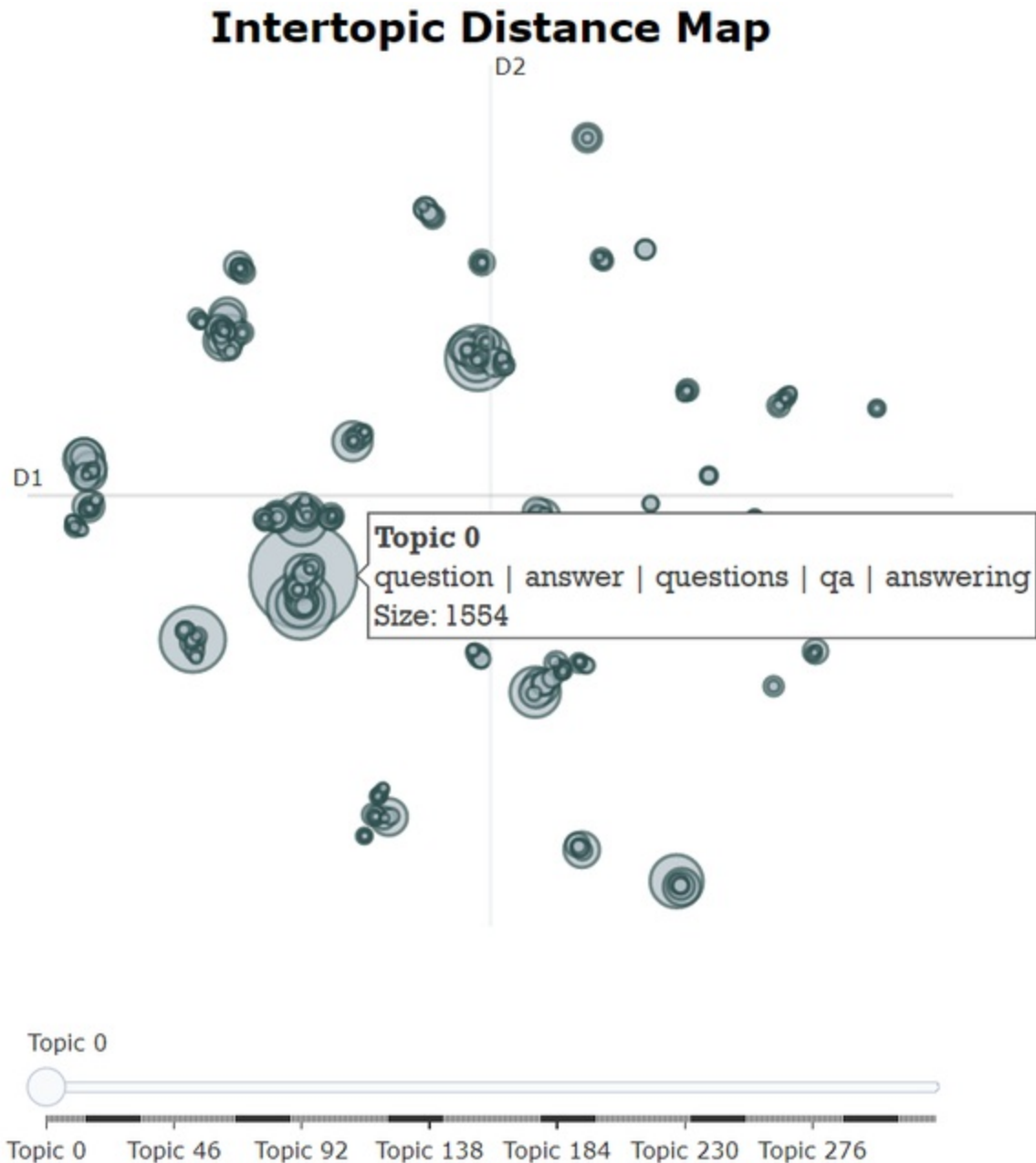


Figure 3-14. The intertopic distance map of topics represented in 2D space.

As shown in [Figure 3-14](#), this generates an interactive visualization that, when hovering over a circle, allows us to see the topic, its keywords, and its size. The larger the circle of a topic is, the more documents it contains. We can quickly see

groups of similar topics through interaction with this visualization.

We can use the `visualize_documents()` function to take this analysis to the next level, namely analyzing topics on a document level.

```
# Visualize a selection of topics and documents
topic_model.visualize_documents(titles,
                               topics=[0, 1, 2, 3, 4, 6, 7, 10, 12,
                                        13, 16, 33, 40, 45, 46, 65])
```



Figure 3-15. Abstracts and their topics are represented in a 2D visualization.

[Figure 3-15](#) demonstrates how BERTopic can visualize documents in a 2D-space.

NOTE

We only visualized a selection of topics since showing all 300 topics would result in quite a messy visualization. Also, instead of passing `abstracts`, we passed `titles` since we only want to view the titles of each paper when we hover over a document and not the entire abstract.

Lastly, we can create a bar chart of the keywords in a selection of topics using `visualize_barchart()`:

```
topic_model.visualize_barchart(topics=list(range(50, 5
```



Figure 3-16. The top 5 keywords for the first 8 topics.

The bar chart in [Figure 3-16](#) gives a nice indication of which keywords are most important to a specific topic. Take topic 2 for example—it seems that the word “summarization” is most representative of that topic and that other words are very similar in importance.

Representation Models

With the neural-search style modularity that BERTopic employs, it can leverage many different types of Large Language Models whilst minimizing computing. This allows for a large range of topic fine-tuning methods, from part-of-speech to text-generation methods, like ChatGPT. [Figure 3-17](#) demonstrates the variety of LLMs that we can leverage to fine-tune topic representations.

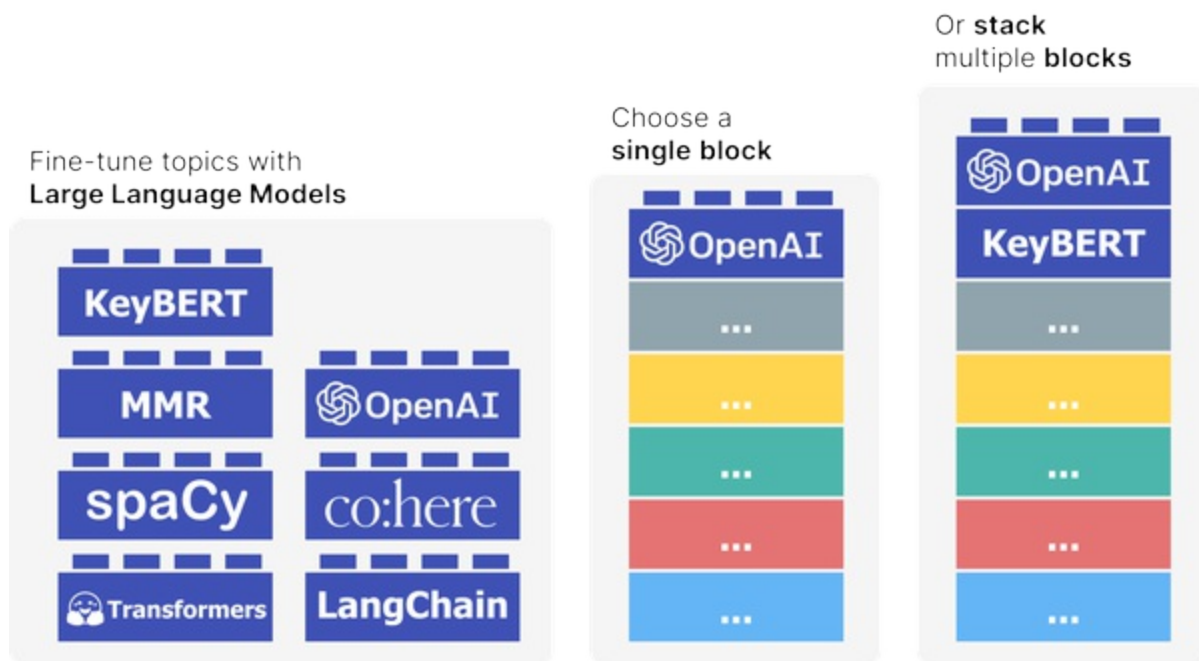


Figure 3-17. After applying the c-TF-IDF weighting, topics can be fine-tuned with a wide variety of representation models. Many of which are Large Language Models.

Topics generated with c-TF-IDF serve as a good first ranking of words with respect to their topic. In this section, these initial rankings of words can be considered candidate keywords for a topic as we might change their rankings based on any

representation model. We will go through several representation models that can be used within BERTopic and that are also interesting from a Large Language Modeling standpoint.

Before we start, we first need to do two things. First, we are going to save our original topic representations so that it will be much easier to compare with and without representation models:

```
# Save original representations
from copy import deepcopy
original_topics = deepcopy(topic_model.topic_representations)
```

Second, let's create a short wrapper that we can use to quickly visualize the differences in topic words to compare with and without representation models:

```
def topic_differences(model, original_topics, max_length=10):
    """ For the first 10 topics, show the differences in
    topic representations between two models """
    for topic in range(nr_topics):

        # Extract top 5 words per topic per model
        og_words = " | ".join(list(zip(*original_topics[topic].top_n_words(5))))
        new_words = " | ".join(list(zip(*model.get_topic(topic).top_n_words(5))))
```

```
# Print a 'before' and 'after'
whitespaces = " " * (max_length - len(og_words))
print(f"Topic: {topic}      {og_words}{whitespaces}-
```

KeyBERTInspired

c-TF-IDF generated topics do not consider the semantic nature of words in a topic which could end up creating topics with stopwords. We can use the module

`bertopic.representation_model.KeyBERTInspired()` to fine-tune the topic keywords based on their semantic similarity to the topic.

KeyBERTInspired is, as you might have guessed, a method inspired by the [keyword extraction package, KeyBERT](#). In its most basic form, KeyBERT compares the embeddings of words in a document with the document embedding using cosine similarity to see which words are most related to the document. These most similar words are considered keywords.

In BERTopic, we want to use something similar but on a topic level and not a document level. As shown in [Figure 3-18](#), KeyBERTInspired uses c-TF-IDF to create a set of representative documents per topic by randomly sampling 500 documents per topic, calculating their c-TF-IDF values, and finding the most

representative documents. These documents are embedded and averaged to be used as an updated topic embedding. Then, the similarity between our candidate keywords and the updated topic embedding is calculated to re-rank our candidate keywords.

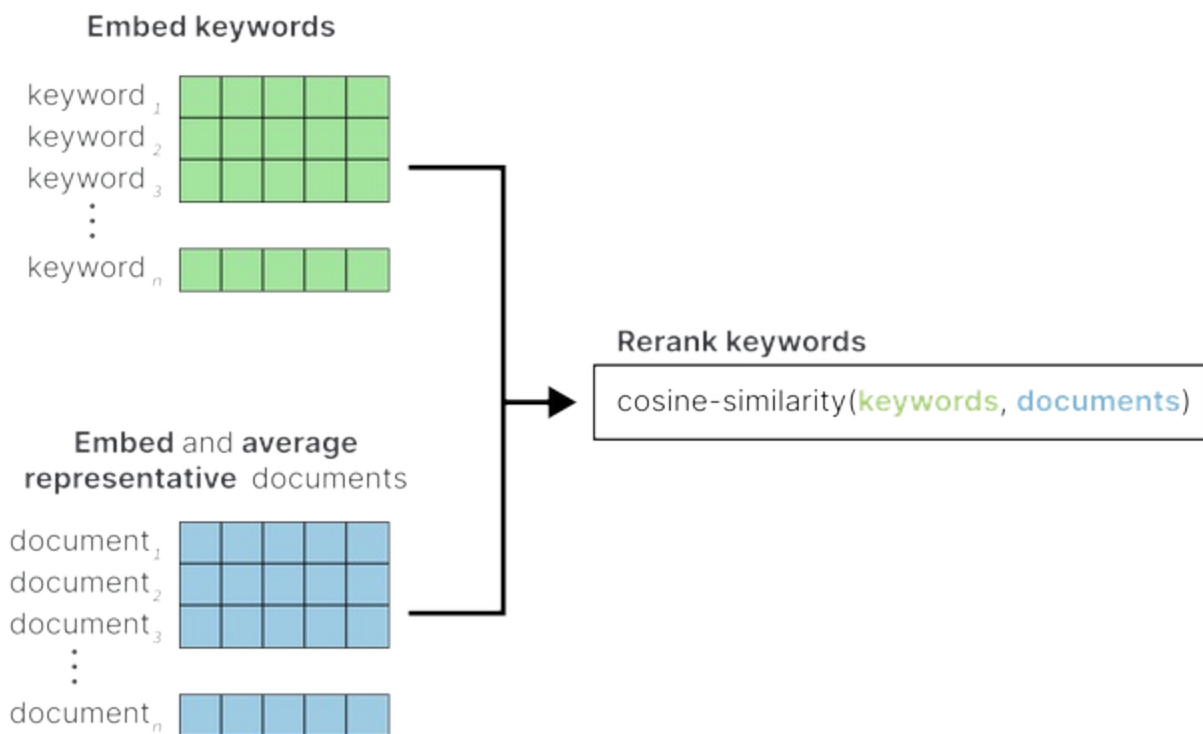


Figure 3-18. The procedure of the KeyBERTInspired representation model

```
# KeyBERTInspired
from bertopic.representation import KeyBERTInspired
representation_model = KeyBERTInspired()

# Update our topic representations
new_topic_model.update_topics(abstracts, representation_model)
```

```
# Show topic differences
topic_differences(topic_model, new_topic_model)
```

Topic: 0 question | qa | questions | answer | answering
--> questionanswering | answering | questionanswer |
attention | retrieval

Topic: 1 hate | offensive | speech | detection | toxic -
-> hateful | hate | cyberbullying | speech | twitter

Topic: 2 summarization | summaries | summary |
abstractive | extractive --> summarizers | summarizer |
summarization | summarisation | summaries

Topic: 3 parsing | parser | dependency | amr | parsers -
-> parsers | parsing | treebanks | parser | treebank

Topic: 4 word | embeddings | embedding | similarity |
vectors --> word2vec | embeddings | embedding |
similarity | semantic

Topic: 5 gender | bias | biases | debiasing | fairness -
-> bias | biases | genders | gender | gendered

Topic: 6 relation | extraction | re | relations | entity
--> relations | relation | entities | entity |
relational

Topic: 7 prompt | fewshot | prompts | incontext | tuning
--> prompttuning | prompts | prompt | prompting |
promptbased

Topic: 8 aspect | sentiment | absa | aspectbased |
opinion --> sentiment | aspect | aspects | aspectlevel |
sentiments

Topic: 9 explanations | explanation | rationales |
rationale | interpretability --> explanations |
explainers | explainability | explaining | attention

The updated model shows that the topics are much easier to read compared to the original model. It also shows the downside of using embedding-based techniques. Words in the original model, like “amr” and “qa” are perfectly reasonable words

Part-of-Speech

c-TF-IDF does not make any distinction of the type of words it deems to be important. Whether it is a noun, verb, adjective, or even a preposition, they can all end up as important keywords. When we want to have human-readable labels that are straightforward and intuitive to interpret, we might want topics that are described by, for example, nouns only.

This is where the well-known SpaCy package comes in. An

industrial-grade NLP framework that comes with a variety of pipelines, models, and deployment options. More specifically, we can use SpaCy to load in an English model that is capable of detecting part of speech, whether a word is a noun, verb, or something else.

As shown in [Figure 3-19](#), we can use SpaCy to make sure that only nouns end up in our topic representations. As with most representation models, this is highly efficient since the nouns are extracted from only a small but representative subset of the data.

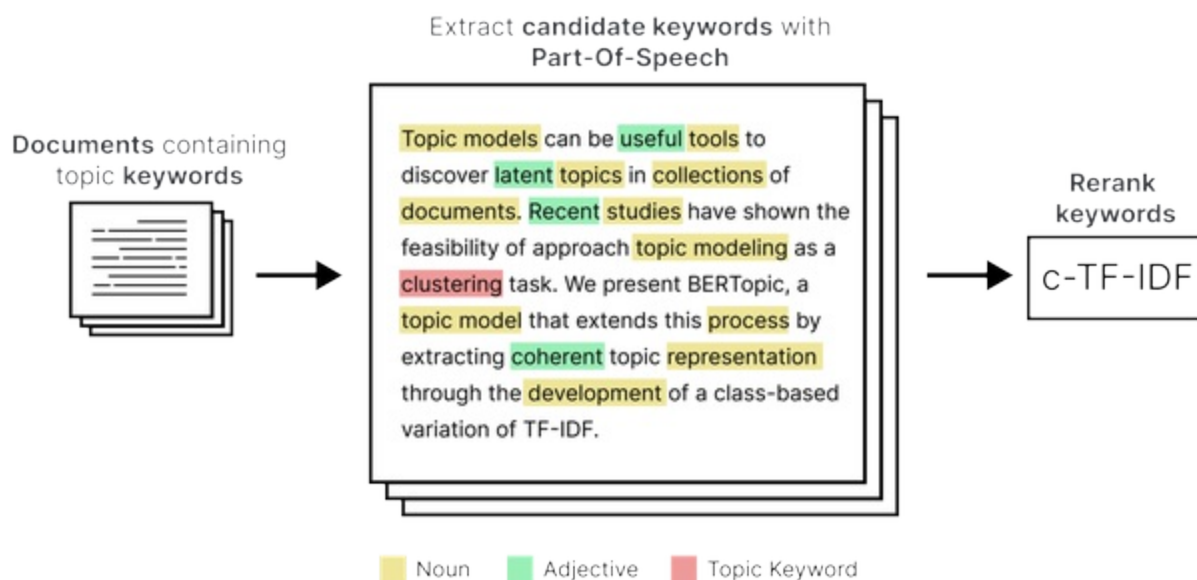


Figure 3-19. The procedure of the PartOfSpeech representation model

```
# Part-of-Speech tagging
from bertopic.representation import PartOfSpeech
representation_model = PartOfSpeech("en_core_web_sm")
```

```
# Use the representation model in BERTopic on top of t
topic_model.update_topics(abstracts, representation_mo

# Show topic differences
topic_differences(topic_model, original_topics)
```

Topic: 0 question | qa | questions | answer | answering
--> question | questions | answer | answering | answers

Topic: 1 hate | offensive | speech | detection | toxic -
-> hate | offensive | speech | detection | toxic

Topic: 2 summarization | summaries | summary |
abstractive | extractive --> summarization | summaries |
summary | abstractive | extractive

Topic: 3 parsing | parser | dependency | amr | parsers -
-> parsing | parser | dependency | parsers | treebank

Topic: 4 word | embeddings | embedding | similarity |
vectors --> word | embeddings | similarity | vectors | words

Topic: 5 gender | bias | biases | debiasing | fairness -
-> gender | bias | biases | debiasing | fairness

Topic: 6 relation | extraction | re | relations | entity
--> relation | extraction | relations | entity | distant

Topic: 7 prompt | fewshot | prompts | incontext | tuning
--> prompt | prompts | tuning | prompting | tasks

Topic: 8 aspect | sentiment | absa | aspectbased |
opinion --> aspect | sentiment | opinion | aspects |
polarity

Topic: 9 explanations | explanation | rationales |
rationale | interpretability --> explanations |
explanation | rationales | rationale | interpretability

Maximal Marginal Relevance

With c-TF-IDF, there can be a lot of redundancy in the resulting keywords as it does not consider words like “car” and “cars” to be essentially the same thing. In other words, we want sufficient diversity in the resulting topics with as little repetition as possible. ([Figure 3-20](#))

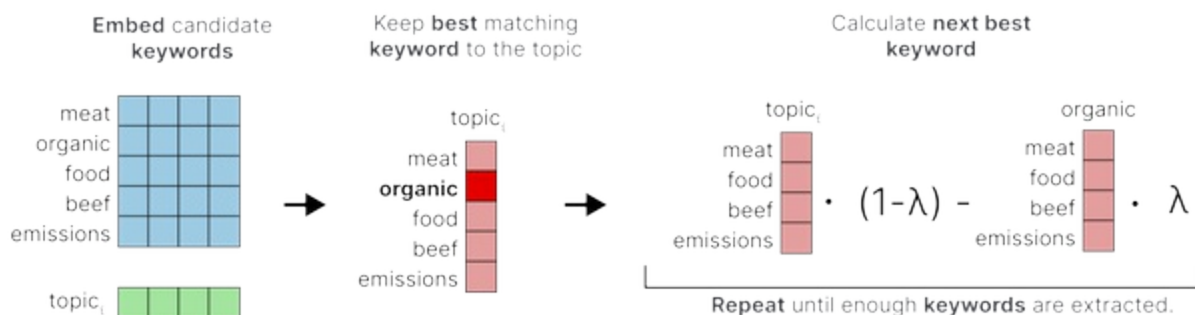


Figure 3-20. The procedure of the Maximal Marginal Relevance representation model. The diversity of the resulting keywords is represented by lambda (λ).

We can use an algorithm, called Maximal Marginal Relevance (MMR) to diversify our topic representations. The algorithm starts with the best matching keyword to a topic and then iteratively calculates the next best keyword whilst taking a certain degree of diversity into account. In other words, it takes a number of candidate topic keywords, for example, 30, and tries to pick the top 10 keywords that are best representative of the topic but are also diverse from one another.

```
# Maximal Marginal Relevance
from bertopic.representation import MaximalMarginalRel
representation_model = MaximalMarginalRelevance(diversity=0.7)

# Use the representation model in BERTopic on top of the
topic_model.update_topics(abstracts, representation_model=representation_model)

# Show topic differences
topic_differences(topic_model, original_topics)
```

Topic: 0 question | qa | questions | answer | answering
--> qa | questions | answering | comprehension |
retrieval

Topic: 1 hate | offensive | speech | detection | toxic -
-> speech | abusive | toxicity | platforms | hateful

Topic: 2 summarization | summaries | summary |
abstractive | extractive --> summarization | extractive
| multidocument | documents | evaluation

Topic: 3 parsing | parser | dependency | amr | parsers -
-> amr | parsers | treebank | syntactic | constituent

Topic: 4 word | embeddings | embedding | similarity |
vectors --> embeddings | similarity | vector | word2vec
| glove

Topic: 5 gender | bias | biases | debiasing | fairness -
-> gender | bias | fairness | stereotypes | embeddings

Topic: 6 relation | extraction | re | relations | entity
--> extraction | relations | entity | documentlevel |
docre

Topic: 7 prompt | fewshot | prompts | incontext | tuning
--> prompts | zeroshot | plms | metalearning | label

Topic: 8 aspect | sentiment | absa | aspectbased |
opinion --> sentiment | absa | aspects | extraction |
polarities

Topic: 9 explanations | explanation | rationales |
rationale | interpretability --> explanations |
interpretability | saliency | faithfulness | methods

The resulting topics are much more diverse! Topic **XXX**, which

originally used a lot of “summarization” words, the topic only contains the word “summarization”. Also, duplicates, like “embedding” and “embeddings” are now removed.

Text Generation

Text generation models have shown great potential in 2023. They perform well across a wide range of tasks and allow for extensive creativity in prompting. Their capabilities are not to be underestimated and not using them in BERTopic would frankly be a waste. We talked at length about these models in Chapter **XXX**, but it’s useful now to see how they tie into the topic modeling process.

As illustrated in [Figure 3-21](#), we can use them in BERTopic efficiently by focusing on generating output on a topic level and not a document level. This can reduce the number of API calls from millions (e.g., millions of abstracts) to a couple of hundred (e.g., hundreds of topics). Not only does this significantly speed up the generation of topic labels, but you also do not need a massive amount of credits when using an external API, such as Cohere or OpenAI.

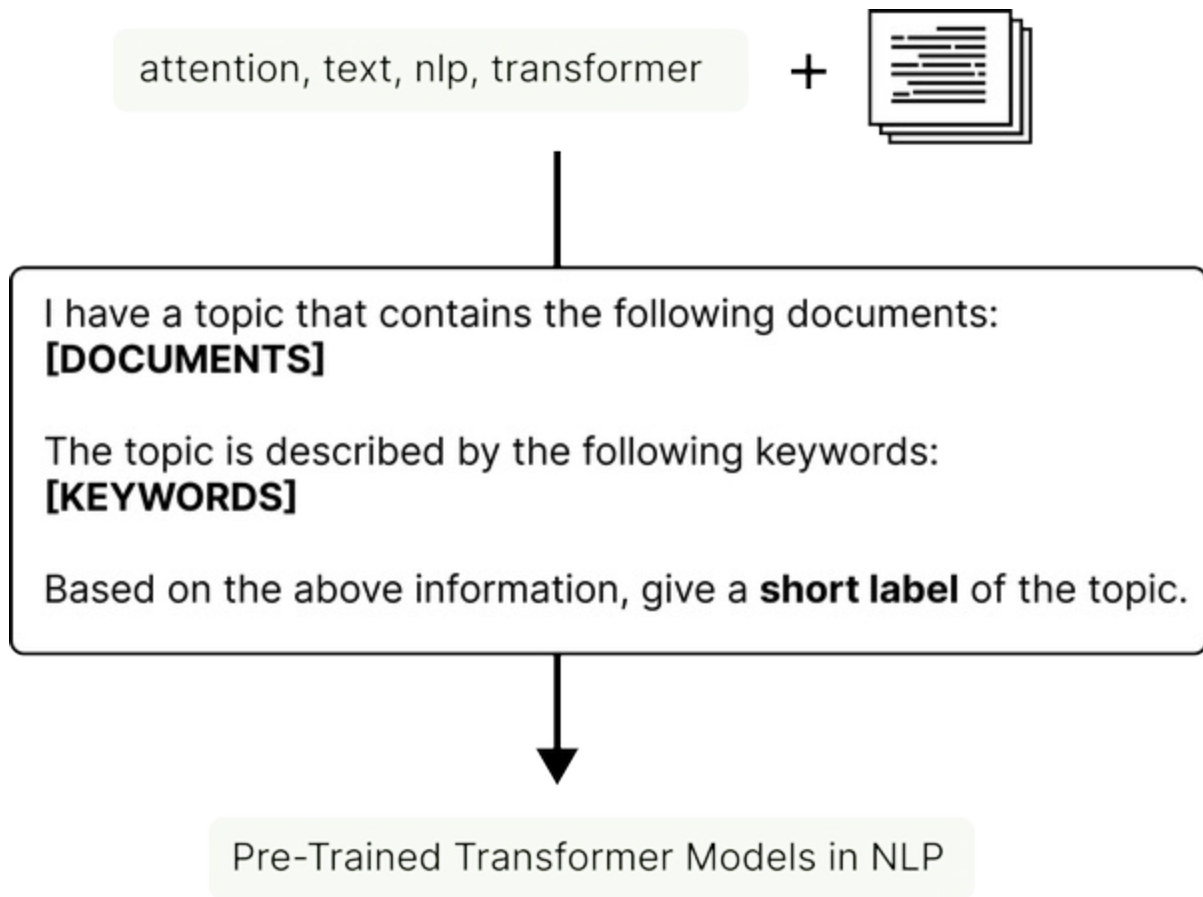


Figure 3-21. Use text generative LLMs and prompt engineering to create labels for topics from keywords and documents related to each topic.

Prompting

As was illustrated back in [Figure 3-21](#), one major component of text generation is prompting. In BERTopic this is just as important since we want to give enough information to the model such that it can decide what the topic is about. Prompts in BERTopic generally look something like this:

```
prompt = ""  
I have a topic that contains the following documents:
```

The topic is described by the following keywords: [KEYWORDS]

Based on the above information, give a short label of the topic.
""

There are three components to this prompt. First, it mentions a few documents of a topic that best describes it. These documents are selected by calculating their c-TF-IDF representations and comparing them with the topic c-TF-IDF representation. The top 4 most similar documents are then extracted and referenced using the “[**DOCUMENTS**]” tag.

I have a topic that contains the following documents:

Second, the keywords that make up a topic are also passed to the prompt and referenced using the “[**KEYWORDS**]” tag. These keywords could also already be optimized using KeyBERTInspired, PartOfSpeech, or any representation model.

The topic is described by the following keywords: [KEYWORDS]

Third, we give specific instructions to the Large Language Model. This is just as important as the steps before since this will decide

how the model generates the label.

```
Based on the above information, give a short label of
```

The prompt will be rendered as follows for topic XXX:

```
"""
```

```
I have a topic that contains the following documents:
```

- Our videos are also made possible by your support on
- If you want to help us make more videos, you can do
- If you want to help us make more videos, you can do
- And if you want to support us in our endeavor to sur

```
The topic is described by the following keywords: vide
```

```
Based on the above information, give a short label of
```

```
"""
```

HuggingFace

Fortunately, as with most Large Language Models, there is an enormous amount of open-source models that we can use through [HuggingFace's Modelhub](#).

One of the most well-known open-source Large Language Models

that is optimized for text generation, is one from the Flan-T5 family of generation models. What is interesting about these models is that they have been trained using a method called **instruction tuning**. By fine-tuning T5 models on many tasks phrased as instructions, the model learns to follow specific instructions and tasks.

BERTopic allows for using such a model to generate topic labels. We create a prompt and ask it to create topics based on the keywords of each topic, labeled with the `[KEYWORDS]` tag.

```
from transformers import pipeline
from bertopic.representation import TextGeneration

# Text2Text Generation with Flan-T5
generator = pipeline('text2text-generation', model='go
representation_model = TextGeneration(generator)

# Use the representation model in BERTopic on top of t
topic_model.update_topics(abstracts, representation_mo

# Show topic differences
topic_differences(topic_model, original_topics)
```

```
Topic: 0 speech | asr | recognition | acoustic |
endtoend --> audio grammatical recognition
```

Topic: 1 clinical | medical | biomedical | notes | health --> ehr

Topic: 2 summarization | summaries | summary | abstractive | extractive --> mds

Topic: 3 parsing | parser | dependency | amr | parsers -> parser

Topic: 4 hate | offensive | speech | detection | toxic -> Twitter

Topic: 5 word | embeddings | embedding | vectors | similarity --> word2vec

Topic: 6 gender | bias | biases | debiasing | fairness -> gender bias

Topic: 7 ner | named | entity | recognition | nested --> ner

Topic: 8 prompt | fewshot | prompts | incontext | tuning --> gpt3

Topic: 9 relation | extraction | re | relations | distant --> docre

There are interesting topic labels that are created but we can also see that the model is not perfect by any means.

OpenAI

When we are talking about generative AI, we cannot forget about ChatGPT and its incredible performance. Although not open source, it makes for an interesting model that has changed the AI field in just a few months. We can select any text generation model from OpenAI's collection to use in BERTopic.

As this model is trained on RLHF and optimized for chat purposes, prompting is quite satisfying with this model.

```
from bertopic.representation import OpenAI

# OpenAI Representation Model
prompt = """
I have a topic that contains the following documents:
The topic is described by the following keywords: [KEYWORDS]

Based on the information above, extract a short topic
topic: <topic label>
"""

representation_model = OpenAI(model="gpt-3.5-turbo", d

# Use the representation model in BERTopic on top of t
topic_model.update_topics(abstracts, representation_mo

# Show topic differences
```

```
topic_differences(topic_model, original_topics)
```

Topic: 0 speech | asr | recognition | acoustic |
endtoend --> audio grammatical recognition

Topic: 1 clinical | medical | biomedical | notes |
health --> ehr

Topic: 2 summarization | summaries | summary |
abstractive | extractive --> mds

Topic: 3 parsing | parser | dependency | amr | parsers -
-> parser

Topic: 4 hate | offensive | speech | detection | toxic -
-> Twitter

Topic: 5 word | embeddings | embedding | vectors |
similarity --> word2vec

Topic: 6 gender | bias | biases | debiasing | fairness -
-> gender bias

Topic: 7 ner | named | entity | recognition | nested -->
ner

Topic: 8 prompt | fewshot | prompts | incontext | tuning
--> gpt3

```
Topic: 9 relation | extraction | re | relations |  
distant --> docre
```

Since we expect ChatGPT to return the topic in a specific format, namely “topic: <topic label>” it is important to instruct the model to return it as such when we create a custom prompt. Note that we also add the `delay\_in\_seconds` parameter to create a constant delay between API calls in case you have a free account.

Cohere

As with OpenAI, we can use Cohere’s API within BERTopic on top of its pipeline to further fine-tune the topic representations with a generative text model. Make sure to grab an API key and you can start generating topic representations.

```
import cohere  
from bertopic.representation import Cohere  
  
# Cohere Representation Model  
co = cohere.Client(my_api_key)  
representation_model = Cohere(co)  
  
# Use the representation model in BERTopic on top of t  
topic_model.update_topics(abstracts, representation_mo  
  
# Show topic differences
```

```
topic_differences(topic_model, original_topics)
```

Topic: 0 speech | asr | recognition | acoustic |
endtoend --> audio grammatical recognition

Topic: 1 clinical | medical | biomedical | notes |
health --> ehr

Topic: 2 summarization | summaries | summary |
abstractive | extractive --> mds

Topic: 3 parsing | parser | dependency | amr | parsers -
-> parser

Topic: 4 hate | offensive | speech | detection | toxic -
-> Twitter

Topic: 5 word | embeddings | embedding | vectors |
similarity --> word2vec

Topic: 6 gender | bias | biases | debiasing | fairness -
-> gender bias

Topic: 7 ner | named | entity | recognition | nested -->
ner

Topic: 8 prompt | fewshot | prompts | incontext | tuning
--> gpt3

Topic: 9 relation | extraction | re | relations |
distant --> docre

LangChain

To take things a step further with Large Language Models, we can leverage the LangChain framework. It allows for any of the previous text generation methods to be supplemented with additional information or even chained together. Most notably, LangChain connects language models to other sources of data to enable them to interact with their environment.

For example, we could use it to build a vector database with OpenAI and apply ChatGPT on top of that database. As we want to minimize the amount of information LangChain needs, the most representative documents are passed to the package. Then, we could use any LangChain-supported language model to extract the topics. The example below demonstrates the use of OpenAI with LangChain.

```
from langchain.llms import OpenAI
from langchain.chains.question_answering import load_q
from bertopic.representation import LangChain

# Langchain representation model
chain = load_qa_chain(OpenAI(temperature=0, openai_api,
```

```
representation_model = LangChain(chain)

# Use the representation model in BERTopic on top of t
topic_model.update_topics(abstracts, representation_mo

# Show topic differences
topic_differences(topic_model, original_topics)
```

Topic: 0 speech | asr | recognition | acoustic |
endtoend --> audio grammatical recognition

Topic: 1 clinical | medical | biomedical | notes |
health --> ehr

Topic: 2 summarization | summaries | summary |
abstractive | extractive --> mds

Topic: 3 parsing | parser | dependency | amr | parsers -
-> parser

Topic: 4 hate | offensive | speech | detection | toxic -
-> Twitter

Topic: 5 word | embeddings | embedding | vectors |
similarity --> word2vec

Topic: 6 gender | bias | biases | debiasing | fairness -
-> gender bias

```
Topic: 7 ner | named | entity | recognition | nested -->  
ner
```

```
Topic: 8 prompt | fewshot | prompts | incontext | tuning  
--> gpt3
```

```
Topic: 9 relation | extraction | re | relations |  
distant --> docre
```

Topic Modeling Variations

The field of topic modeling is quite broad and ranges from many different applications to variations of the same model. This also holds for BERTopic as it has implemented a wide range of variations for different purposes, such as dynamic, (semi-) supervised, online, hierarchical, and guided topic modeling.

[Figure 3-22-X](#) shows a number of topic modeling variations and how to implement them in BERTopic.

Guided Topic Modeling	<code>BERTopic(seed_topic_list=seed_topic_list)</code>
(semi)-Supervised Topic Modeling	<code>topic_model.fit(abstracts, y=classes)</code>
Incremental Topic Modeling	<code>topic_model.partial_fit(abstracts)</code>
Hierarchical Topic Modeling	<code>topic_model.hierarchical_topics(abstracts)</code>
Dynamic Topic Modeling	<code>topic_model.topics_over_time(abstracts, years)</code>
Class-based Topic Modeling	<code>topic_model.topics_per_class(abstracts, classes)</code>
Topic Distributions	<code>topic_model.approximate_distribution(abstracts)</code>

Figure 3-22. -X Topic Modeling Variations in BERTopic

Summary

In this chapter we discussed a cluster-based method for topic modeling, BERTopic. By leveraging a modular structure, we used a variety of Large Language Models to create document representations and fine-tune topic representations. We extracted the topics found in ArXiv abstracts and saw how we could use BERTopic's modular structure to develop different kinds of topic representations.

Chapter 4. Text Generation with GPT Models

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. *In particular, some of the formatting may not match the description in the text: this will be resolved when the book is finalized.*

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

In the first chapters of this book, we have taken our first steps into the world of Large Language Models (LLMs). We delved into various applications, such as classification and semantic search, employing models that focus on representing text, like BERT and

its derivatives.

As we progressed, we used models trained primarily for text generation, models that are often referred to as Generative Pre-trained Transformers (GPT). These models have the remarkable ability to generate text in response to *prompts* from the user. Through *prompt engineering*, we can design these prompts in a way that enhances the quality of the generated text.

In this chapter, we will explore these generative models in more detail and dive into the realm of prompt engineering, reasoning with generative models, verification and even evaluating their output.

Using Text Generation Models

Before we start with the fundamentals of prompt engineering, it is essential to explore the basics of utilizing a text generation model. How do we select the model to use? Do we use a proprietary or open-source model? How can we control the generated output? These questions will serve as our stepping stones into using text generation models.

Choosing a Text Generation Model

Choosing a text generation model starts with choosing between proprietary models or open-source models. Although proprietary models are generally more performant, we focus in this book more on open-source models as they offer more flexibility and are free to use.

[Figure 4-1](#) shows a small selection of impactful foundation models, LLMs that have been pre-trained on vast amounts of text data and are often fine-tuned for specific applications.

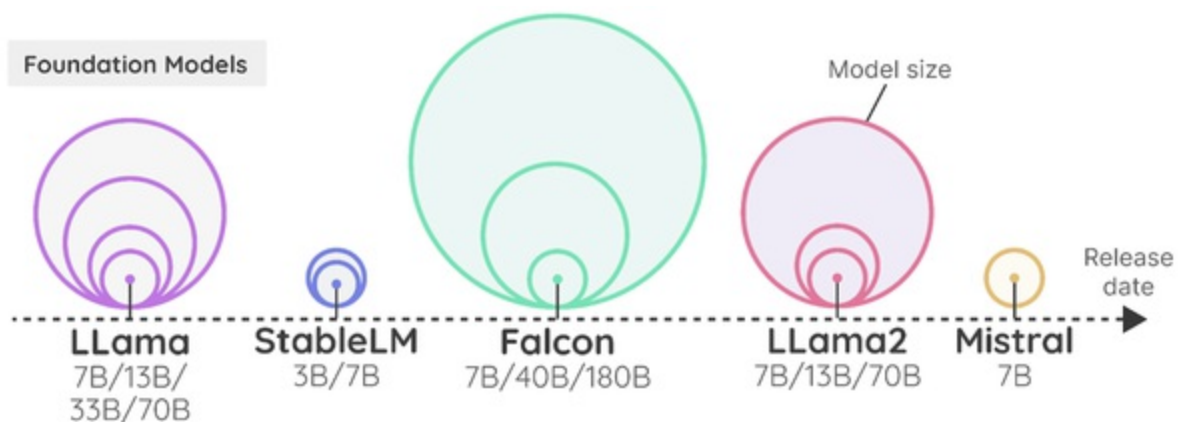


Figure 4-1. Foundation models

From those foundation models, hundreds if not thousands of models have been fine-tuned, one more suitable for certain tasks than another. Choosing the model to use can be a daunting task!

We generally advise starting out with a small and recently released foundation model, like Llama 2 or Mistral 7B in the illustration in [Figure 4-1](#). This allows for quick iteration and as a

result a thorough understanding of whether the model is suitable for your use case. Moreover, a smaller model requires less GPU memory (VRAM) which makes it easier and faster to run if you do not have a large GPU. Scaling up tends to be a nicer experience than scaling down.

In the examples throughout this chapter, we will employ a model from the Zephyr family, namely [Zephyr 7B-beta](#). They are models fine-tuned on Mistral 7B, a relatively small but quite capable open-source LLM.

If you're taking your first steps in generative AI, it's important to start with a smaller model. This provides a great introduction and lays a solid foundation for progressing to larger models.

Loading a Text Generation Model

“How to load a text generation model” can actually be a chapter by itself. There are dozens of packages out there each with their compression and inference strategies to squeeze out performance.

The most straightforward method of doing so is through the well-known HuggingFace Transformers library:

```

import torch
from transformers import pipeline
# Load our model
pipe = pipeline(
    "text-generation",
    model="HuggingFaceH4/zephyr-7b-beta",
    torch_dtype=torch.bfloat16,
    device_map="auto"
)

```

To use the model, we will have to take a closer look at its prompt template. Any LLM requires a specific template so that it can differentiate between recent and older query/answer pairs.

To illustrate, let us ask the LLM to make a joke about chickens:

```

def format_prompt(query="", messages=False):
    """Use the internal chat template to format our qu
    # The system prompt (what the LLM should know befo
    if not messages:
        messages = [
            {
                "role": "system",
                "content": "You are a helpful assistan
            },
            {"role": "user", "content": query},
        ]

```

```
# We apply the LLMs internal chat template to our
prompt = pipe.tokenizer.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True
)
return prompt
prompt = format_prompt("Write a short joke about chick
```

Aside from our main prompt, we also generated a system prompt that provides context or guidance to an LLM for generating the response. As illustrated in [Figure 4-2](#), the prompt template helps the LLM understand the difference between types of prompts and also between text generated by the LLM and the user.

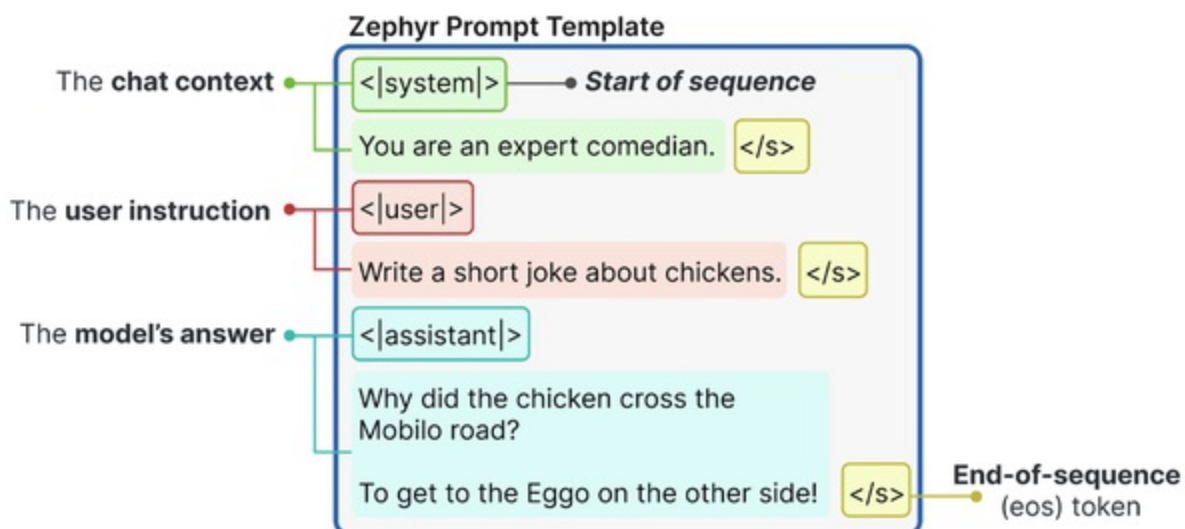


Figure 4-2. The template Zephyr expects when interacting with the model.

Using that prompt, we can let the LLM give an answer:

```
# Generate the output
outputs = pipe(
    prompt,
    max_new_tokens=256,
    do_sample=True,
    temperature=0.1,
    top_p=0.95
)
print(outputs[0]["generated_text"])
```

Which outputs:

```
"""
<|system|>
You are a friendly chatbot.</s>
<|user|>
Write a joke about chickens.</s>
<|assistant|>
Why did the chicken cross the Mobilo?
Because the Eggspressway was closed for pavement!
"""
```

Now that we know how to create a prompt using a chat template, let us explore how we can control the output of the model.

Controlling the Model Output

Other than prompt engineering, we can control the kind of output that we want by adjusting the model parameters. In our previous example, you might have noticed that we used several parameters in the `pipe` function, including `temperature` and `top_p`.

These parameters control the randomness of the output. A part of what makes LLMs exciting technology is that it can generate different responses for the exact same prompt. Each time an LLM needs to generate a token, it assigns a likelihood number to each possible token.

As illustrated in [Figure 4-3](#), in the sentence “*I am driving a...*” the likelihood of that sentence being followed by tokens like “*car*” or “*truck*” is generally higher than a token like “*elephant*”.

However, there is still a possibility of “*elephant*” being generated but it is much lower.

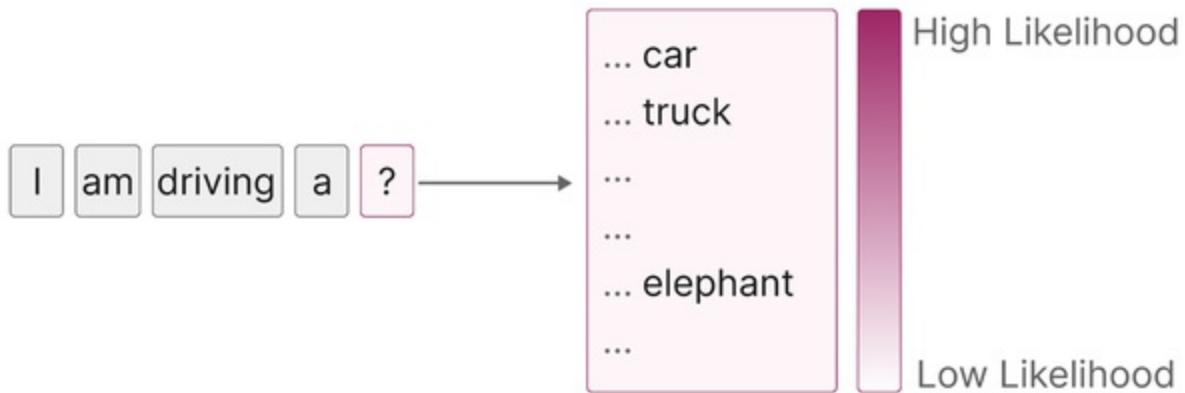


Figure 4-3. The model chooses the next token to generate based on their likelihood scores.

Temperature

The `temperature` controls the randomness or the creativity of the text generated. It defines how likely it is to choose tokens that are less probable. The underlying idea is that a temperature of 0 generates the same response every time because it always chooses the most likely word. As illustrated in [Figure 4-4](#), a higher value allows less probable words to be generated.

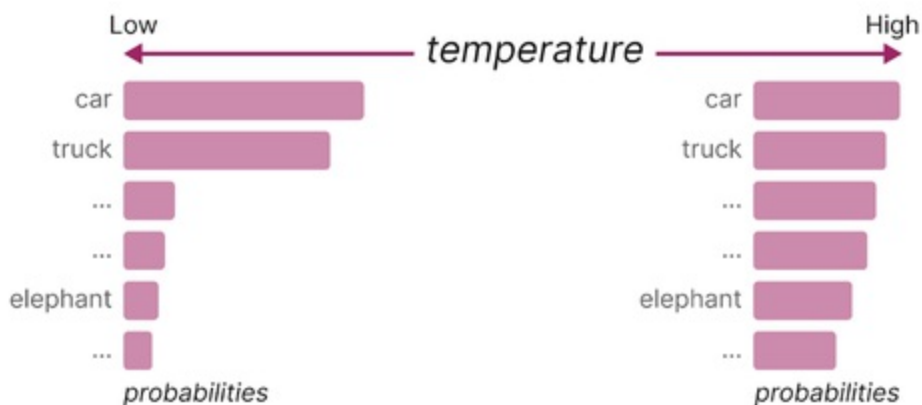


Figure 4-4. A higher temperature increases the likelihood that less probable tokens are generated, and vice versa.

As a result, a higher temperature (e.g., 0.8) generally results in a more diverse output while a lower temperature (e.g., 0.2) creates a more deterministic output.

top_p

`top_p`, also known as nucleus sampling, is a sampling technique that controls which subset of tokens (the nucleus) the LLM can consider. It will consider tokens until it reaches their cumulative probability. If we set `top_p` to 0.1, it will consider tokens until it reaches that value. If we set `top_p` to 1, it will consider all tokens.

As shown in [Figure 4-5](#), by lowering the value, it will consider fewer tokens and generally give less “creative” output whilst increasing the value allows the LLM to choose from more tokens.

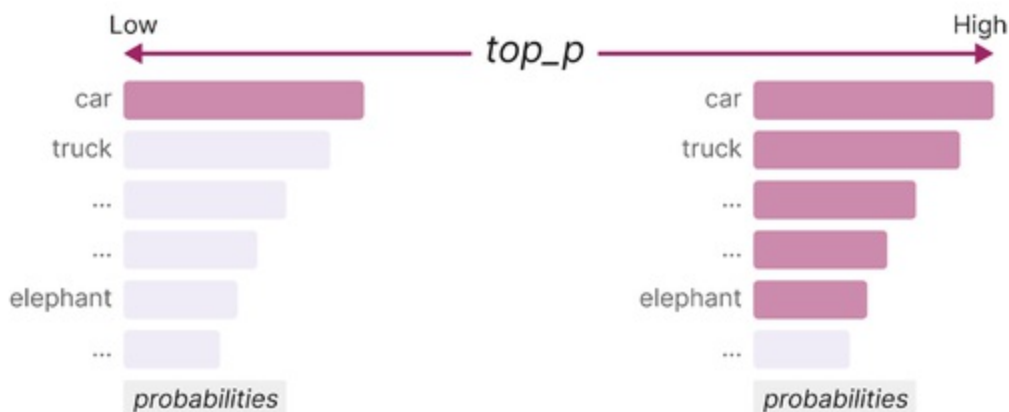


Figure 4-5. A higher `top_p` increases the number of tokens that can be selected to generate, and vice versa.

Similarly, the `top_k` parameter controls exactly how many tokens the LLM can consider. If you change its value to 100, the LLM will only consider the top 100 most probable tokens.

As shown in Table 5-1, these parameters allow the user to have a sliding scale between being creative (high `temperature` and `top_p`) and being predictable (lower `temperature` and `top_p`).

Example Use Case	Temperature	Top_p	Description
Brainstorming Session	High	High	High randomness with large pool of potential tokens. The results will be highly diverse and often leading to very creative and unexpected outputs.
Email Generation	Low	Low	Deterministic output with high probable predicted tokens. This results in predictable, focused, and conservative outputs.
Creative Writing	High	Low	High randomness with a small pool of potential tokens. This combination produces creative outputs but still remains coherent.
Translation	Low	High	Deterministic output with high probable predicted tokens. Produces coherent output with a wider range of vocabulary, leading to outputs with linguistic variety.

Figure 4-6. Examples of use cases when selecting values for `temperature` and `top_p`.

Intro to Prompt Engineering

An essential part of working with text-generative LLMs is prompt engineering. By carefully designing our prompts we can guide the LLM to generate desired responses. Whether the prompts are

questions, statements, or instructions, the main goal of prompt engineering is to elicit a useful response from the model.

However, prompt engineering is also more than just designing effective prompts. It can be used as a tool to evaluate the output of a model, design safeguards, and safety mitigation methods. This is an iterative process of prompt optimization and requires experimentation. There is not and unlikely will ever be a perfect prompt design.

In this section, we will go through common methods for prompt engineering, and small tips and tricks to understand what the effect is of certain prompts. These skills allow us to understand the capabilities of LLMs and lie at the foundation of interfacing with these kinds of models.

We begin by answering the question: What should be in a prompt?

The Basic Ingredients of a Prompt

An LLM is a prediction machine. Based on a certain input, the prompt, it tries to predict the words that might follow it. At its core, and as illustrated in [Figure 4-7](#), the prompt does not need to be more than just a few words to elicit a response from the LLM.

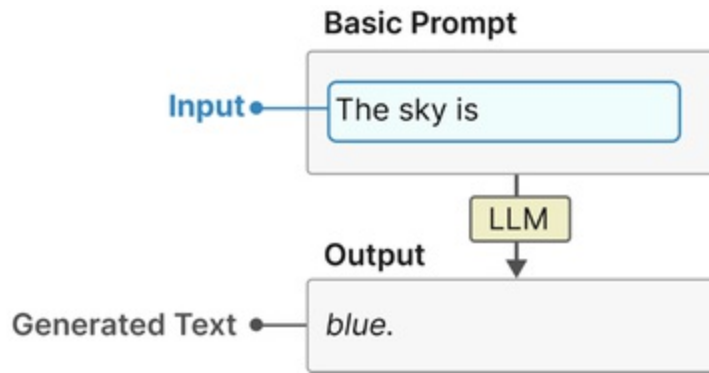


Figure 4-7. A basic example of a prompt. No instruction is given so the LLM will simply try to complete the sentence.

However, although the illustration works as a basic example, it fails to complete a specific task. Instead, we generally approach prompt engineering by asking a specific question or task the LLM should complete. To elicit the desired response, we need a more structured prompt.

For example, and as shown in [Figure 4-8](#), we could ask the LLM to classify a sentence into either having positive or negative sentiment.

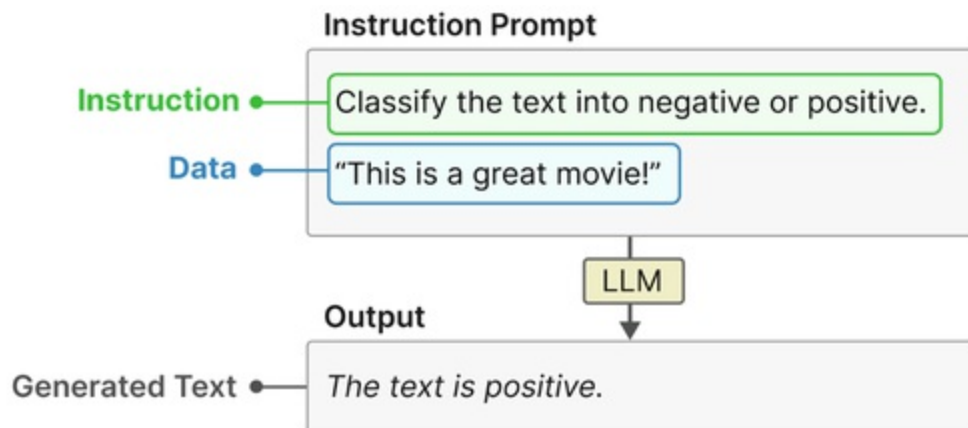


Figure 4-8. Two components of a basic instruction prompt, the instruction itself and the

data it refers to.

This extends the most basic prompt to one consisting of two components—the instruction itself and the data that relates to the instruction.

More complex use cases might require more components to be necessary in a prompt. For instance, to make sure the model only outputs “negative” or “positive” we can introduce output indicators that help guide the model. In [Figure 4-9](#), we prefix the sentence with “Text:” and add “Sentiment:” to prevent the model from generating a complete sentence. Instead, this structure indicates that we expect either “negative” or “positive”.

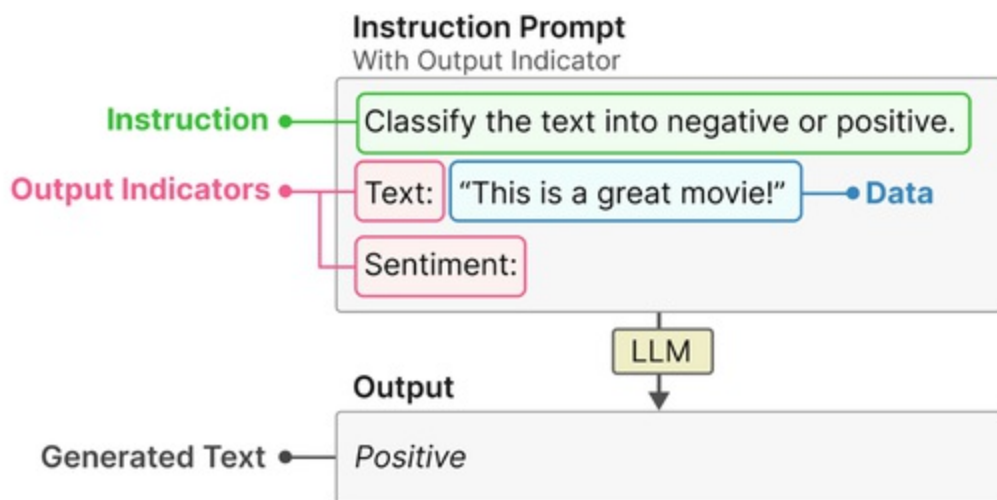


Figure 4-9. Extending the prompt with an output indicator which allows for a specific output.

We can continue adding or updating the elements of a prompt until we elicit the response we were looking for. We could add

additional examples, describe the use case in more detail, provide additional context, etc. These components are merely examples and are not a limited set of possibilities. The creativity that comes with designing these components is key.

Although a prompt is a single piece of text it is tremendously helpful to think of prompts as pieces of a larger puzzle. Have I described the context of my question? Does the prompt have an example of the output?

Instruction-based Prompting

Although prompting comes in many flavors, from discussing philosophy with the LLM to role-playing with your favorite superhero, prompting is often used to have the LLM answer a specific question or resolve a certain task. This is referred to as instruction-based prompting.

[Figure 4-10](#) illustrates a number of use cases in which instruction-based prompting plays an important role. We already did one of these in the previous example, namely supervised classification.

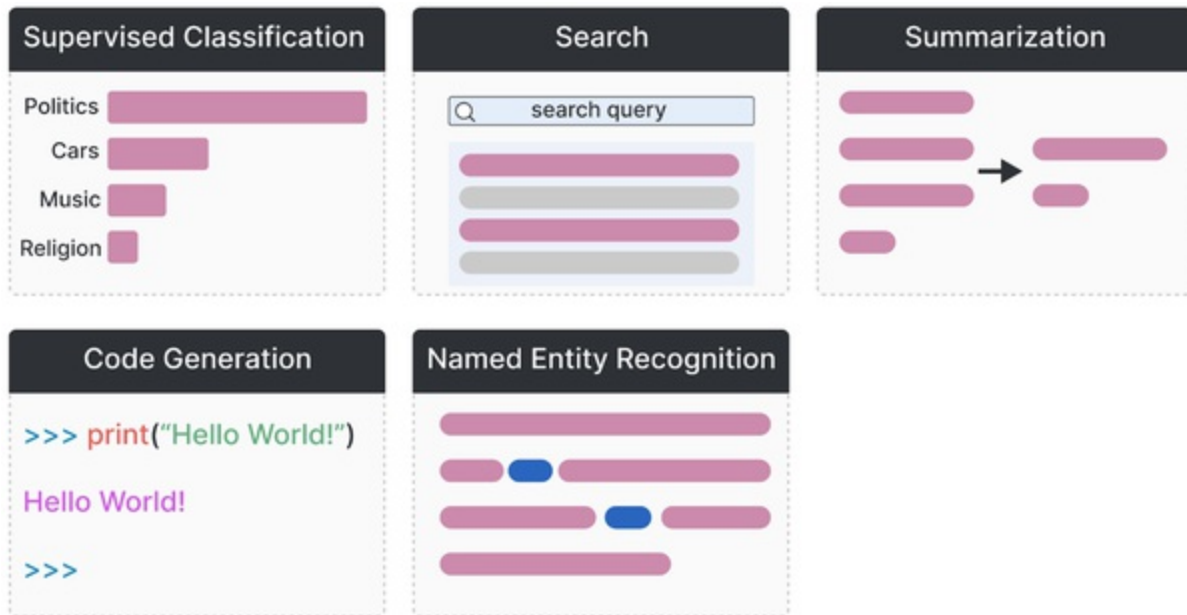


Figure 4-10. Examples of use cases that employ instruction-based prompting.

Each of these tasks requires different formats of prompting and more specifically, different questions to be asked of the LLM. Asking the LLM to summarize a piece of text will not suddenly result in classification. To illustrate, examples of prompts for some of these use cases can be found in [Figure 4-11](#).

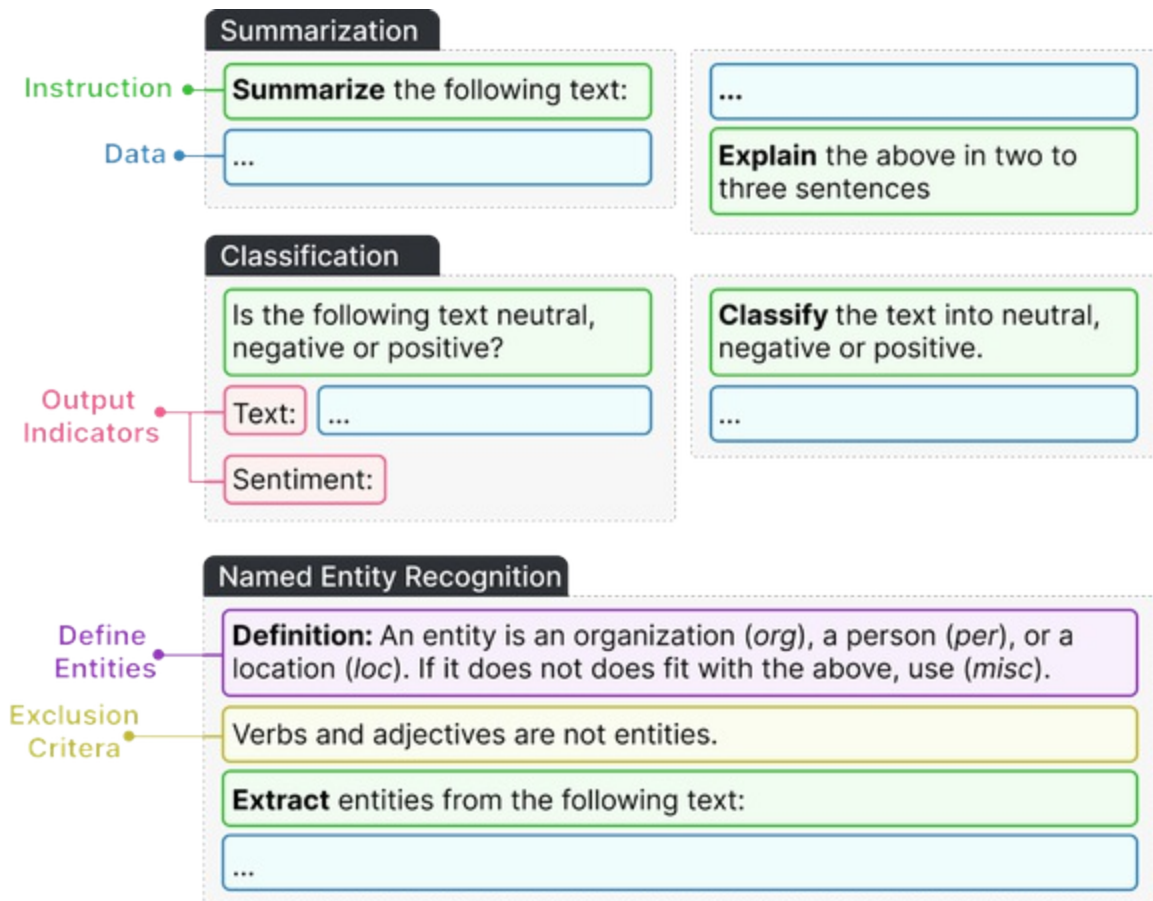


Figure 4-11. Prompt examples of common use cases. Notice how within a use case, the structure and location of the instruction can be changed.

Although these tasks require different instructions, there is actually a lot of overlap in the prompting techniques used to improve the quality of the output. A non-exhaustive list of these techniques includes:

Specificity

Accurately describe what you want to achieve. Instead of asking the LLM to “Write a description for a product.” ask it to “Write a description for a product in less than two sentences

and use a formal tone.”.

Hallucination

LLMs may generate incorrect information confidently, which is referred to as hallucination. To reduce its impact, we can ask the LLM to only generate an answer if it knows the answer. If it does not know the answer, respond with “I don’t know”.

Order

Either begin or end your prompt with the instruction. Especially with long prompts, information in the middle is often forgotten. LLMs tend to focus on information either at the beginning of a prompt (primacy effect) or the end of a prompt (recency effect).

Here, specificity is arguably the most important aspect. An LLM does not know what you want unless you are specific in what you want to achieve and why.

Advanced Prompt Engineering

On the surface, creating a good prompt might seem straightforward. Ask a specific question, be accurate, add some examples and you are done! However, prompting can grow complex quite quickly and as a result is an often underestimated

component of leveraging LLMs.

Here, we will go through several advanced techniques for building up your prompts, starting with the iterative workflow of building up complex prompts all the way to using LLMs sequentially to get improved results. Eventually, we will even build up to advanced reasoning techniques.

The Potential Complexity of a Prompt

As we explored in the intro to prompt engineering, a prompt generally consists of multiple components. In our very first example, our prompt consisted of instruction, data, and output indicators. As we mentioned before, no prompt is limited to just these three components and you can build it up as complex as you want.

These advanced components can quickly make a prompt quite complex. Some common components are:

Persona

Describe what role the LLM should take on. For example, use “*You are an expert in astrophysics.*” if you want to ask a question about astrophysics.

Instruction

The task itself. Make sure this is as specific as possible. We do not want to leave much room for interpretation.

Context

Additional information describing the context of the problem or task. It answers questions like “*What is the reason for the instruction?*”.

Format

The format the LLM should use to output the generated text. Without it, the LLM will come up with a format itself which is troublesome in automated systems.

Audience

For whom the generated text should be. This also describes the level of the generated output. For education purposes, it is often helpful to use ELI5 (“*Explain it like I’m 5.*”)

Tone

The tone of voice the LLM should use in the generated text. If you are writing a formal email to your boss, you might not want to use an informal tone of voice.

Data

The main data related to the task itself.

To illustrate, let us extend the classification prompt that we had earlier and use all of the above components. This is demonstrated in [Figure 4-12](#).

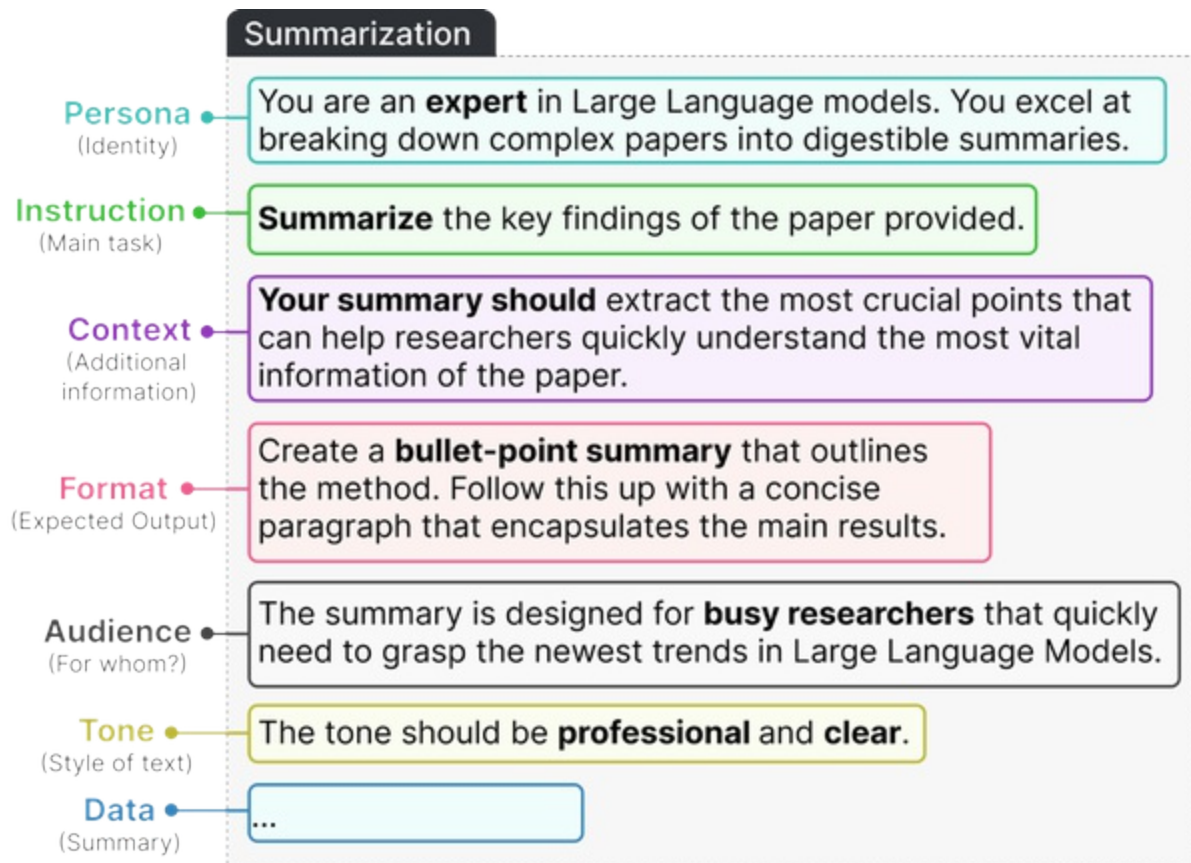


Figure 4-12. An example of a complex prompt with many components.

This complex prompt demonstrates the modular nature of prompting. We can add and remove components freely and judge their effect on the output. As illustrated in [Figure 4-13](#), we can slowly build up our prompt and explore the effect of each change.

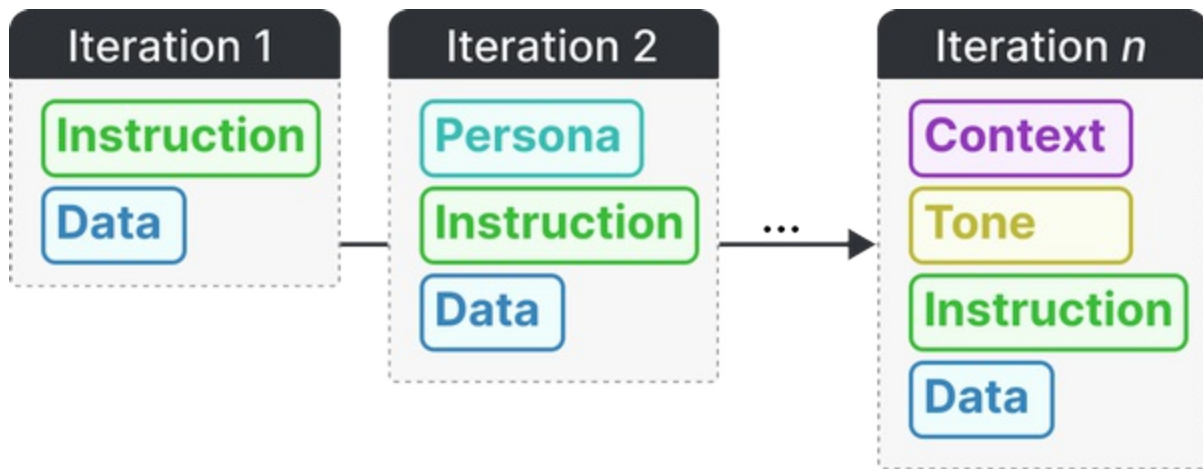


Figure 4-13. Iterating over modular components is a vital part of prompt engineering.

The changes are not limited to simply introducing or removing components. Their order, as we saw before with the recency and primacy effects, can affect the quality of the LLM's output.

In other words, experimentation is vital when finding the best prompt for your use case. With prompting, we essentially have ourselves in an iterative cycle of experimentation.

Try it out yourself! Use the complex prompt to add and/or remove parts to observe its impact on the generated prompts. You will quickly notice when pieces of the puzzle are worth keeping. You can use your own data by adding it to the `data` variable:

```
# Prompt components
persona = "You are an expert in Large Language models.
instruction = "Summarize the key findings of the paper
```

```
context = "Your summary should extract the most crucial  
data_format = "Create a bullet-point summary that outlines  
audience = "The summary is designed for busy researchers  
tone = "The tone should be professional and clear.\n"  
data = "Text to summarize: PUT_THE_DATA_TO_SUMMARIZE_HERE"  
# The full prompt - remove and add pieces to view its structure  
query = persona + instruction + context + data_format  
prompt = format_prompt(query)
```

TIP

Almost weekly there are new components of a prompt that might increase the accuracy of the output. There are all manners of components that we could add and creative components like using emotional stimuli (e.g., “This is very important for my career.”) are discovered on a weekly basis.

Part of the fun in prompt engineering is that you can be as creative as possible to figure out which combination of prompt components contribute to your use case. There are few constraints to develop a format that works for you.

However, note that some prompts work better for certain models compared to others as their training data might be different or if they are trained for different purposes.

In-Context Learning: Providing Examples

In the previous sections, we tried to accurately describe what the LLM should do. Although accurate and specific descriptions help

the LLM to understand the use case, we can go one step further.

Instead of describing the task, why do we not just show the task?

We can provide the LLM with examples of exactly the thing that we want to achieve. This is often referred to as in-context learning, where we provide the model with correct examples.

As illustrated in [Figure 4-14](#), this comes in a number of forms depending on how many examples you show the LLM. Zero-shot prompting does not leverage examples, one-shot prompts use a single example, and few-shot prompts use two or more examples.

Zero-shot Prompt

Prompting without examples

Classify the text into neutral, negative or positive.

Text: I think the food was okay.
Sentiment: ...

One-shot Prompt

Prompting with a single example

Classify the text into neutral, negative or positive.

Text: I think the food was alright.
Sentiment: Neutral

Text: I think the food was okay.
Sentiment:

Few-shot Prompt

Prompting with more than one example

Classify the text into neutral, negative or positive.

Text: I think the food was alright.
Sentiment: Neutral

Text: I think the food was great!
Sentiment: Positive

Text: I think the food was horrible...
Sentiment: Negative

Text: I think the food was okay.
Sentiment:

Figure 4-14. An example of a complex prompt with many components.

Adopting the original phrase, we believe that “an example is worth a thousand words”. These examples provide a direct example of what and how the LLM should achieve.

We can illustrate this method with a simple example taken from the original paper describing this method. The goal of the prompt is to generate a sentence with a made-up word. To improve the quality of the resulting sentence, we can show the generative model an example of what a proper sentence with a made-up word would be.

To do so, we will need to differentiate between our question (user) and the answers that were provided by the model (assistant):

```
# Use a single example of using the made-up word in a
one_shot_prompt = format_prompt(messages=[
    {"role": "user", "content": "Q: A 'Gigamuru' is a"},
    {"role": "assistant", "content": "A: I have a Giga"},
    {"role": "user", "content": "Q: To 'screeg' someth
])
print(one_shot_prompt)
```

The prompt illustrates the need to differentiate between the user and assistant. If we did not, it would seem as if we were talking

to ourselves:

```
"""  
<|user|>  
Q: A 'Gigamuru' is a type of Japanese musical instrume  
<|assistant|>  
A: I have a Gigamuru that my uncle gave me as a gift.  
<|user|>  
Q: To 'screeg' something is to swing a sword at it. An  
<|assistant|>  
"""
```

We can use this prompt to run our model:

```
# Run generative model  
outputs = pipe(one_shot_prompt, max_new_tokens=64, do_  
print(outputs[0]["generated_text"]))
```

The result is a proper sentence using the made-up word “screeg”:

```
"A: I screeged the dragon's tail with my sword, but it
```

As with all prompt components, one or few shot-prompting is not the be-all and end-all of prompt engineering. We can use it as

one piece of the puzzle to further enhance the descriptions that we gave it. The model can still “choose”, through random sampling, to ignore the instructions

Chain Prompting: Breaking up the Problem

In previous examples, we explored splitting up prompts into modular components to improve the performance of LLMs. Although this works well for many use cases, this might not be feasible for highly complex prompts or use cases.

Instead of breaking the problem within a prompt, we can do so between prompts. Essentially, we take the output of one prompt and use it as input for the next. Thereby creating a continuous chain of interactions that solves our problem.

To illustrate, let us say we want to use an LLM to create a product name, slogan, and sales pitch for us based on a number of product features. Although we can ask the LLM to do this in one go, we can instead break the problem up into pieces.

As a result, and as illustrated in [Figure 4-16](#), we get a sequential pipeline that first creates the product name, uses that with the product features as input to create the slogan, and finally, uses the features, product name, and slogan to create the sales pitch.

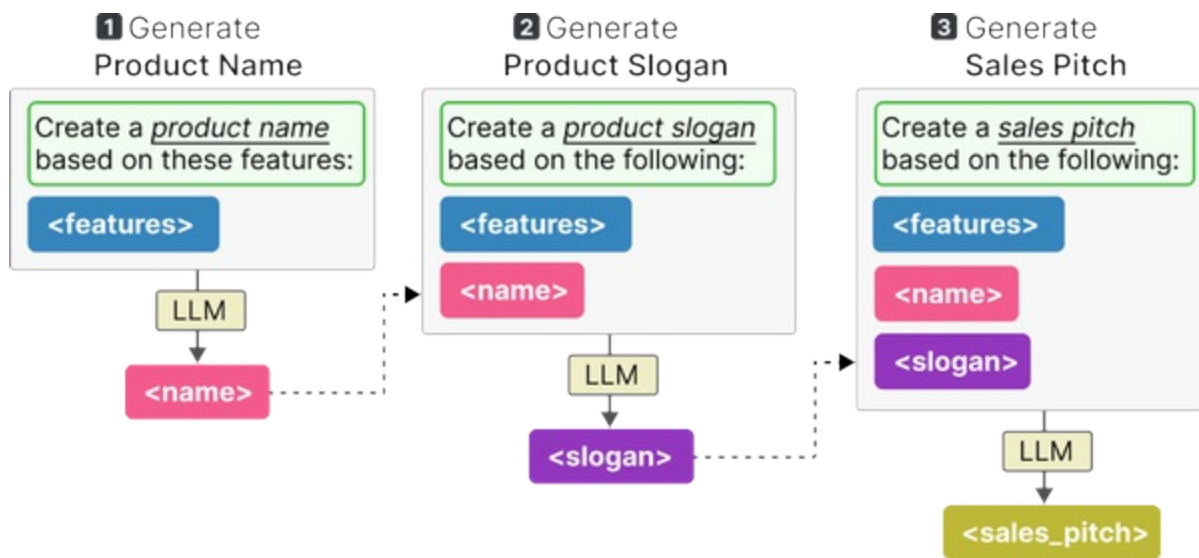


Figure 4-15. Using a description of a product's features, chain prompts to create a suitable name, slogan, and sales pitch.

This technique of chaining prompts allows the LLM to spend more time on each individual question instead of tackling the whole problem.

Let us illustrate this with a small example.

```
# Create name and slogan for a product
product_prompt = format_prompt("Create a name and slog
outputs = pipe(product_prompt, max_new_tokens=32, do_s
product_description = outputs[0]["generated_text"]
# Use name and slogan as input for a sales pitch
sales_prompt = format_prompt(f"What would be a good sa
outputs = pipe(sales_prompt, max_new_tokens=128, do_sa
sales_pitch = outputs[0]["generated_text"]
# Results
print(product_description)
```

```
print(sales_pitch)
```

In this example, we ask the model first to create a name and slogan. Then, we can use the output to ask for a good sales pitch based on the product's characteristics.

This gives us the following output:

```
"""
Name: LLM Assistant
Slogan: "Your go-to chatbot powered by cutting-edge la
Introducing LLM Assistant, the revolutionary chatbot t
"""
```

Although we need two calls to the model, a major benefit is that we can give each call different parameters. For instance, the number of tokens created was relatively small for the name and slogan whereas the pitch can be much longer.

It can be used for a variety of use cases, including:

Response validation

Asking the LLM to double-check previously generated outputs

Parallel prompts

Create multiple prompts in parallel and do a final pass to merge them. For example, ask multiple LLMs to generate multiple recipes in parallel and use the combined result to create a shopping list.

Writing stories

Leverage the LLM to write books or stories by breaking the problem down into components. For example, by first writing a summary, develop characters and build the story beats before diving into creating the dialogue.

In Chapter 6, we will go beyond chaining LLMs and chain other pieces of technology together, like memory, search, and more! Before that, this idea of prompt chaining will be explored further in the next sections describing more complex prompt chaining methods like self-consistency, chain-of-thought, and tree-of-thought.

Reasoning with Generative Models

In the previous sections, we focused mostly on the modular component of prompts, building them up through iteration. These advanced prompt engineering techniques, like prompt chaining, proved to be the first step toward enabling complex

reasoning with generative models.

To allow for this complex reasoning, it is a good moment to step back and explore what reasoning entails. To simplify, our methods of reasoning can be divided into system 1 and 2 thinking processes, as illustrated in Figure 5-X.

System 1 thinking represents automatic, intuitive, and near-instantaneous. It shares similarities with generative models that automatically generate tokens without any self-reflective behavior. In contrast, systems 2 thinking is a conscious, slow, and logical process, akin to brainstorming and self-reflection.

If we could give a generative model the ability of self-reflection, we would essentially be emulating the system 2 way of thinking which tends to produce more thoughtful responses than system 1 thinking.

In this section, we will explore several techniques that attempt to mimic these kinds of thought processes of human reasoners with the aim of improving the output of the model.

Chain-of-Thought: Think Before Answering

The first and major step towards complex reasoning in generative models was through a method called Chain-of-

Thought (CoT). CoT aims to have the generative model “think” first rather than answering the question directly without any reasoning.

As illustrated in [Figure 4-16](#), it provides examples in a prompt that demonstrate the reasoning the model should do before generating its response. These reasoning processes are referred to as “thoughts”. This helps tremendously for tasks that involve a higher degree of complexity, like mathematical questions. Adding this reasoning step allows the model to distribute more compute over the reasoning process.

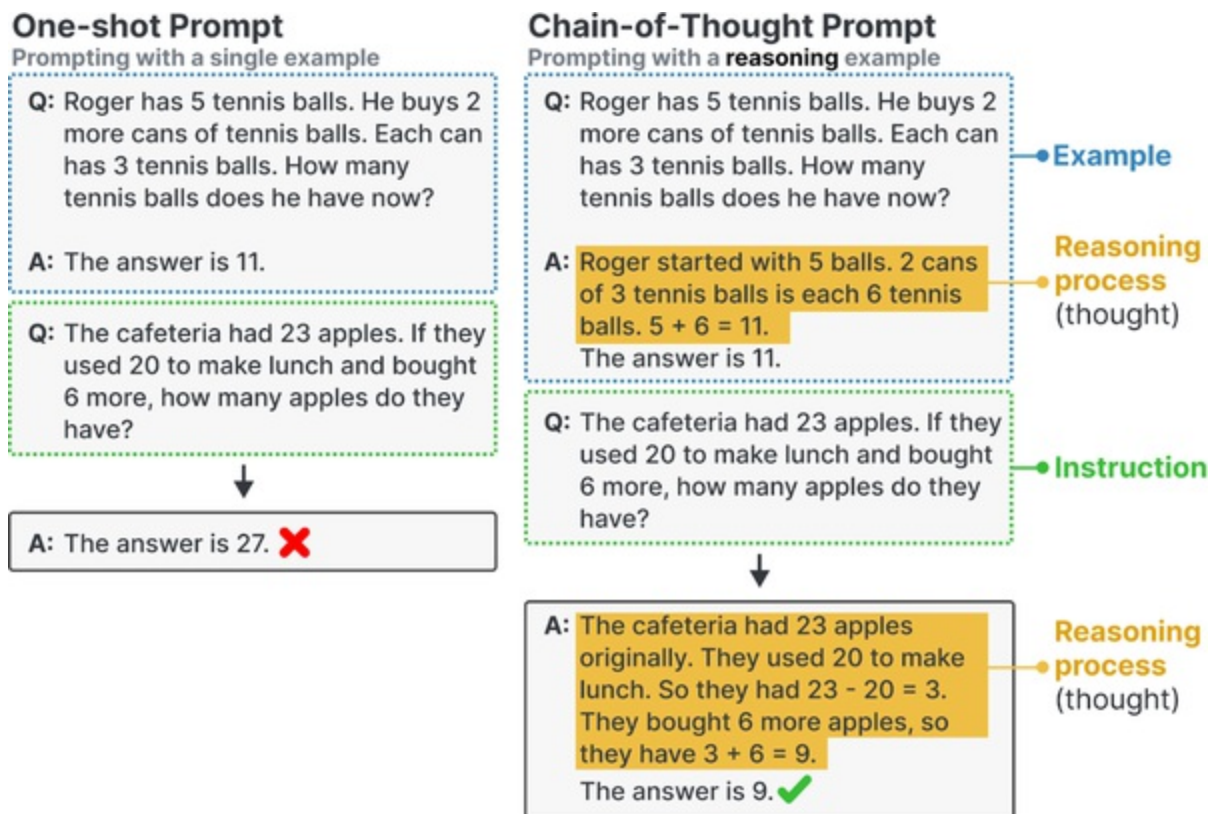


Figure 4-16. Chain-of-Thought prompting uses reasoning examples to persuade the generative model to use reasoning in its answer.

We will use the example they used in their paper to demonstrate this phenomenon. To start with, let's explore the output of a standard prompt without CoT. Instead of providing a single query, we differentiate between the user and the assistant when providing examples:

```
# Answering without explicit reasoning
standard_prompt = format_prompt(messages=[
    {"role": "user", "content": "Q: Roger has 5 tennis"},
    {"role": "assistant", "content": "A: The answer is"},
    {"role": "user", "content": "Q: The cafeteria had"}
])
# Run generative model
outputs = pipe(standard_prompt, max_new_tokens=64, do_
print(outputs[0]["generated_text"]))
```

This gives us the incorrect answer:

```
"A: The answer is 26."
```

Instead, we will use CoT to have the model present its reasoning before giving the answer:

```
# Answering with chain-of-thought
cot_prompt = format_prompt(messages=[
```

```
    {"role": "user", "content": "Q: Roger has 5 tennis  
    {"role": "assistant", "content": "A: Roger started  
    {"role": "user", "content": "Q: The cafeteria had  
  ])  
# Run generative model  
outputs = pipe(cot_prompt, max_new_tokens=256, do_samp  
print(outputs[0]["generated_text"])
```

This time, we got the correct response:

```
"A: Initially, there were 23 apples. They used 20 appl
```

This reasoning process is especially helpful because the model does so before generating the answer. By doing so, it can leverage the knowledge it has generated thus far to compute the correct answer.

Zero-shot Chain-of-Thought

Although CoT is a great method for enhancing the output of a generative model, it does require one or more examples of reasoning in the prompt which the user might not have access to.

Instead of providing examples, we can simply ask the generative model to provide the reasoning. There are many different forms

that work but a common and effective method is to use the phrase “Let’s think step-by-step” which is illustrated in [Figure 4-17](#).

Zero-shot Chain-of-Thought

Prompting without examples

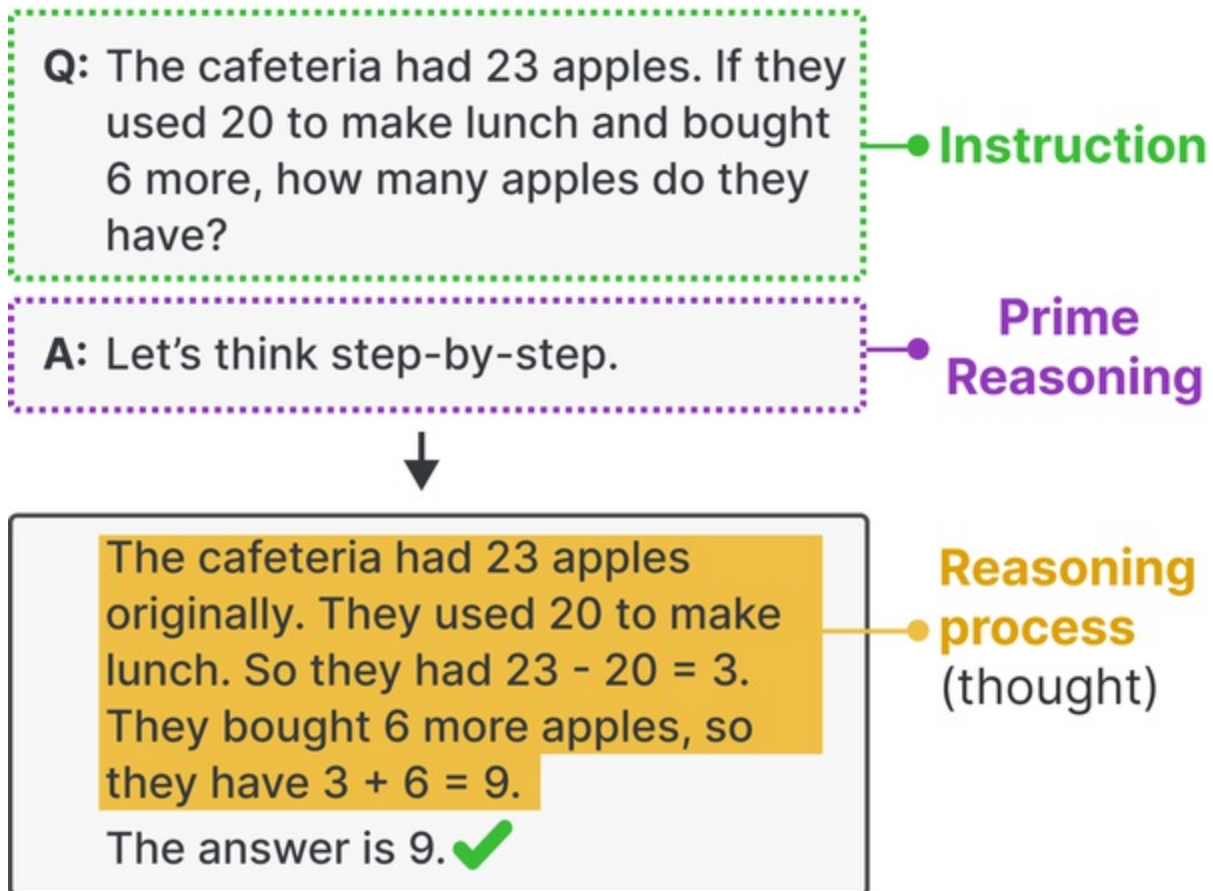


Figure 4-17. Chain-of-Thought prompting without using examples. Instead, it uses the phrase “Let’s think step-by-step” to prime reasoning in its answer.

Using the example we used before, we can simply append that phrase to the prompt to enable CoT-like reasoning:

```
# Zero-shot Chain-of-Thought
zeroshot_cot = format_prompt(
    "The cafeteria had 23 apples. If they used 20 to m
)
# Run generative model
outputs = pipe(zeroshot_cot, max_new_tokens=512, do_sam
print(outputs[0]["generated_text"])
```

Again, we got the correct response but now without needing to provide examples:

```
"""
1. We start with the original number of apples in the
2. We determine how many apples were used to make lunch
3. We subtract the number of apples used to make lunch
4. We purchase 6 more apples:  $3 + 6 = 9$ 
5. So the total number of apples now in the cafeteria
6. We can confirm that the calculation is correct by c
"""
```

This is why it is so important to “show your work” when doing calculations. By addressing the reasoning process we can justify the answer and be more sure of the answer.

Although the prompt “*Let’s think step-by-step*” can improve the output, you are not constrained by this exact formulation. Alternatives exist like “*Take a deep breath and think step-by-step*” and “*Let’s work through this problem step-by-step*”. The authors demonstrated the usefulness of coming up with alternative formulations.

Self-Consistency: Sampling Outputs

Using the same prompt multiple times can lead to different results if we allow for a degree of creativity through parameters like `temperature` and `top_p`. As a result, the quality of the output might improve or degrade depending on the random selection of tokens. In other words, luck!

To counteract this degree of randomness and improve the performance of generative models, self-consistency was introduced. This method asks the generative model the same prompt multiple times and takes the majority result as the final answer. During this process, each answer can be affected by different `temperature` and `top_p` values to increase the diversity of sampling.

As illustrated in [Figure 4-18](#), this method can further be improved by adding Chain-of-Thought prompting to improve its reasoning whilst only using the answer for the voting procedure.

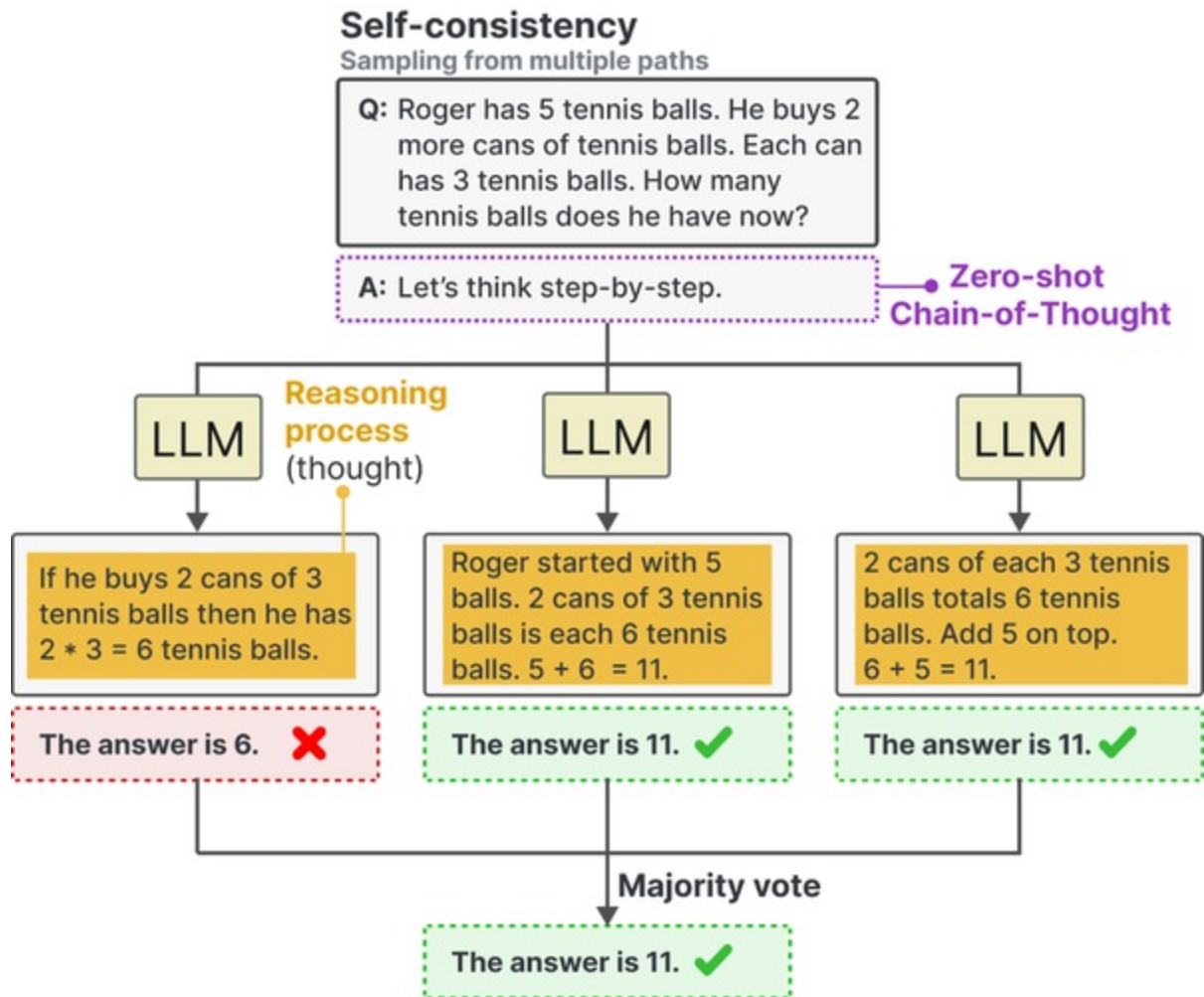


Figure 4-18. By sampling from multiple reasoning paths, we can use majority voting to extract the most likely answer.

Although this method works quite well to improve the output, it does require a single question to be asked multiple times. As a result, although the method can improve performance, it becomes n times slower where n is the number of output samples.

Tree-of-Thought: Exploring Intermediate

Steps

The ideas of Chain-of-Thought and Self-consistency are meant to enable more complex reasoning. By sampling from multiple “thoughts” and making them more thoughtful, we aim to improve the output of generative models.

These techniques scratch only the surface of what is currently being done to enable this complex reasoning. An improvement to these approaches can be found in Tree-of-Thought which allows for an in-depth exploration of several ideas.

The method works as follows. When faced with a problem that requires multiple reasoning steps, it often helps to break it down into pieces. At each step, and as illustrated in [Figure 4-19](#), the generative model is prompted to explore different solutions to the problem at hand. It then votes for the best solution and then continues to the next step.

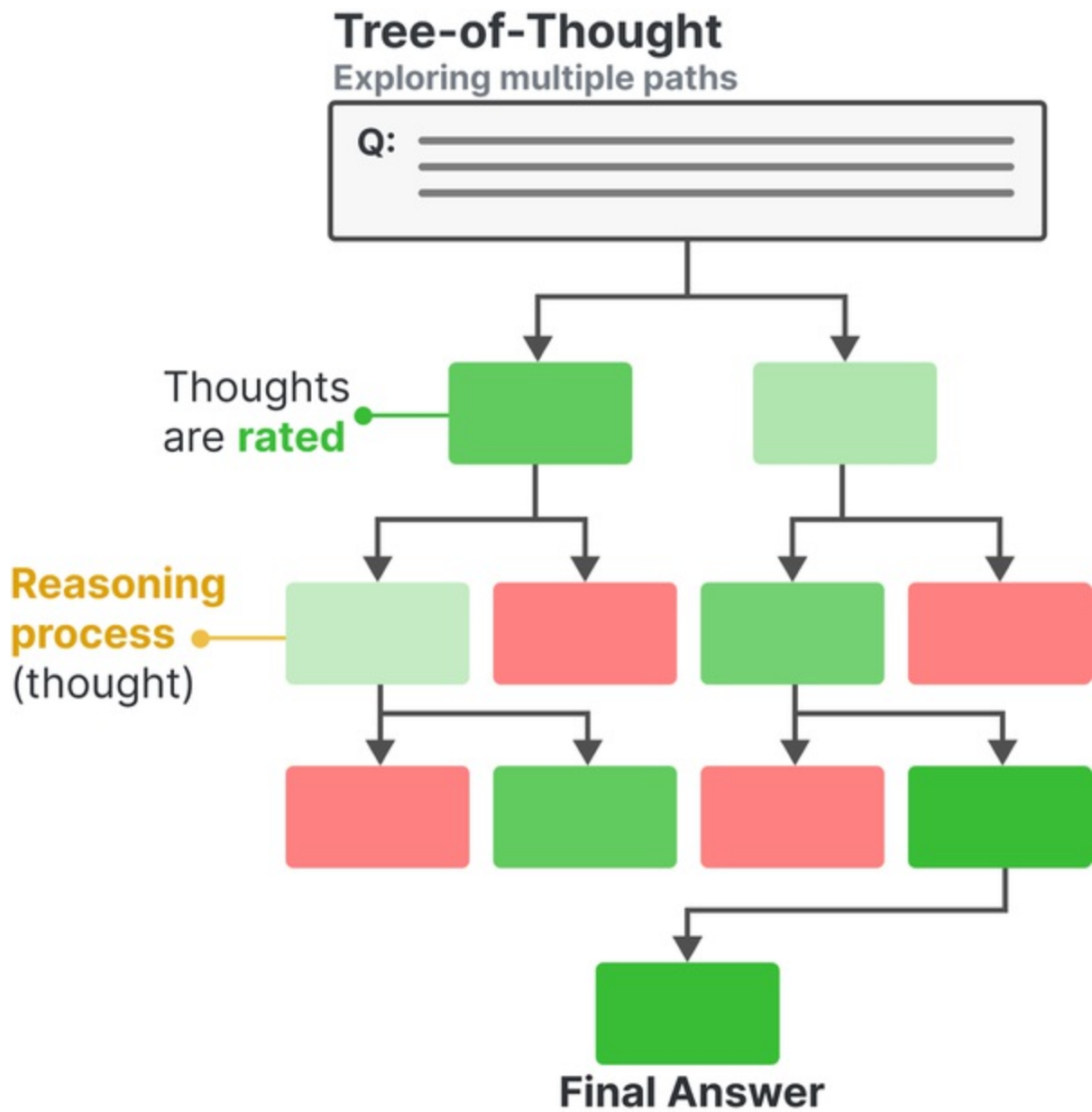


Figure 4-19. By leveraging a tree-based structure, generative models can generate intermediate thoughts to be rated. The most promising thoughts are kept and the lowest are pruned.

This method is tremendously helpful when needing to consider multiple paths, like when writing a story or coming up with creative ideas.

A disadvantage of this method is that it requires many calls to the generative models which slows the application significantly.

Fortunately, there has been a successful attempt to convert the Tree-of-Thought framework into a simple prompting technique.

Instead of calling the generative model multiple times, we ask the model to mimic that behavior by emulating a conversation between multiple experts. These experts will question each other until they reach a consensus. An example of a Tree-of-Thought prompt is:

```
# Zero-shot Chain-of-Thought
zeroshot_tot = format_prompt(
    "Imagine three different experts are answering thi
)
```

We can use this prompt to explore how an LLM might respond to complex questions:

```
# Run generative model
outputs = pipe(zeroshot_tot, max_new_tokens=512, do_sa
print(outputs[0]["generated_text"]))
```

As a result, it generated the correct answer by leveraging the

discussion between multiple experts:

""

Expert 1: The cafeteria started with 23 apples.

Expert 2: They used 20 of those apples to make lunch.

Expert 3: After making lunch, they bought 6 more apple

Expert 2: Now, they have a total of $(23-20)$, initial ap

Expert 1: Wait a minute... If they started with 23 app

[Expert 1 realizes they made a math mistake and leaves

Expert 2: I'm going to double-check my math... [Takes

Expert 3: I'm confident in my figures, it's clear we h

""

It is interesting to see such an elaborate conversation between “experts” and demonstrates the creativity that comes with prompt engineering.

Output Verification

Systems and applications built with generative models might eventually end up in production. When that happens, it is important that we verify and control the output of the model to prevent breaking the application and to create a robust generative AI application.

Reasons for validating the output might include:

Structured output

By default, most generative models create free-form text without adhering to specific structures other than those defined by natural language. Some use cases require their output to be structured in certain formats, like JSON.

Valid output

Even if we allow the model to generate structured output, it still has the capability to freely generate its content. For instance, when a model is asked to output either one of two choices, it should not come up with a third.

Ethics

Some open-source generative models have no guardrails and will generate outputs that do not consider safety or ethical considerations. For instance, use cases might require the output to be free of profanity, personally identifiable information (PII), bias, cultural stereotypes, etc.

Accuracy

Many use cases require the output to adhere to certain standards or performance. The aim is to double-check whether the generated information is factually accurate, coherent, or

free from hallucination.

Controlling the output of a generative model, as we explored with parameters like `top_p` and `temperature`, is not an easy feat. These models require help to generate consistent output conforming to certain guidelines.

Generally, there are three ways of controlling the output of a generative model:

Examples

Provide a number of examples of the expected output.

Grammar

Control the token selection process.

Fine-tuning

Tune a model on data that contains the expected output

In this section, we will go through the first two methods. The third, fine-tuning a model, is left for Chapter 12 where we will in-depth into fine-tuning methods.

Providing Examples

A simple and straightforward method to fix the output is to provide the generative model with examples of what the output

should look like. As we explored before, few-shot learning is a helpful technique that guides the output of the generative model. This method can be generalized to guide the structure of the output as well.

For example, let us consider an example where we want the generative model to create a character profile for an RPG game. We start by using no examples:

```
# Zero-shot learning: Providing no examples
zero_shot = format_prompt("Create a character profile
outputs = pipe(zero_shot, max_new_tokens=128, do_sampl
print(outputs[0]["generated_text"])
```

This gives us the following structure which we truncated to prevent overly long descriptions:

```
{
  "name": "Aurelia",
  "race": "Human",
  "class": "Mage",
  "age": 22,
  "gender": "Female",
  "description": "Aurelia is a young woman with a stri
  "stats": {
    "strength": 8
```

```
}  
}
```

Although this is valid JSON, we might not want certain attributes like “strength” or “age”. Instead, we can provide the model with a number of examples that indicate the expected format:

```
# Providing an example of the output structure  
one_shot_prompt = format_prompt("""Create a character  
{  
    "description": "A SHORT DESCRIPTION",  
    "name": "THE CHARACTER'S NAME",  
    "armor": "ONE PIECE OF ARMOR",  
    "weapon": "ONE OR MORE WEAPONS"  
}  
""")  
outputs = pipe(one_shot_prompt, max_new_tokens=256, do  
print(outputs[0]["generated_text"]))
```

This gives us the following which we again truncated to prevent overly long descriptions:

```
{  
    "description": "A human wizard with long, wild grey  
    "name": "Sybil Astrid",
```

```
"armor": "None",  
"weapon": [  
  "Crystal Staff",  
  "Oak Wand"  
]  
}
```

The model perfectly followed the example we gave it which allows for more consistent behavior. This also demonstrates the importance of leveraging few-shot learning to improve the structure of the output and not only its content.

An important note here is that it is still up to the model whether it will adhere to your suggested format or not. Some models are better than others at following instructions.

Grammar: Constrained Sampling

Few-shot learning has a big disadvantage: we cannot explicitly prevent certain output from being generated. Although we guide the model and give it instructions, it might still not follow it entirely.

Instead, packages have been rapidly to constrain and validate the output of generative models, like Guidance, Guardrails, and

LMQL. In part, they leverage generative models to validate their own output, as illustrated in [Figure 4-20](#). The generative models retrieve the output as new prompts and attempt to validate it based on a number of predefined guardrails.

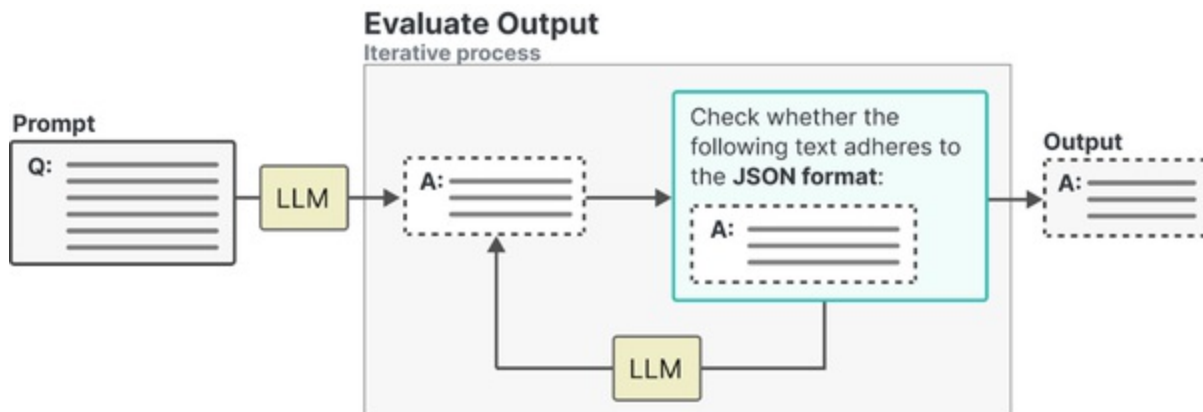


Figure 4-20. Use an LLM to check whether the output correctly follows our rules.

Similarly, as illustrated in [Figure 4-21](#), it can also be used to control the formatting of the output by generating parts of its format ourselves as we already know how it should be structured.

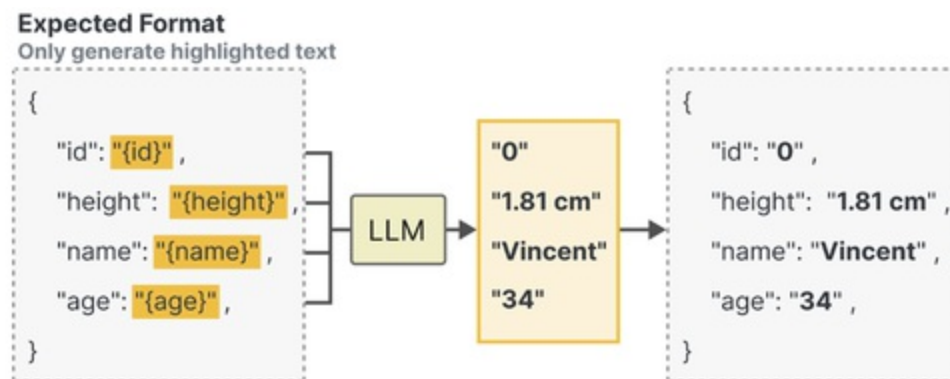


Figure 4-21. Use an LLM to generate only the pieces of information we do not know beforehand.

This process can be taken one step further and instead of validating the output we can already perform validation during the token sampling process. When sampling tokens, we can define a number of grammars or rules that the LLM should adhere to when choosing its next token. For instance, if we ask the model to either return “positive”, “negative” or “neutral” when performing sentiment classification, it might still return something else. As illustrated in [Figure 4-22](#), by constraining the sampling process, we can have the LLM only output what we are interested in.

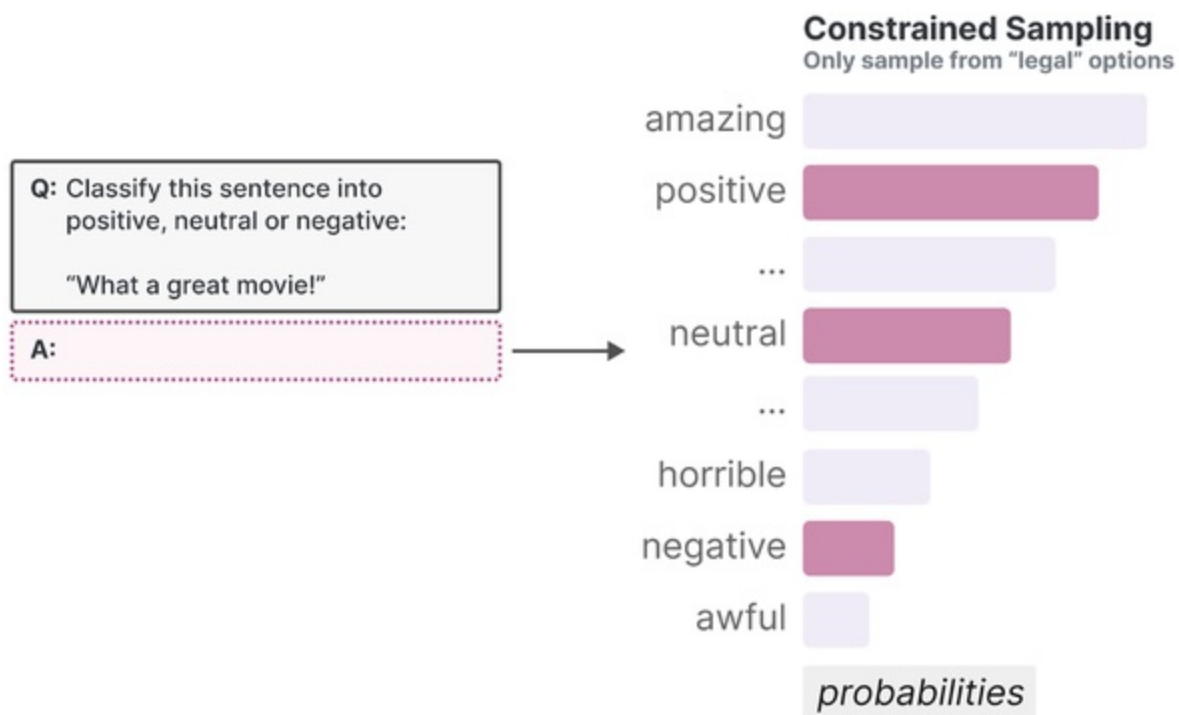


Figure 4-22. Constrain the token selection to only three possible tokens: “positive”, “neutral”, and “negative”.

Note that this is still affected by parameters such as `top_p` and

temperature and the illustrated is quite constrained.

Let us illustrate this phenomenon with llama-cpp-python, which is a library, like transformers, that we can use to load in our language model. It is generally used to efficiently load and use compressed models (through quantization; see Chapter 13).

We start by downloading the quantized version of the model by running the following in your terminal:

```
wget https://huggingface.co/TheBloke/zephyr-7B-beta-GG
```

Then, we load the model using llama-cpp-python and choose a JSON grammar to use. This will ensure that the output of the model adheres to JSON:

```
import httpx
from llama_cpp.llama import Llama, LlamaGrammar
# We load the JSON grammar from the official llama.cpp
grammar = httpx.get(
    "https://raw.githubusercontent.com/ggerganov/llama
)
grammar = LlamaGrammar.from_string(grammar.text)
# Load a pre-quantized LLM
llm = Llama("zephyr-7b-beta.Q4_K_M.gguf")
```

The rules are described in the grammar file we downloaded.

Using the JSON grammar, we can ask the model for an RPG character in JSON format to be used in our Dungeons and Dragons session:

```
import json
# Run the generative model and ask it to create a char
response = llm(
    "Create a warrior for an RPG in JSON format.",
    max_tokens=-1,
    grammar=grammar
)
# Print the output in nicely-formatted JSON
print(json.dumps(json.loads(response['choices'][0]['te
```

This gives us valid JSON:

```
[
  {
    "name": "Swordmaster",
    "level": 10,
    "health": 250,
    "mana": 100,
    "strength": 18,
```

```
        "dexterity": 16,  
        "intelligence": 10,  
        "armor": 75,  
        "weapon": "Two-Handed Sword",  
        "specialty": "One-handed Swords"  
    }  
]
```

This allows us to more confidently use generative models in applications where we expect the output to adhere to certain formats.

NOTE

Note that we set the number of tokens to be generated with `max_tokens` to be, in principle, unlimited. This means that the model will continue generating until it has completed its JSON output or until it reaches its context limit.

Summary

In this chapter, we explored the basics of using generative models through prompt engineering and output verification. We focused on the creativity and potential complexity that comes with prompt engineering. We discovered that the components of a prompt are key in generating the output that is right for our

use case. As a result, experimentation is vital when prompt engineering.

In the next chapter, we explore advanced techniques for leveraging generative models. These techniques go beyond prompt engineering and are meant to enhance the capabilities of these models. From giving a model external memory to using external tools, we aim to give a generative model superpowers!

Chapter 5. Multimodal Large Language Models

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. *In particular, some of the formatting may not match the description in the text: this will be resolved when the book is finalized.*

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

When you think about Large Language Models (LLMs), multimodality might not be the first thing that comes to mind. After all, they are *Language Models*!

We have seen all manner of emerging behaviors rising from LLMs, from generalization capabilities and reasoning to arithmetic and linguistics. As models grow larger and smarter, so do their skill sets.¹

The ability to receive and reason with multimodal input might further increase and help emerge capabilities that were previously locked. In practice, Language does not solely live in a vacuum. As an example, your body language, facial expressions, intonation, etc. are all methods of communication that enhance the spoken word.

The same thing applies to Large Language Models, if we can enable them to reason about multimodal information, their capabilities might increase.

In this chapter, we will explore a number of different LLMs that have multimodal capabilities and what that means for practical use cases. We will start by exploring how images are converted to numerical representations using an adaption of the original transformer technique. Then, we will show how LLMs can be extended to include vision tasks using this transformer.

Transformers for Vision

Throughout the chapters of this book, we have seen the success of using transformer-based models for a variety of language modeling tasks, from classification and clustering to search and generative modeling.

So it might not be surprising that researchers have been looking at a way to generalize some of the transformer's success to the field of computer vision.

The method they came up with is called the Vision Transformer (ViT) which has been shown to do tremendously well on image recognition tasks compared to the previously default Convolutional Neural Networks (CNNs).² Like the original transformer, ViT is used to transform unstructured data, an image, into representations that can be used for a variety of tasks, like classification as illustrated in [Figure 5-1](#).

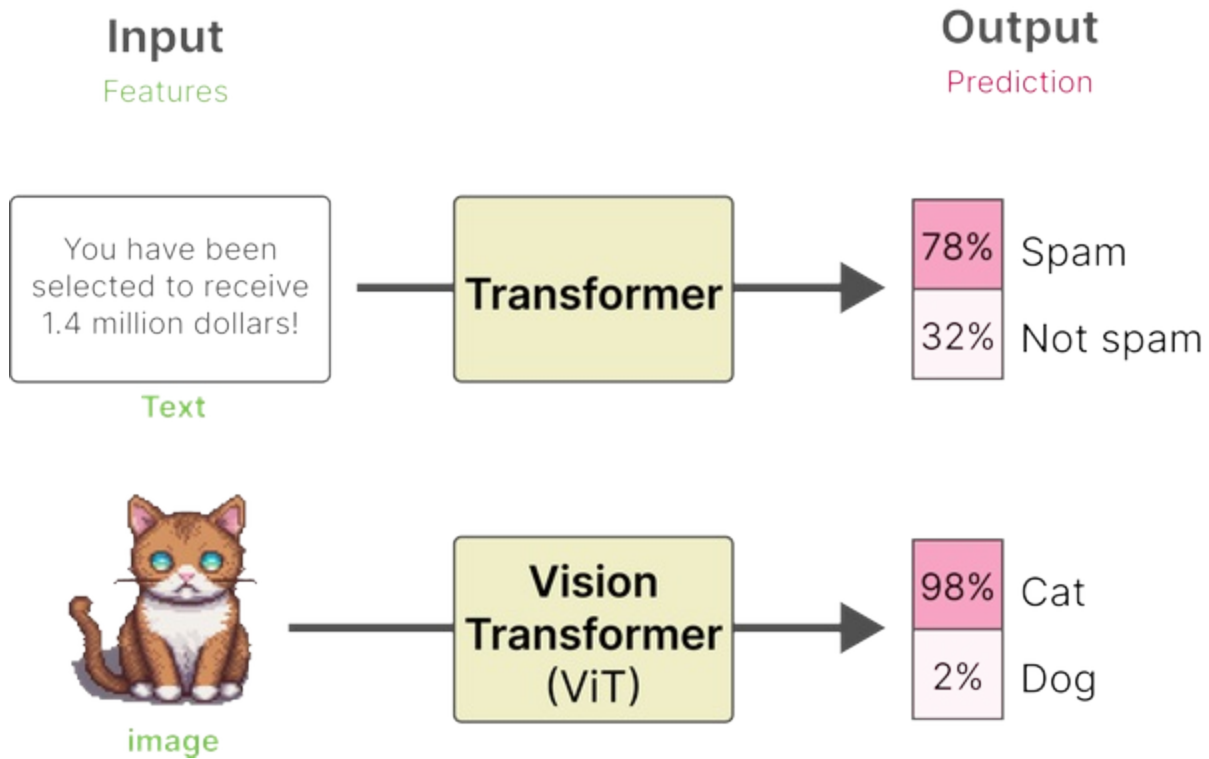


Figure 5-1. Both the original transformer as well as the vision transformer take unstructured data, convert it to numerical representations, and finally use that for tasks like classification.

ViT relies on an important component of the transformer architecture, namely the encoder. As we saw in Chapter 1, the encoder is responsible for converting textual input into numerical representations before being passed to the decoder. However, before the encoder can perform its duties, the textual input needs to be tokenized first as is illustrated in [Figure 5-2](#).

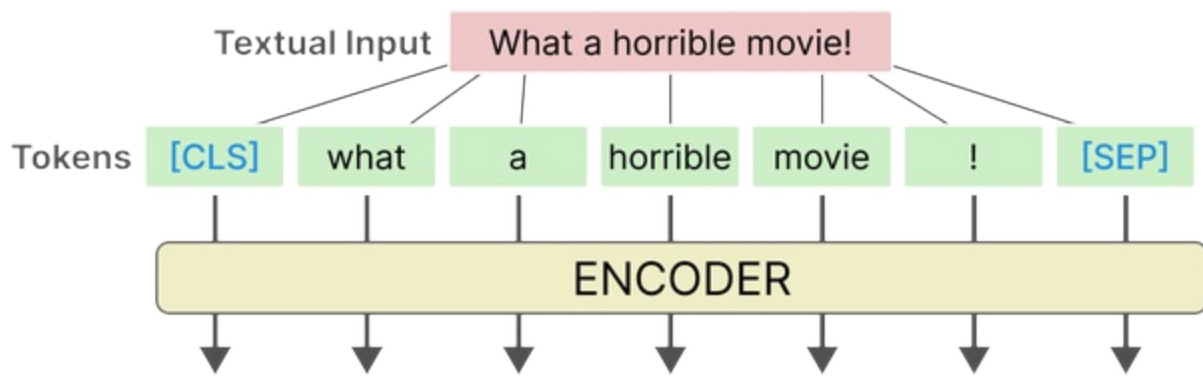


Figure 5-2. Text is passed to one or multiple encoders by first tokenizing it using a tokenizer.

Since an image does not consist of words this tokenization process cannot be used for visual data. Instead, the authors of ViT came up with a method for tokenizing images into “words” which allowed them to use the original encoder structure.

Imagine that you have an image of a cat. This image is represented by a number of pixels, let’s say 512 by 512 pixels. Each individual pixel does not convey much information but when you combine patches of pixels, you slowly start to see more information.

ViT uses a principle much like that. Instead of splitting text up into tokens, it converts the original image into patches of images. In other words, it cuts the image into a number of pieces horizontally and vertically as illustrated in [Figure 5-3](#).

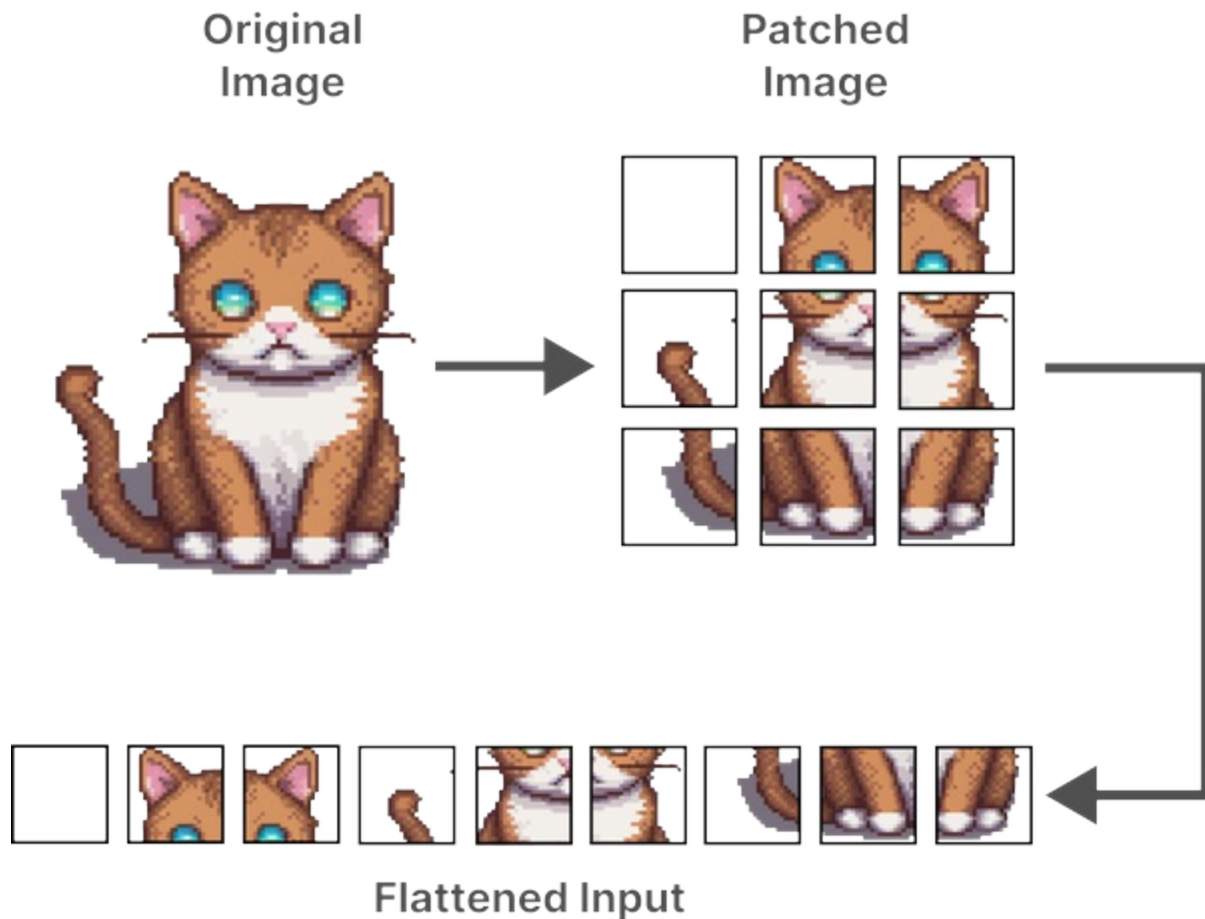


Figure 5-3. The “tokenization” process for image input. It converts an image into patches of sub-images.

Just like we are converting text into tokens of text, we are converting an image into patches of images. The flattened input of image patches can be thought of as the tokens in a piece of text.

However, unlike tokens, we cannot just assign each patch with an ID since these patches will rarely be found in other images, unlike the vocabulary of a text.

Instead, the patches are linearly embedded to create numerical representations, namely embeddings. These can then be used as the input of a transformer model. That way, the patches of images are treated the same way as tokens. The full process is illustrated in [Figure 5-4](#).

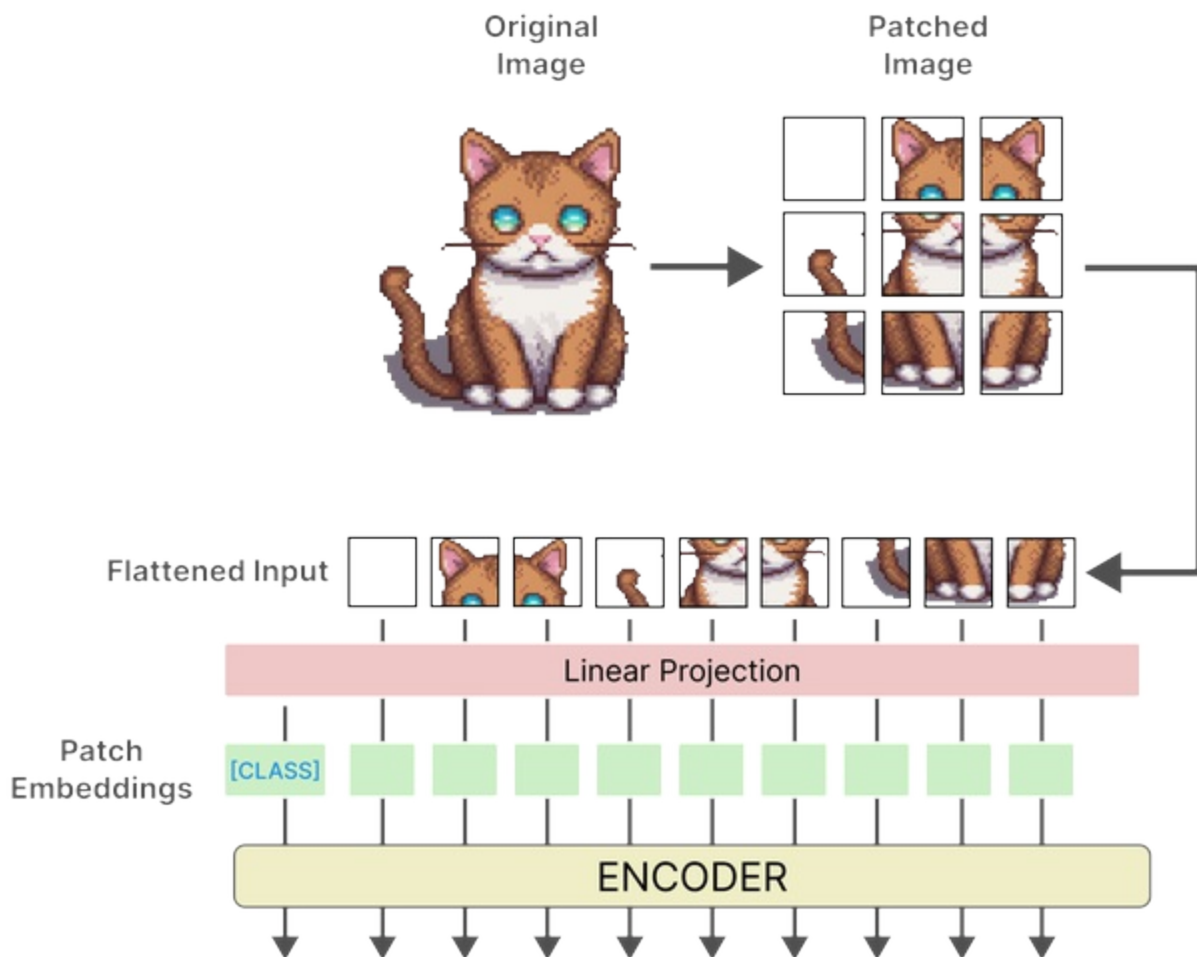


Figure 5-4. The main algorithm behind ViT. After patching the images and linearly projecting them, the patch embeddings are passed to the encoder and treated as if they were textual tokens.

For illustrative purposes, the images in the examples were patched into 3 by 3 patches but the original implementation used

16 by 16 patches. After all, the paper is called “An image is worth 16x16 words”.

What is so interesting about this approach is that the moment the embeddings are passed to the encoder, they are treated as if they were textual tokens. From that point forward, there is no difference in how a textual or image trains and their outputs

Due to their similarities, the ViT is often used to make all kinds of language models multimodal. One of the most straightforward ways to use them is during the training of embedding models.

Multimodal Embedding Models

In previous chapters, like Chapters X, X, and X, we used embedding models to capture the semantic content of textual representations, such as books and documents. We saw that we could use these embeddings or numerical representations to find similar documents, apply classification tasks, and even perform topic modeling.

As we have seen many times before, embeddings often are an important driver behind LLM applications. They are an efficient method for capturing large-scale information and searching for the needle in the haystack of information.

That said, we have only looked at monomodal embedding models thus far. Embedding models that only focus on generating embeddings for textual representations. Although embedding models exist for solely embedding imagery, we will look at embedding models that can capture both textual as well as vision representations. We illustrate this in [Figure 5-5](#).

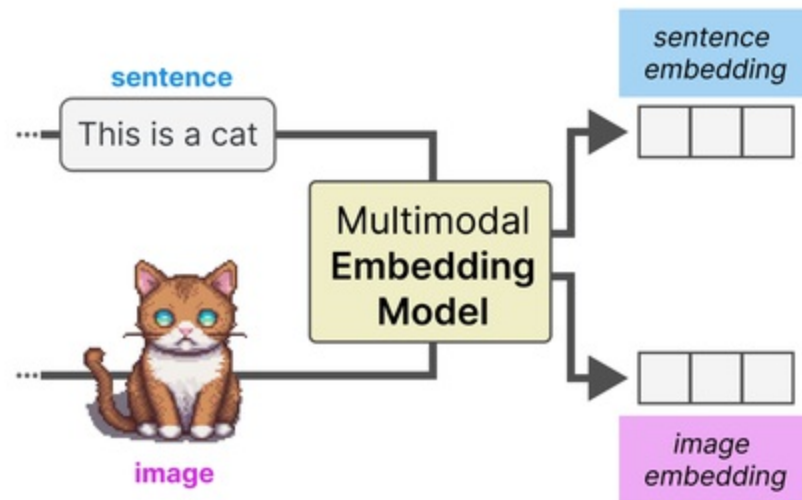


Figure 5-5. Multimodal embedding models can create embeddings for multiple modalities in the same vector space.

A big advantage is that it allows for comparing multimodal representations since the resulting embeddings lie in the same vector space, as illustrated in [Figure 5-6](#). For instance, using such a multimodal embedding model, we can find images based on input text. What images would we find if we search for images similar to “pictures of a puppy”? Vice versa would also be possible. Which documents are best related to this question?

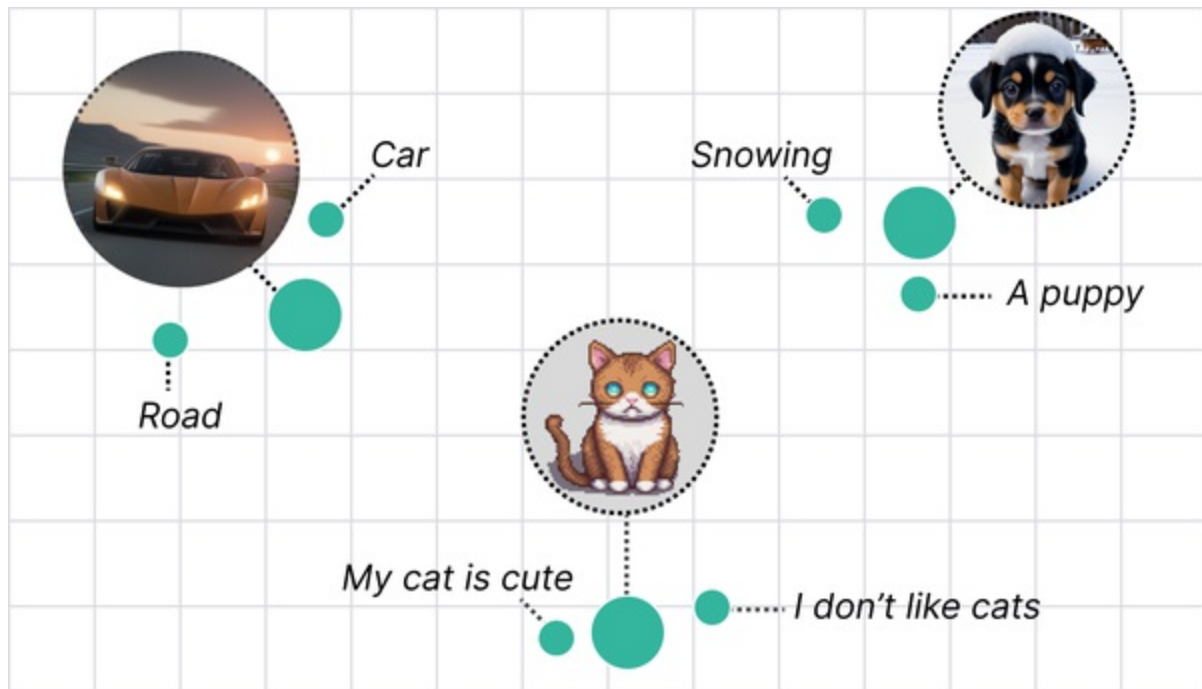


Figure 5-6. Multimodal embedding models can create embeddings for multiple modalities in the same vector space.

There are a number of multimodal embedding models out there but the most well-known and currently most-used model is CLIP (Contrastive Language-Image Pre-Training).

CLIP: Connecting Text and Images

CLIP is an embedding model that can compute embeddings of both images and texts. The resulting embeddings lie in the same vector space which means that the embeddings of images can be compared with the embeddings of text.³

This capability of comparison makes CLIP, and similar models, usable for tasks such as:

Zeroshot classification

We can compare the embedding of an image with that of the description of its possible classes to find which class is most similar

Clustering

Cluster both images and a collection of keywords to find which keywords belong to which sets of images

Search

Across billions of texts or images, we can quickly find what relates to an input text or image

Generation

Use multimodal embeddings to drive the generation of images (e.g., stable diffusion)

How can CLIP generate multimodal embeddings?

The procedure of CLIP is actually quite straightforward. Imagine that you have a dataset with millions of images alongside captions as we illustrate in [Figure 5-7](#).



Figure 5-7. The type of data that is needed to train a multimodal embedding model.

This dataset can be used to create two representations for each pair, the image and its caption. To do so, CLIP uses a text encoder to embed text and an image encoder to embed images. As is shown in [Figure 5-8](#), the result is an embedding for both the image and its corresponding caption.

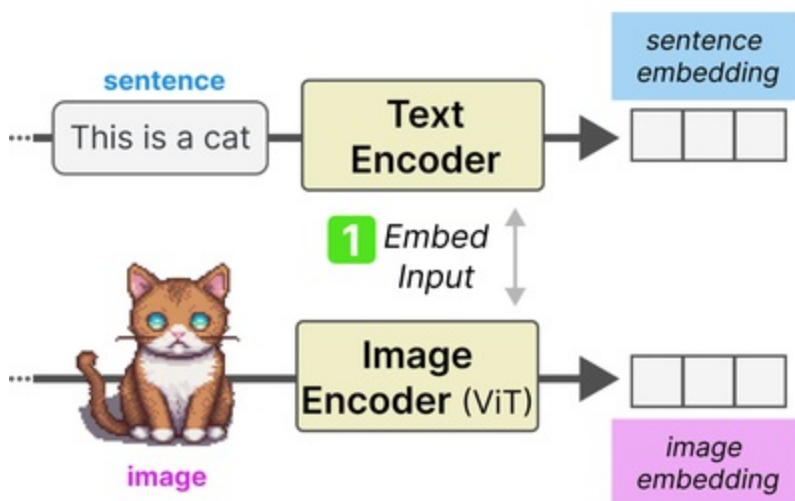


Figure 5-8. In the first step of training CLIP, both images and text are embedded using an image and text encoder respectively.

The pair of embeddings that are generated are compared through cosine similarity. As we saw in Chapter 2, cosine similarity is the cosine of the angle between vectors which is calculated through the dot product of the embeddings and divided by the product of their lengths.

When we start training, the similarity between the image embedding and text embedding will be low as they are not yet optimized to be within the same vector space. During training, we optimize for the similarity between the embeddings and want to maximize them for similar image/caption pairs and minimize them for dissimilar image/caption pairs.

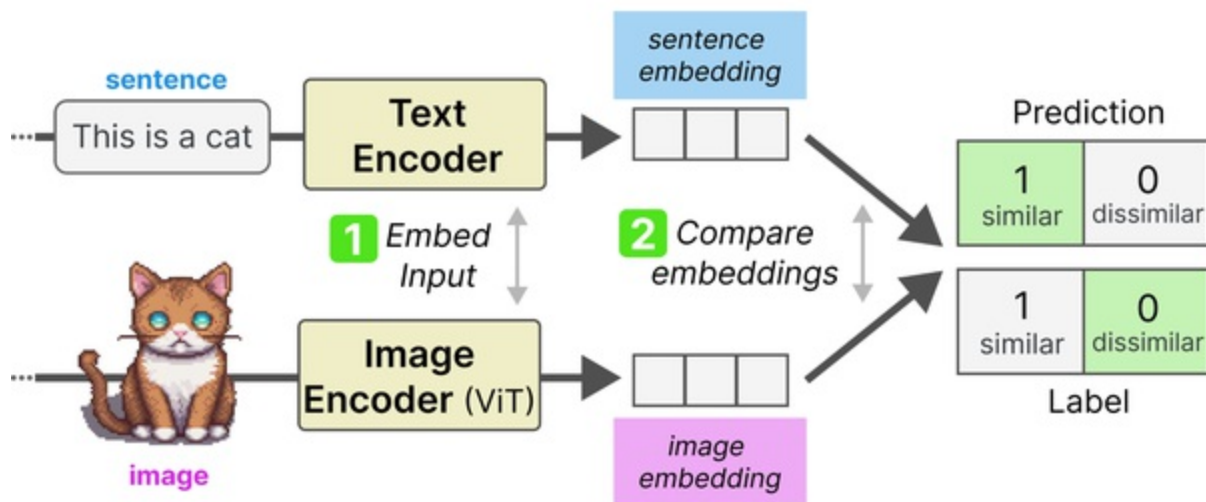


Figure 5-9. In the second step of training CLIP, the similarity between the sentence and image embedding is calculated using cosine similarity.

After calculating their similarity, the model is updated and the process starts again with new batches of data and updated representations. This method is called contrastive learning and we will go in-depth into its inner workings in Chapter 13 where we will create our own embedding model.

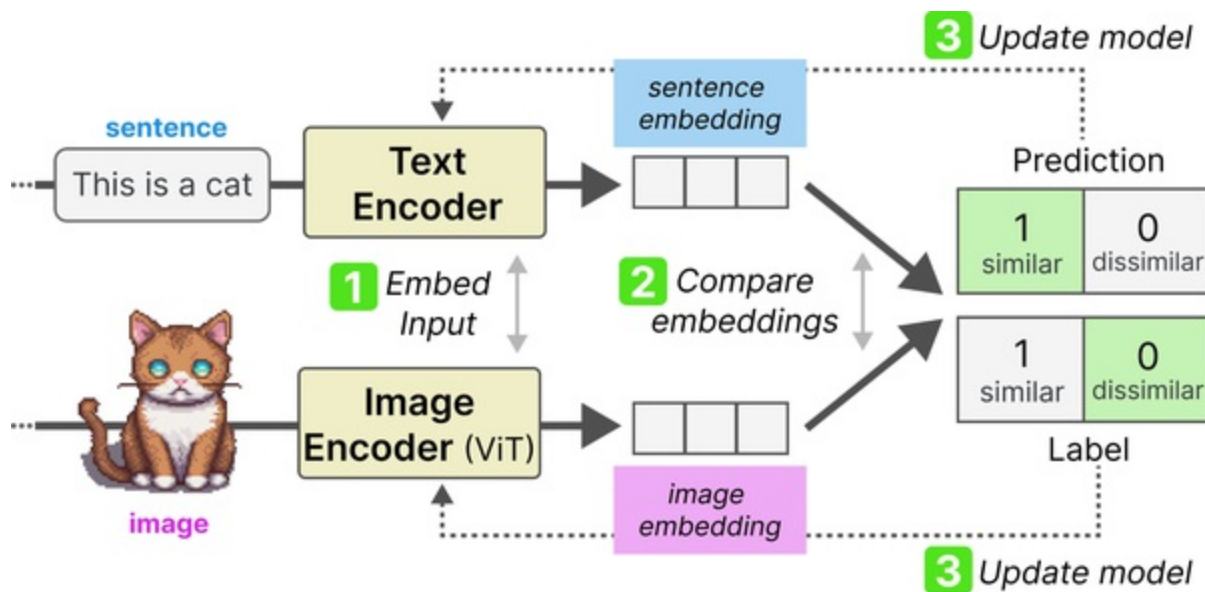


Figure 5-10. In the third step of training CLIP, the text and image encoders are updated to match what the intended similarity should be. This updates the embeddings such that they are closer in vector space if the inputs are similar.

Eventually, we expect the embedding of an image of a cat would be similar to the embedding of the sentence “a picture of a cat”. As we will see in Chapter 13, to make sure the representations are as accurate as possible, negative examples of images and captions that are not related should also be included in the training process.

Modeling similarity is not only knowing what makes things similar to one another but also what makes them different and dissimilar.

OpenCLIP

For this example, we are going to be using models from the open-source variant of CLIP, namely OpenCLIP (https://github.com/mlfoundations/open_clip).

Using OpenCLIP, or any CLIP model, boils down to two things, processing the textual and image inputs before passing them to the main model.

Before doing so, let's take a look at a small example where we will be using one of the images we have seen before. Namely, an AI-generated image (though stable-diffusion) of a puppy playing in the snow as illustrated in [Figure 5-11](#):

```
from urllib.request import urlopen
from PIL import Image

# Load an AI-generated image of a puppy playing in the
image = Image.open(urlopen("https://i.imgur.com/iQ50tW
caption = "a puppy playing in the snow"
```



Figure 5-11. An AI-generated image of a puppy playing in the snow.

Since we have a caption for this image, we can use OpenCLIP to generate embeddings for both.

To do so, we load in three models:

A tokenizer for tokenizing the textual input

A preprocessor to preprocess and resize the image

The main model that converts the previous outputs to embeddings

```
from transformers import CLIPTokenizerFast, CLIPProcessor

model_id = "openai/clip-vit-base-patch32"

# Load a tokenizer to preprocess the text
tokenizer = CLIPTokenizerFast.from_pretrained(model_id)

# Load a processor to preprocess the images
processor = CLIPProcessor.from_pretrained(model_id)

# Main model for generating text and image embeddings
model = CLIPModel.from_pretrained(model_id)
```

After having loaded in the models, preprocessing our input is straightforward. Let's start with the tokenizer and see what happens if we preprocess our input:

```
>>> # Tokenize our input
>>> inputs = tokenizer(caption, return_tensors="pt");

{'input_ids': tensor([[49406,   320,  6829, 1629,   5
```

Our input text has been converted to input ids. To see what those represent, let's convert them to tokens:

```
>>> tokenizer.convert_ids_to_tokens(inputs["input_ids"]

['<|startoftext|>',
 'a</w>',
 'puppy</w>',
 'playing</w>',
 'in</w>',
 'the</w>',
 'snow</w>',
 '<|endoftext|>']
```

As we often have seen before, the text is split up into tokens. Additionally, we now also see that the start and end of the text is indicated to separate it from a potential image embedding. You might also notice that the [CLS] token is missing. In CLIP, the [CLS] token is actually used to represent the image embedding.

Now that we have preprocessed our caption, next up is to create the embedding:

```
>>> # Create a text embedding
>>> text_embedding = model.get_text_features(**inputs)
>>> text_embedding.shape
```

```
torch.Size([1, 512])
```

Before we can create our image embedding, like the text embedding, we will need to preprocess it as the model expects the input image to have certain characteristics, like its size and shape.

To do so, we can use the processor that we created before:

```
>>> # Preprocess image
>>> processed_image = processor(text=None, images=image)
>>> processed_image.shape

torch.Size([1, 3, 224, 224])
```

The original image was 512 by 512 pixels. Notice that the preprocessing of this image reduced its size to 224 by 224 pixels as that is its expected size.

Let's visualize, in [Figure 5-12](#), the preprocessed image to see what it actually is doing:

```
import numpy as np
```

```
# Prepare image for visualization
img = processed_image.squeeze(0).T
img = np.einsum('ijk->jik', img)

# Visualize preprocessed image
plt.imshow(a)
plt.axis('off')
```



Figure 5-12. The preprocessed input image by CLIP.

To convert this preprocessed image into embeddings, we can call the `model` as we did before:

```
>>> # Create the image embedding
```

```
>>> image_embedding = model.get_image_features(process
>>> image_embedding.shape

torch.Size([1, 512])
```

Notice that the shape of the resulting image embedding is exactly the same as that of the text embedding. This is important as it allows us to compare their embeddings and see whether they actually are similar.

We can use these embeddings to calculate the probability that the caption belongs to the image by calculating their dot product and taking the softmax:

```
>>> # Calculate the probability of the text belonging
>>> text_probs = (100.0 * image_embedding @ text_embed
>>> text_probs

tensor([[1.]], grad_fn=<SoftmaxBackward0>)
```

It gives us back a score of 1 indicating that the model is certain that the caption belongs to the image.

We can extend this example by calculating the similarity between the embeddings. By normalizing the embeddings first

before calculating the dot product, we get a value that lies between 0 and 1:

```
>>> # Normalize the embeddings
>>> text_embedding /= text_embedding.norm(dim=-1, keep
>>> image_embedding /= image_embedding.norm(dim=-1, ke
>>>
>>> # Calculate their similarity
>>> text_embedding = text_embedding.detach().cpu().num
>>> image_embedding = image_embedding.detach().cpu().n
>>> score = np.dot(text_embedding, image_embedding.T)
>>> score

array([[0.33149636]], dtype=float32)
```

We get a similarity score of 0.33 which is difficult to interpret considering we do not know what the model considers a low versus a high similarity score.

Instead, let's extend the example with more images and captions as illustrated in [Figure 5-13](#).

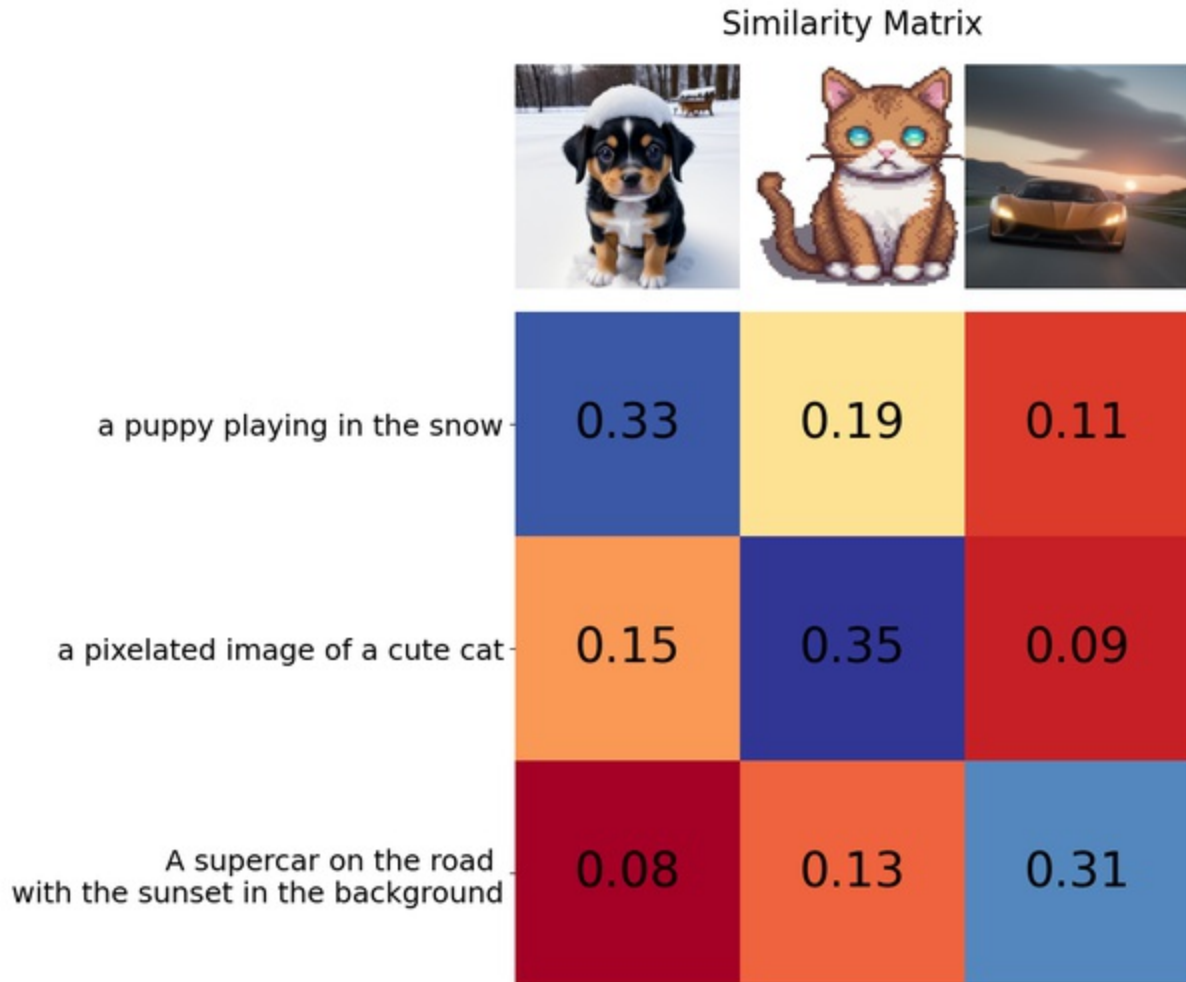


Figure 5-13. The similarity matrix between three images and three captions.

It seems that a score of 0.33 is indeed high considering the similarities with other images are quite a bit lower.

TIP

In sentence-transformers, there are a few CLIP-based models implemented that make it much easier to create embeddings. It only takes a few lines of code:

```
from sentence_transformers import SentenceTransformer, util

# Load SBERT-compatible CLIP model
```

```
model = SentenceTransformer('clip-ViT-B-32')

# Encode the images
image_embeddings = model.encode(images)

# Encode the captions
text_embeddings = model.encode(captions)

#Compute cosine similarities
sim_matrix = util.cos_sim(image_embeddings, text_embeddings)
print(sim_matrix)
```

Making Text Generation Models Multimodal

Traditionally, text generation models have been, as you might expect, models that interpret textual representations. Models like Llama 2 and ChatGPT excel at reasoning about textual information and responding with natural language.

They are, however, limited to the modality they were trained in, namely text. As we have seen before with multimodal embedding models, the addition of vision can enhance the capabilities of a model.

In the case of text generation models, we would like it to reason about certain input images. For example, we could give it an

image of a pizza and ask it what ingredients it contains. You could show it a picture of the Eiffel Tower and ask it when it was built or where it is located. This conversational ability is further illustrated in [Figure 5-14](#).

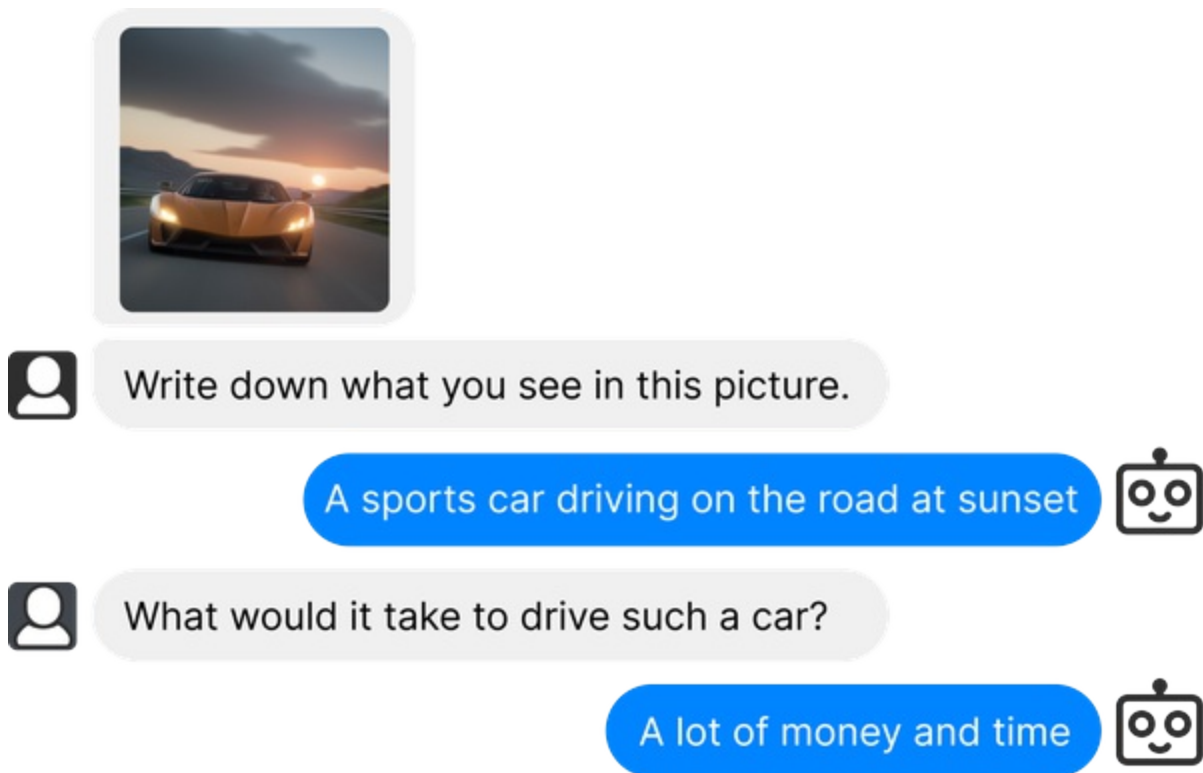


Figure 5-14. A multimodal text generation model that can reason about input images.

To bridge the gap between these two domains, attempts have been made to introduce a form of multimodality to existing models. One such method is called BLIP-2: *Bootstrapping Language Image Pre-training for unified vision-language understanding and generation 2*. BLIP-2 introduces an easy-to-use and modular technique that allows for introducing vision

capabilities to existing language models.

BLIP-2: Bridging the Modality Gap

Creating a multimodal language model from scratch requires significant computing power and data. We would have to use billions of images, text, and image-text pairs to create such a model. As you can imagine, this is not easily feasible!

Instead of building the architecture from scratch, BLIP-2 bridges the vision-language gap by building a bridge, named the Q-former, that connects a pre-trained image encoder and a pre-trained LLM.

By leveraging pre-trained models, BLIP-2 only needs to train the bridge without needing to train the image encoder and LLM from scratch. It makes great use of the technology and models that are already out there! This bridge is illustrated in [Figure 5-15](#).

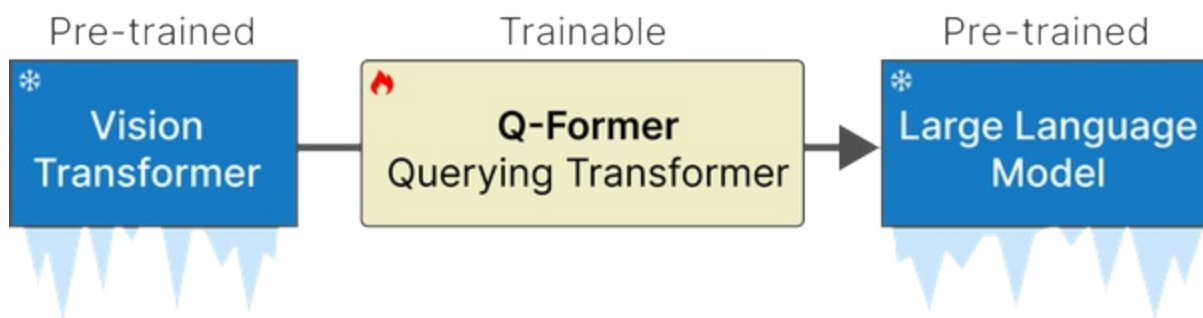


Figure 5-15. The Querying Transformer is the bridge between vision (ViT) and text (LLM) which is the only trainable component of the pipeline.

To connect the two pre-trained models, the Q-Former, also known as the Querying Transformer, mimics their architectures. It has two modules that share their attention layers:

- An image transformer to interact with the frozen vision transformer for feature extraction
- A text transformer that can interact with the LLM

The Q-Former is trained in two stages, one for each modality as illustrated in [Figure 5-16](#).

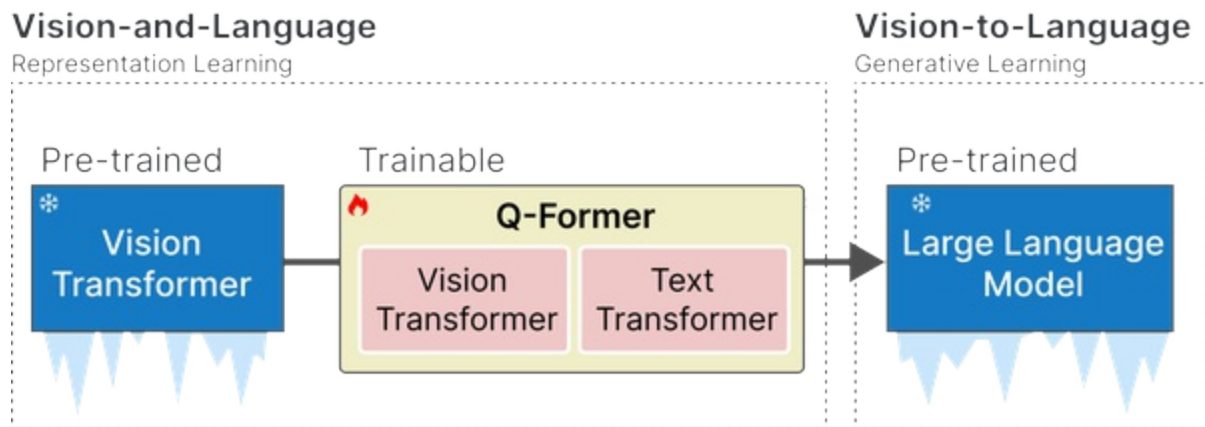


Figure 5-16. In step 1, representation learning is applied to learn representations for vision and language simultaneously. In step 2, these representations are converted to soft visual prompts to feed the LLM.

In step 1, a number of image-document pairs are used to train the Q-Former to represent both images and text. These pairs are generally captions of images, as we have seen before with training CLIP.

The images are fed to the frozen vision transformer to extract vision embeddings. These embeddings are used as the input of Q-Former's vision transformer. The captions are used as the input of Q-Former's text transformer.

With these inputs, the Q-Former is then trained on three tasks:

1. Image-Text Contrastive Learning
2. Image-Text Matching
3. Image-grounded Text Generation

These three objectives are jointly optimized to improve the visual representations that are extracted from the frozen vision transformer. In a way, we are trying to inject textual information into the embeddings of the frozen vision transformer so that we can use them in the LLM. This first step of BLIP-2 is illustrated in [Figure 5-17](#).

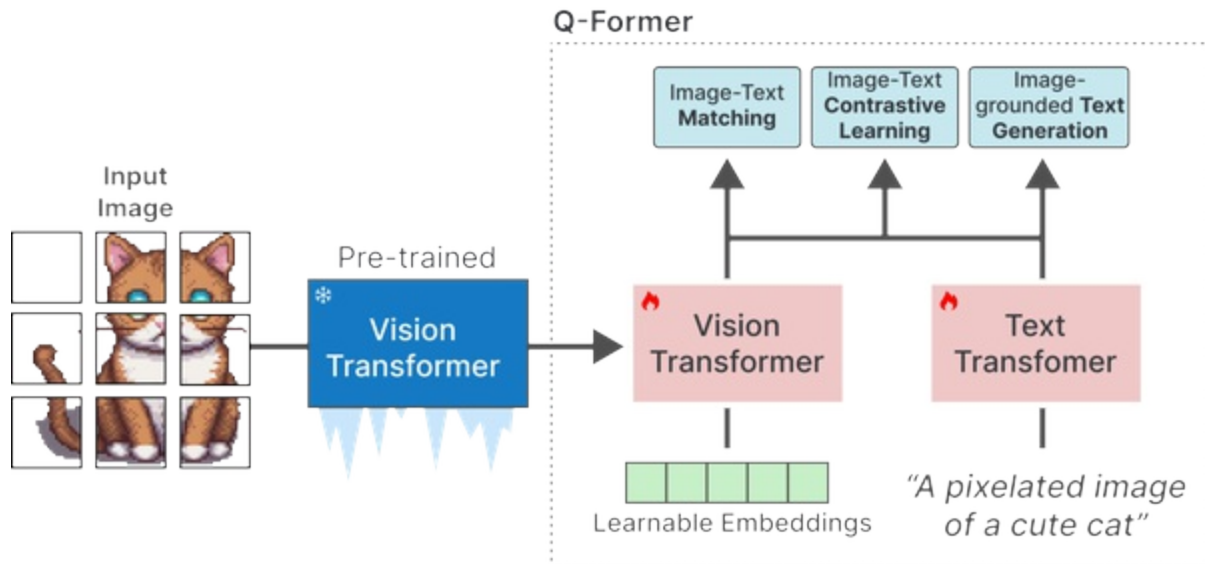


Figure 5-17. In step 1, the output of the frozen vision transformer is used together with its caption and trained on three contrastive-like tasks to learn visual-text representations.

In step 2, the learnable embeddings derived from step 1 now contain visual information in the same dimensional space as its corresponding textual information.

The learnable embeddings are then passed to the LLM as a soft prompt. In a way, these embeddings contain textual representations of the input image.

The learnable embeddings are then passed to the LLM. In a way, these embeddings serve as soft visual prompts that condition the LLM on the visual representations that were extracted by the Q-Former.

There is also a fully connected linear layer in between them to

make sure that the learnable embeddings have the same shape as the LLM expects. This second step of converting vision to language is represented in [Figure 5-18](#).

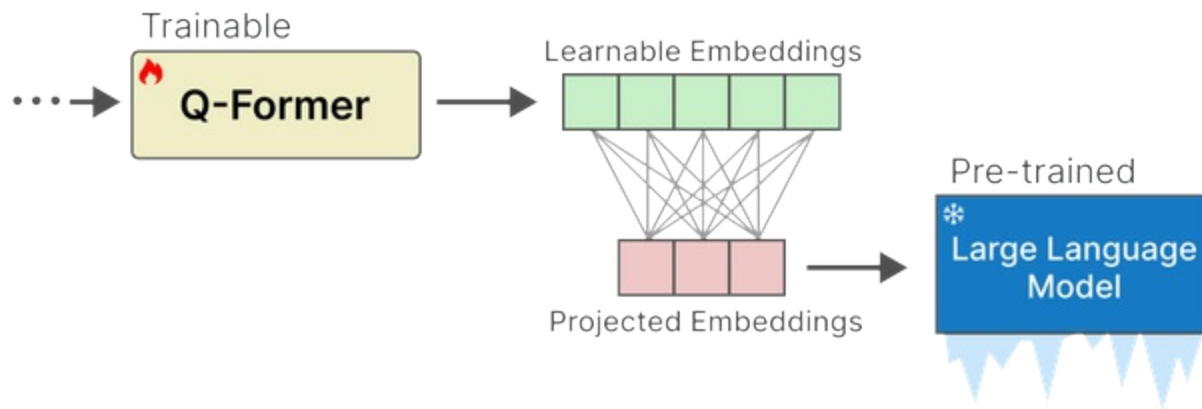


Figure 5-18. In step 2, the learned embeddings from the Q-Former are passed to the LLM through a projection layer. The projected embeddings serve as a soft visual prompt.

When we put these steps together, they make it possible for the Q-Former to learn visual and textual representations in the same dimensional space which can be used as a soft prompt to the LLM. As a result, the LLM will be given information about the image and is similar to the context you would provide an LLM when prompting. The full in-depth process is illustrated in [Figure 5-19](#).

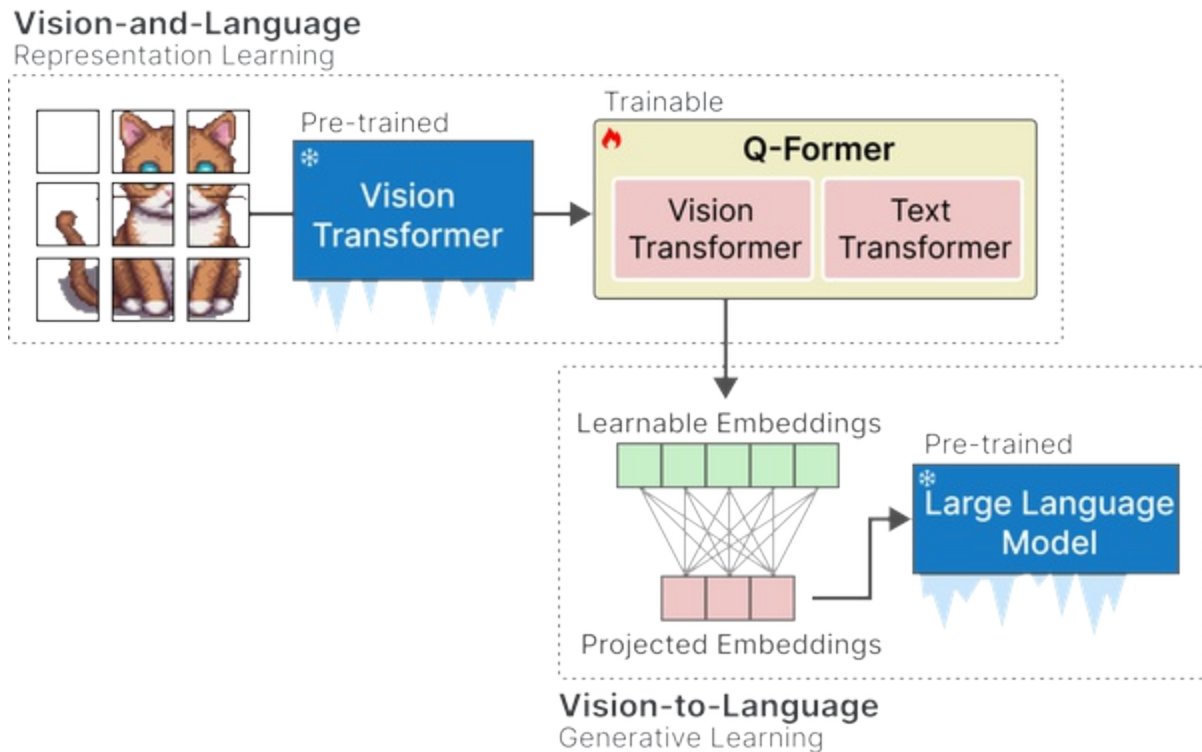


Figure 5-19. The full procedure of BLIP-2.

Preprocessing Multimodal Inputs

Now that we know how BLIP-2 is created, there are a number of interesting use cases for which you can use such a model. Not limited to captioning images, answering visual questions, and even performing prompting.

Before we go through some use cases, let's first load the model and explore how you can use it:

```
from transformers import AutoProcessor, Blip2ForCond
import torch
```

```
# Load processor and main model
processor = AutoProcessor.from_pretrained("Salesforce/
model = Blip2ForConditionalGeneration.from_pretrained(

# Send the model to GPU to speed up inference
device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)
```

NOTE

Using `model.vision_model` and `model.language_model` we can see which vision transformer and large language model are respectively used in the BLIP-2 model that we loaded.

We loaded two components that make up our full pipeline, a `processor` and a `model`. The `processor` can be compared to the tokenizer of language models. It converts unstructured input, such as images and text, to representations that the model generally expects.

Preprocessing Images

Let's start by exploring what the `processor` does to images. We start by loading the picture of a very wide image for illustration purposes:

```
from urllib.request import urlopen
from PIL import Image

# Load a wide image
link = "https://images.unsplash.com/photo-152460201076
image = Image.open(urlopen(link)).convert("RGB")
image
```



Figure 5-20. Caption to come

The image has 520 by 492 pixels which is generally an unusual format. So let's see what our processor does to it.

```
>>> np.array(image).shape  
  
(520, 492, 3)
```

When we check its shape after converting it to Numpy, it shows us an additional dimension that is of size 3. This represents the RGB coding of each pixel, namely its color.

Next, we pass the original image to the processor so that the image can be processed to the shape the model expects:

```
>>> inputs = processor(image, return_tensors="pt").to(  
>>> inputs["pixel_values"].shape  
  
torch.Size([1, 3, 224, 224])
```

The result is a 224 by 224 sized image. Quite a bit smaller than we initially had! This also means that all different shapes of images will be processed into squares. So be careful inputting very wide or tall images as they might get distorted.

Preprocessing Text

Let's continue this exploration of the `processor` with text instead. First, we can access the tokenizer used to tokenize the input text:

```
>>> processor.tokenizer  
  
GPT2TokenizerFast(name_or_path='Salesforce/blip2-opt-2
```

The BLIP-2 model that we are using uses a GPT2Tokenizer. Most tokenizers work very similarly but have slight differences in when and how they tokenize the input text.

To explore how this GPT2Tokenizer works, we can try it out with a small sentence. We start by converting the sentence to token ids before converting them back to tokens:

```
# Preprocess the text  
text = "Her vocalization was remarkably melodic"  
token_ids = processor(image, text=text, return_tensors  
  
# Convert input ids back to tokens  
tokens = processor.tokenizer.convert_ids_to_tokens(tok
```

When we inspect the tokens, you might notice a strange symbol

at the beginning of some tokens. Namely, the Ġ symbol. This is actually supposed to be a space. However, an internal function takes characters in certain code points and moves them up by 256 to make them printable. As a result, the space (code point 32) becomes Ġ (code point 288).

We will convert them to underscores for illustrative purposes:

```
>>> tokens = [token.replace("Ġ", "_") for token in tokens]
>>> tokens

['</s>', 'Her', '_vocal', 'ization', '_was', '_remarka
```

The output shows that the underscore indicates the beginning of a word. That way, words that are made up of multiple tokens can be recognized.

Use Case 1: Image Captioning

The most straightforward usage of a model like BLIP-2 is to create captions of images that you have in your data. You might be a store that wants to create descriptions of its clothing or perhaps you are a photographer that does not have the time to manually label its 1000+ pictures of a wedding.

The process of captioning an image closely follows the processing. An image is converted to pixel values that the model can read. These pixel values are passed to BLIP-2 to be converted into soft visual prompts that the LLM can use to decide on a proper caption.

Let's take the image of a supercar and process it using the processor to derive pixels in the expected shape:

```
from urllib.request import urlopen
from PIL import Image

# Load an AI-generated image of a supercar
image = Image.open(urlopen("https://i.imgur.com/zehSvA

# Convert an image into inputs and preprocess it
inputs = processor(image, return_tensors="pt").to(devi
image
```

The next step is converting the image into token IDs using the BLIP-2 model. After doing so, we can convert the IDs into text which is the generated caption:

```
# Generate token ids using the full BLIP-2 model
generated_ids = model.generate(**inputs, max_new_token
```

```
# Convert the token ids to text
generated_text = processor.batch_decode(generated_ids,
```

When we print out the `generated_text`, we can take a look at the caption:

```
>>> print(generated_text)

an orange supercar driving on the road at sunset
```

“An orange supercar driving on the road at sunset” seems like a perfect description for this image!

Image captioning is a great way to get to learn this model before stepping into more complex use cases. Try it out with a few images yourself and see where it performs well and where it performs poorly.

Domain specific images, like pictures of specific cartoon characters or imaginary creations may fail as the model was trained on largely public data.

Let’s end this use case with a fun example, namely an image of the Rorschach which is illustrated in [Figure 5-21](#). This test is an

old psychological test which tests the individual's perception of inkblots.⁴ What someone sees in such an inkblot supposedly tells you something about a person's personality characteristics.

It is quite a subjective test but that just makes it more fun!



Figure 5-21. An image from the Rorschach test. What do you see in it?

Let's take the image illustrated in Figure 7-X and use that as our input:

```
# Load rorschach image
url = "https://upload.wikimedia.org/wikipedia/commons/
image = Image.open(urlopen(url)).convert("RGB")
```

```
# Generate caption
inputs = processor(image, return_tensors="pt").to(device)
generated_ids = model.generate(**inputs, max_new_tokens=100)
generated_text = processor.batch_decode(generated_ids, skip_special_tokens=True)
```

As before, when we print out the `generated_text`, we can take a look at the caption:

```
>>> print(generated_text)

"a black and white ink drawing of a bat"
```

“A black and white ink drawing of a bat”. I can definitely see how the model would caption this image using such a description. Since this is a Rorschach test, what do you think it says about the model?

Use Case 2: Multimodal Chat-based Prompting

Although captioning is an important task, we can extend its use case even further. In that example, we showed going from one modality, vision (image), to another, text (caption).

Instead of following this linear structure, we can try to present both modalities simultaneously by performing what is called visual question answering. In this particular use case, we give the model an image along with a question about that specific image for it to answer. The model would need to process both the image as well as the question as once.

To demonstrate, let's start with the picture of a car and ask BLIP-2 to describe the image. To do so, we first need to preprocess the image as we did a few times before:

```
# Load an AI-generated image of a supercar and process
image = Image.open(urlopen("https://i.imgur.com/zehSvA
inputs = processor(image, return_tensors="pt").to(devi
```

To perform our visual question answering we need to give BLIP-2 more than just the image, namely the prompt. Without it the model would generate a caption as it did before.

We will ask the model to describe the image we just processed:

```
# Visual Question Answering
prompt = "Question: Write down what you see in this pi

# Process both the image and the prompt
```

```
inputs = processor(image, text=prompt, return_tensors=

# Generate text
generated_ids = model.generate(**inputs, max_new_token
generated_text = processor.batch_decode(generated_ids,
```

When we print out the `generated_text`, we can explore the answer it has given to the question we asked it:

```
>>> print(generated_text)

A sports car driving on the road at sunset
```

It correctly describes the image. However, this is a rather simple example since our question is essentially asking the model to create a caption. Instead, we can ask it follow-up questions in a chat-based manner.

To do so, we can give the model our previous conversation, including its answer to our question. We then ask it a follow-up question.

```
>>> # Chat-like prompting
>>> prompt = "Question: Write down what you see in thi
>>>
```

```
>>> # Generate output
>>> inputs = processor(image, text=prompt, return_tensors='pt')
>>> generated_ids = model.generate(**inputs, max_new_tokens=100)
>>> generated_text = processor.batch_decode(generated_ids, skip_special_tokens=True)
>>> print(generated_text)
```

\$1,000,000

\$1,000,000 is highly specific! This shows a more chat-like behavior from BLIP-2 which allows for some interesting conversations.

Finally, we can make this process a bit smoother by creating an interactive chat-bot using ipywidgets, an extension for Jupyter Notebooks that allows us to make interactive buttons, input text, etc.

```
from IPython.display import HTML, display
import ipywidgets as widgets
```

```
def text_eventhandler(*args):
    question = args[0]["new"]
    if question:
        args[0]["owner"].value = ""

    # Create prompt
```

```

if not memory:
    prompt = " Question: " + question + " Answer:"
else:
    template = "Question: {} Answer: {}."
    prompt = " ".join([template.format(memory[i][0],

# Generate text
inputs = processor(image, text=prompt, return_tens
generated_ids = model.generate(**inputs, max_new_t
generated_text = processor.batch_decode(generated_

# Update memory
memory.append((question, generated_text))

# Assign to output
output.append_display_data(HTML("<b>USER:</b> " +
output.append_display_data(HTML("<b>BLIP-2:</b> "
output.append_display_data(HTML("<br>"))

# Prepare widgets
in_text = widgets.Text()
in_text.continuous_update = False
in_text.observe(text_eventhandler, "value")
output = widgets.Output()
memory = []

# Display chat box
display(

```

```
widgets.VBox(  
    children=[output, in_text],  
    layout=widgets.Layout(display="inline-flex", f  
)  
)
```

USER: Write down what you see in this picture.

BLIP-2: A sports car driving on the road at sunset

USER: What would it cost me to drive that car?

BLIP-2: \$1,000,000

USER: Why that much money?

BLIP-2: Because it's a sports car.

USER: Why are sports cars expensive?

BLIP-2: Because they're fast.

Figure 5-22. Figure Caption to come

It seems that we can continue the conversation and ask it a bunch of questions. Using this chat-based approach, we essentially created a chatbot that can reason about images!

Summary

In this chapter, we explored two methods making language

models multimodal.

1 Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., & others (2022). Emergent abilities of large language models. arXiv preprint arXiv:2206.07682.

2 Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., & others (2020). An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.

3 Radford, A., Kim, J., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., & others (2021). Learning transferable visual models from natural language supervision. In *International conference on machine learning* (pp. 8748–8763).

4 Schafer, R. (1954). Psychoanalytic interpretation in Rorschach testing: theory and application.

Chapter 6. Tokens & Token Embeddings

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. *In particular, some of the formatting may not match the description in the text: this will be resolved when the book is finalized.*

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Embeddings are a central concept to using large language models (LLMs), as you’ve seen over and over in part one of the book. They also are central to understanding how LLMs work, how they’re built, and where they’ll go in the future.

The majority of the embeddings we've looked at so far are *text embeddings*, vectors that represent an entire sentence, passage, or document. [Figure 6-1](#) shows this distinction.



Figure 6-1. The difference between text embeddings (one vector for a sentence or paragraph) and token embeddings (one vector per word or token).

In this chapter, we begin to discuss token embeddings in more detail. Chapter 2 discussed tasks of token classification like Named Entity Recognition. In this chapter, we look more closely at what tokens are and the tokenization methods used to power LLMs. We will then go beyond the world of text and see how these concepts of token embeddings empower LLMs that can understand images and data modes (other than text, for example video, audio...etc). LLMs that can process modes of data in addition to text are called *multi-modal* models. We will then delve into the famous word2vec embedding method that preceded modern-day LLMs and see how it's extending the concept of token embeddings to build commercial recommendation systems that power a lot of the apps you use.

LLM Tokenization

How tokenizers prepare the inputs to the language model

Viewed from the outside, generative LLMs take an input prompt and generate a response, as we can see in [Figure 6-2](#).

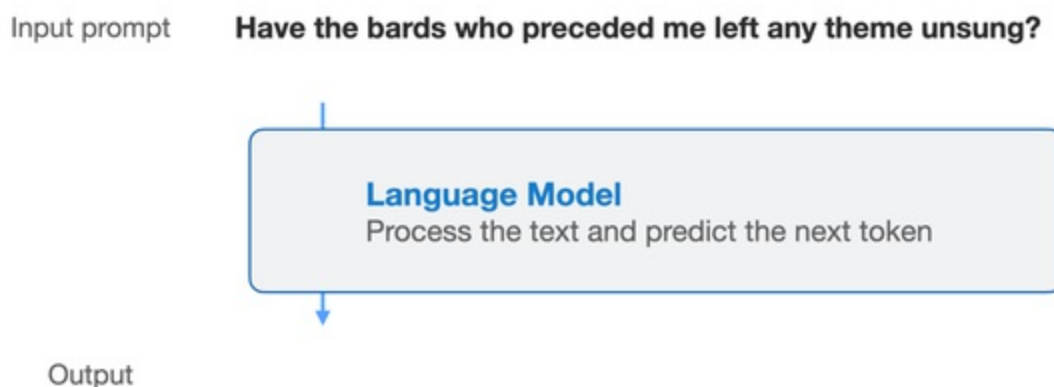


Figure 6-2. High-level view of a language model and its input prompt.

As we've seen in Chapter 5, instruction-tuned LLMs produce better responses to prompts formulated as instructions or questions. At the most basic level of the code, let's assume we have a `generate` method that hits a language model and generates text:

```
prompt = "Write an email apologizing to Sarah for the
```

```
# Placeholder definition. The next code blocks show th
def generate(prompt, number_of_tokens):
    # TODO: pass prompt to language model, and return th
    pass
output = generate(prompt, 10)
print(output)
```

Generation:

```
Subject: Apology and Condolences
Dear Sarah,
I am deeply sorry for the tragic gardening accident th
```

Let us look closer into that generation process to examine more of the steps involved in text generation. Let's start by loading our model and its tokenizer.

```
from transformers import AutoModelForCausalLM, AutoT
# openchat is a 13B LLM
model_name = "openchat/openchat"
# If your environment does not have the required ,
# then try a smaller model like "gpt2" or "openlm-
# Load a tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name)
# Load a language model
```

```
model = AutoModelForCausalLM.from_pretrained(model_name)
```

We can then proceed to the actual generation. Notice that the generation code always includes a tokenization step prior to the generation step.

```
prompt = "Write an email apologizing to Sarah for the"
# Tokenize the input prompt
input_ids = tokenizer(prompt, return_tensors="pt").input_ids
# Generate the text
generation_output = model.generate(
    input_ids=input_ids,
    max_new_tokens=256
)
# Print the output
print(tokenizer.decode(generation_output[0]))
```

Looking at this code, we can see that the model does not in fact receive the text prompt. Instead, the tokenizers processed the input prompt, and returned the information the model needed in the variable `input_ids`, which the model used as its input.

Let's print `input_ids` to see what it holds inside:

```
tensor([[ 1, 14350, 385, 4876, 27746, 5281, 304, 19235
```

This reveals the inputs that LLMs respond to. A series of integers as shown in [Figure 6-3](#). Each one is the unique ID for a specific token (character, word or part of word). These IDs reference a table inside the tokenizer containing all the tokens it knows.

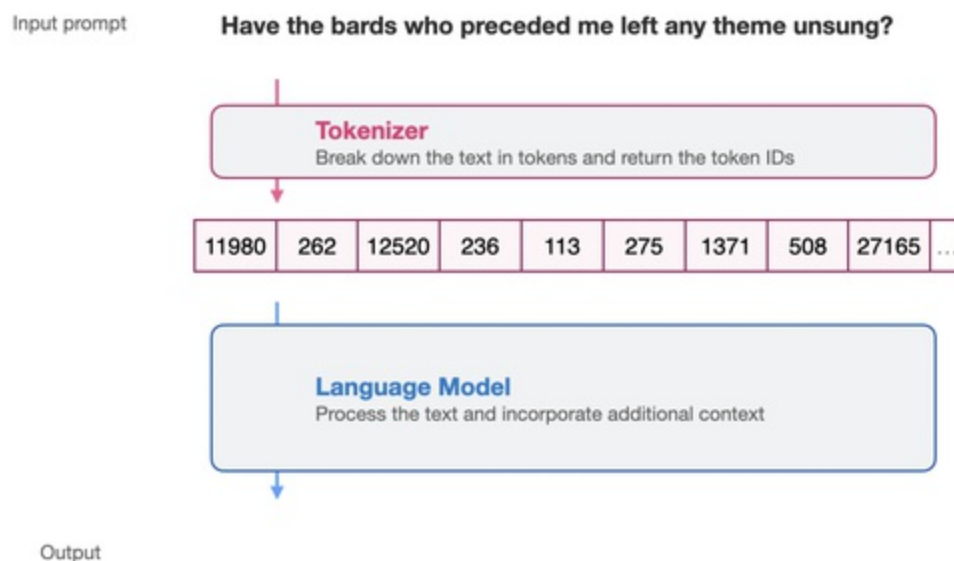


Figure 6-3. A tokenizer processes the input prompt and prepares the actual input into the language model: a list of token ids.

If we want to inspect those IDs, we can use the tokenizer’s `decode` method to translate the IDs back into text that we can read:

```
for id in input_ids[0]:  
    print(tokenizer.decode(id))
```

Which prints:

```
<s>
Write
an
email
apolog
  izing
to
Sarah
for
the
trag
  ic
garden
  ing
m
  ish
  ap
.
Exp
  lain
how
it
happened
.
```

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

- The first token is the token with ID #1, which is <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
- Notice how the space character does not have its own token. Instead, partial tokens (like ‘izing’ and ‘ic’) have a special hidden character at their beginning that indicate that they’re connected with the token that precedes them in the text.

There are three major factors that dictate how a tokenizer breaks down an input prompt. First, at model design time, the creator of the model chooses a tokenization method. Popular methods include Byte-Pair Encoding (BPE for short, widely used by GPT models), WordPiece (used by BERT), and SentencePiece (used by LLAMA). These methods are similar in that they aim to optimize an efficient set of tokens to represent a text dataset, but they arrive at it in different ways.

Second, after choosing the method, we need to make a number of tokenizer design choices like vocabulary size, and what special tokens to use. More on this in the “Comparing Trained LLM

Tokenizers” section.

Thirdly, the tokenizer needs to be trained on a specific dataset to establish the best vocabulary it can use to represent that dataset. Even if we set the same methods and parameters, a tokenizer trained on an English text dataset will be different from another trained on a code dataset or a multilingual text dataset.

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 as [Figure 6-4](#) shows.

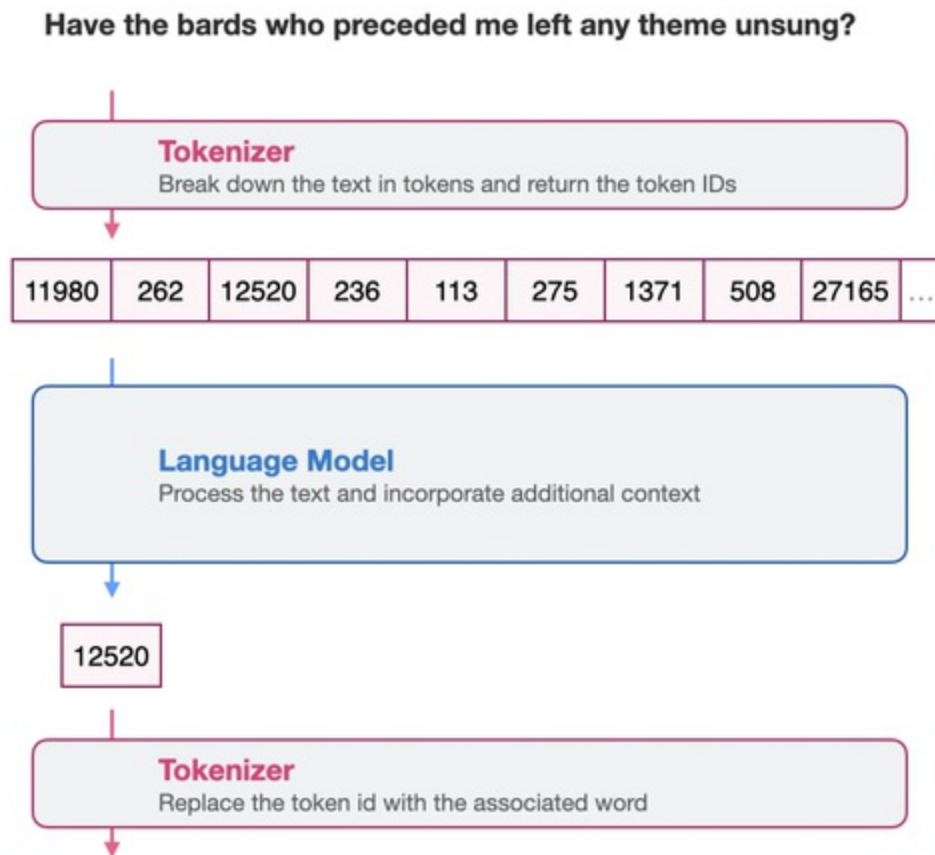


Figure 6-4. Tokenizers are also used to process the output of the model by converting the output token ID into the word or token associated with that ID.

Word vs. Subword vs. Character vs. Byte Tokens

The tokenization scheme we've seen above is called subword tokenization. It's the most commonly used tokenization scheme but not the only one. The four notable ways to tokenize are shown in [Figure 6-5](#). Let's go over them:

Word tokens

This approach was common with earlier methods like Word2Vec but is being used less and less in NLP. Its usefulness, however, led it to be used outside of NLP for use cases such as recommendation systems, as we'll see later in the chapter.

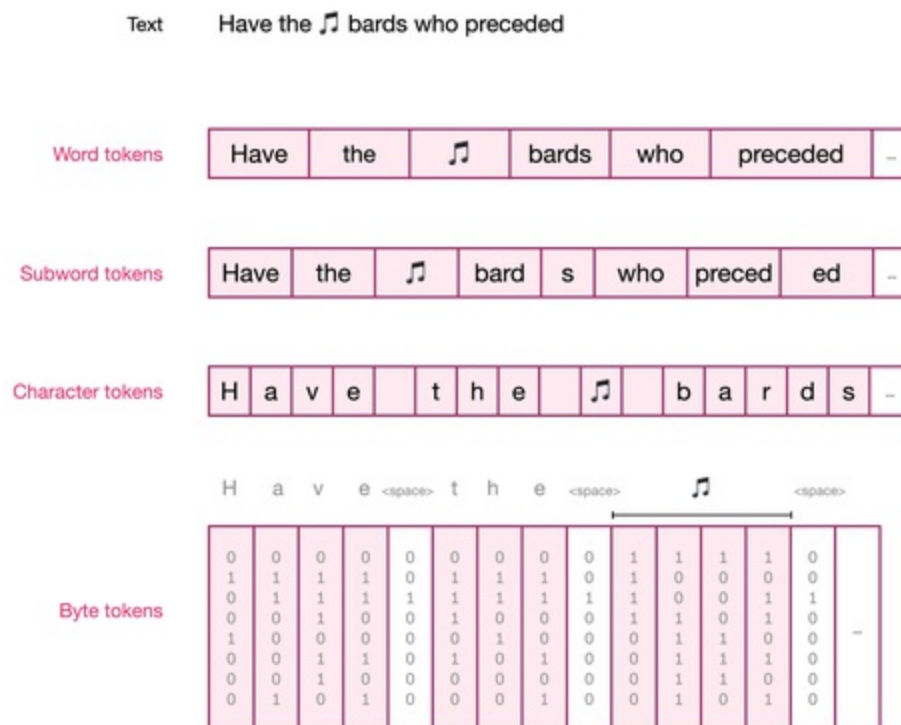


Figure 6-5. There are multiple methods of tokenization that break down the text to different sizes of components (words, subwords, characters, and bytes).

One challenge with word tokenization is that the tokenizer becomes unable to deal with new words that enter the dataset after the tokenizer was trained. It 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 we've seen as it has a token for '*apolog*', and then suffix tokens (e.g., '-y', '-ize', '-etic', '-ist') that are common with many other tokens, resulting in a more expressive vocabulary.

Subword Tokens

This method contains full and partial words. In addition to the vocabulary expressivity mentioned earlier, another benefit of the approach is its ability to represent new words by breaking the new token down into smaller characters, which tend to be a part of the vocabulary.

When compared to character tokens, this method benefits from the ability to fit more text within the limited context length of a Transformer model. So with a model with a context length of 1024, you may be able to fit three times as much text using subword tokenization than using character tokens (subword tokens often average three characters per token).

Character Tokens

This is another method that is able to deal successfully with new words because it has the raw letters to fall-back on. While that makes the representation easier to tokenize, it makes the modeling more difficult. Where a model with subword

tokenization can represent “play” as one token, a model using character-level tokens needs to model the information to spell out “p-l-a-y” in addition to modeling the rest of the sequence.

Byte Tokens

One additional tokenization method breaks down tokens into the individual bytes that are used to represent unicode characters. Papers like [CANINE: Pre-training an Efficient Tokenization-Free Encoder for Language Representation](#) outline methods like this which are also called “tokenization free encoding”. Other works like [ByT5: Towards a token-free future with pre-trained byte-to-byte models](#) show that this can be a competitive method.

One distinction to highlight here: some subword tokenizers also include bytes as tokens in their vocabulary to be the final building block to fall back to when they encounter characters they can’t otherwise represent. The GPT2 and RoBERTa tokenizers do this, for example. This doesn’t make them tokenization-free byte-level tokenizers, because they don’t use these bytes to represent everything, only a subset as we’ll see in the next section.

Tokenizers are discussed in more detail in [Suhas’ book]

Comparing Trained LLM Tokenizers

We've pointed out earlier three major factors that dictate the tokens that appear within a tokenizer: the tokenization method, the parameters and special tokens we use to initialize the tokenizer, and the dataset the tokenizer is trained on. Let's compare and contrast a number of actual, trained tokenizers to see how these choices change their behavior.

We'll use a number of tokenizers to encode the following text:

```
text = """
English and CAPITALIZATION
𐄂垠蟠
show_tokens False None elif == >= else: two tabs:"
12.0*50=600
"""
```

This will allow us to see how each tokenizer deals with a number of different kinds of tokens:

- Capitalization
- Languages other than English
- Emojis
- Programming code with its keywords and whitespaces often

used for indentation (in languages like python for example)

- Numbers and digits

Let's go from older to newer tokenizers and see how they tokenize this text and what that might say about the language model. We'll tokenize the text, and then print each token with a gray background color.

bert-base-uncased

Tokenization method: WordPiece, introduced in [Japanese and Korean voice search](#)

Vocabulary size: 30522

Special tokens: 'unk_token': '[UNK]'

'sep_token': '[SEP]'

'pad_token': '[PAD]'

'cls_token': '[CLS]'

'mask_token': '[MASK]'

Tokenized text:

```
[CLS] english and capital ##ization [UNK] [UNK] show _
```

With the uncased (and more popular) version of the BERT tokenizer, we notice the following:

- The newline breaks are gone, which makes the model blind to information encoded in newlines (e.g., a chat log when each turn is in a new line)
- All the text is in lower case
- The word “capitalization” is encoded as two subtokens capital ##ization . The ## characters are used to indicate this token is a partial token connected to the token the precedes it. This is also a method to indicate where the spaces are, it is assumed tokens without ## before them have a space before them.
- The emoji and Chinese characters are gone and replaced with the [UNK] special token indicating an “unknown token”.

bert-base-cased

Tokenization method: WordPiece

Vocabulary size: 28,996

Special tokens: Same as the uncased version

Tokenized text:

```
[CLS] English and CA ##PI ##TA ##L ##I ##Z ##AT ##ION
```

The cased version of the BERT tokenizer differs mainly in including upper-case tokens.

- Notice how “CAPITALIZATION” is now represented as eight tokens: CA ##PI ##TA ##L ##I ##Z ##AT ##ION
- Both BERT tokenizers wrap the input within a starting [CLS] token and a closing [SEP] token. [CLS] and [SEP] are utility tokens used to wrap the input text and they serve their own purposes. [CLS] stands for Classification as it’s a token used at times for sentence classification. [SEP] stands for Separator, as it’s used to separate sentences in some applications that require passing two sentences to a model (For example, in the rerankers in chapter 3, we would use a [SEP] token to separate the text of the query and a candidate result).

gpt2

Tokenization method: BPE, introduced in [Neural Machine Translation of Rare Words with Subword Units](#)

Vocabulary size: 50,257

Special tokens: <|endoftext|>

Tokenized text:

English and CAPITALIZATION

❖❖❖❖❖❖

show_tokens False None elif == >= else :

Four spaces : " " Two tabs : " "

12.0 * 50 = 600

With the GPT-2 tokenizer, we notice the following:

The newline breaks are represented in the tokenizer

Capitalization is preserved, and the word “CAPITALIZATION” is represented in four tokens

The 🎵 蟠 characters are now represented into multiple tokens each. While we see these tokens printed as the ❖ character, they actually stand for different tokens. For example, the 🎵 emoji is broken down into the tokens with token ids: 8582, 236, and 113. The tokenizer is successful in reconstructing the original character from these tokens. We can see that by printing

`tokenizer.decode([8582, 236, 113])`, which prints out 🎵

The two tabs are represented as two tokens (token number 197 in that vocabulary) and the four spaces are represented as three tokens (number 220) with the final space being a part of the token for the closing quote character.

NOTE

What is the significance of white space characters? These are important for models that understand or generate code. A model that uses a single token to represent four consecutive white space characters can be said to be more tuned to a python code dataset. While a model can live with representing it as four different tokens, it does make the modeling more difficult as the model needs to keep track of the indentation level. This is an example of where tokenization choices can help the model improve on a certain task.

google/flan-t5-xxl

Tokenization method: SentencePiece, introduced in [SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing](#)

Vocabulary size: 32,100

Special tokens:

- 'unk_token': '<unk>'

- 'pad_token': '<pad>'

Tokenized text:

English and CAPITALIZATION <unk> <unk> show _ to ken s Fal
s e None e l if = = > = else : Four spaces : " " Two tab s : " " 12.0 *
50 = 600 </s>

The FLAN-T5 family of models use the sentencepiece method. We notice the following:

- No newline or whitespace tokens, this would make it challenging for the model to work with code.
- The emoji and Chinese characters are both replaced by the <unk> token. Making the model completely blind to them.

GPT-4

Tokenization method: BPE

Vocabulary size: a little over 100,000

Special tokens:

<|endoftext|>

Fill in the middle tokens. These three tokens enable the GPT-4

capability of generating a completion given not only the text before it but also considering the text after it. This method is explained in more detail in the paper [Efficient Training of Language Models to Fill in the Middle](#). These special tokens are:

<|fim_prefix|>

<|fim_middle|>

<|fim_suffix|>

Tokenized text:

```
English and CAPITAL IZATION
? ? ? ? ? ?
show _tokens False None elif == >= else :
Four spaces : "      " Two tabs : "          "
12 . 0 * 50 = 600
```

The GPT-4 tokenizer behaves similarly with its ancestor, the GPT-2 tokenizer. Some differences are:

- The GPT-4 tokenizer represents the four spaces as a single token. In fact, it has a specific token to every sequence of white spaces up until a list of 83 white spaces.
- The python keyword elif has its own token in GPT-4. Both this

and the previous point stem from the model's focus on code in addition to natural language.

- The GPT-4 tokenizer uses fewer tokens to represent most words. Example here include 'CAPITALIZATION' (two tokens, vs. four) and 'tokens' (one token vs. three).

bigcode/starcoder

Tokenization method:

Vocabulary size: about 50,000

Special tokens:

'<|endoftext|>'

Fill in the middle tokens:

'<fim_prefix>'

'<fim_middle>'

'<fim_suffix>'

'<fim_pad>'

When representing code, managing the context is important. One

file might make a function call to a function that is defined in a different file. So the model needs some way of being able to identify code that is in different files in the same code repository, while making a distinction between code in different repos. That's why starcoder uses special tokens for the name of the repository and the filename:

'<filename>'

'<reponame>'

'<gh_stars>'

The tokenizer also includes a bunch of the special tokens to perform better on code. These include:

'<issue_start>'

'<jupyter_start>'

'<jupyter_text>'

Paper: [StarCoder: may the source be with you!](#)

Tokenized text:

English and CAPITAL IZATION

❓ ❓ ❓ ❓ ❓

```
show _ tokens False None elif == >= else :  
Four spaces : "    " Two tabs : "          "  
1 2 . 0 * 5 0 = 6 0 0
```

This is an encoder that focuses on code generation.

- Similarly to GPT-4, it encodes the list of white spaces as a single token
- A major difference here to everyone we've seen so far is that each digit is assigned its own token (so 600 becomes 6 0 0). The hypothesis here is that this would lead to better representation of numbers and mathematics. In GPT-2, for example, the number 870 is represented as a single token. But 871 is represented as two tokens (8 and 71). You can intuitively see how that might be confusing to the model and how it represents numbers.

facebook/galactica-1.3b

The galactica model described in [Galactica: A Large Language Model for Science](#) is focused on scientific knowledge and is trained on many scientific papers, reference materials, and knowledge bases. It pays extra attention to tokenization that makes it more sensitive to the nuances of the dataset it's

representing. For example, it includes special tokens for citations, reasoning, mathematics, Amino Acid sequences, and DNA sequences.

Tokenization method:

Vocabulary size: 50,000

Special tokens:

<s>

<pad>

</s>

<unk>

References: Citations are wrapped within the two special tokens:

[START_REF]

[END_REF]

One example of usage from the paper is:

Recurrent neural networks, long short-term memory

[START_REF]Long Short-Term Memory, Hochreiter[END_REF]

Step-by-Step Reasoning -

<work> is an interesting token that the model uses for chain-of-thought reasoning.

Tokenized text:

```
English and CAP ITAL IZATION
? ? ? ? ? ? ?
show _ tokens False None elif == > = else :
Four spaces : "      " Two tabs : "          "
1 2 . 0 * 5 0 = 6 0 0
```

The Galactica tokenizer behaves similar to star coder in that it has code in mind. It also encodes white spaces in the same way - assigning a single token to sequences of whitespace of different lengths. It differs in that it also does that for tabs, though. So from all the tokenizers we've seen so far, it's the only one that's assigned a single token to the string made up of two tabs ('\t\t')

We can now recap our tour by looking at all these examples side by side:

```
bert-base-uncased [CLS] english and capital ##ization [UNK
```

```
bert-base-cased
```

	[CLS] English and CA ##PI ##TA ##L ##I #:
gpt2	English and CAP ITAL IZ ATION ? ? ? ?
google/flan-t5-xxl	English and CA PI TAL IZ ATION <unk> <unl
GPT-4	English and CAPITAL IZATION ? ? ? ? ?
bigcode/starcoder	English and CAPITAL IZATION ? ? ? ? ?
facebook/galactica-1.3b	English and CAP ITAL IZATION ? ? ? ?
meta-llama/Llama-2-70b-chat-hf	<s> English and C AP IT AL IZ ATION ? ?

Notice how there's a new tokenizer added in the bottom. By now, you should be able to understand many of its properties by just glancing at this output. This is the tokenizer for LLaMA2, the most recent of these models.

Tokenizer Properties

The preceding guided tour of trained tokenizers showed a

number of ways in which actual tokenizers differ from each other. But what determines their tokenization behavior? There are three major groups of design choices that determine how the tokenizer will break down text: The tokenization method, the initialization parameters, and the dataset we train the tokenizer (but not the model) on.

Tokenization methods

As we've seen, there are a number of tokenization methods with Byte-Pair Encoding (BPE), WordPiece, and SentencePiece being some of the more popular ones. Each of these methods outlines an algorithm for how to choose an appropriate set of tokens to represent a dataset. A great overview of all these methods can be found in the Hugging Face [Summary of the tokenizers page](#).

Tokenizer Parameters

After choosing a tokenization method, an LLM designer needs to make some decisions about the parameters of the tokenizer.

These include:

Vocabulary size

How many tokens to keep in the tokenizer's vocabulary? (30K, 50K are often used vocabulary size values, but more and more we're seeing larger sizes like 100K)

Special tokens

What special tokens do we want the model to keep track of.

We can add as many of these as we want, especially if we want to build LLM for special use cases. Common choices include:

- Beginning of text token (e.g., <s>)
- End of text token
- Padding token
- Unknown token
- CLS token
- Masking token

Aside from these, the LLM designer can add tokens that help better model the domain of the problem they're trying to focus on, as we've seen with Galactica's <work> and [START_REF] tokens.

Capitalization

In languages such as English, how do we want to deal with capitalization? Should we convert everything to lower-case? (Name capitalization often carries useful information, but do we want to waste token vocabulary space on all caps versions of words?). This is why some models are released in both cased and uncased versions (like [Bert-base cased](#) and the more popular [Bert-base uncased](#)).

The Tokenizer Training Dataset

Even if we select the same method and parameters, tokenizer behavior will be different based on the dataset it was trained on (before we even start model training). The tokenization methods mentioned previously work by optimizing the vocabulary to represent a specific dataset. From our guided tour we've seen how that has an impact on datasets like code, and multilingual text.

For code, for example, we've seen that a text-focused tokenizer may tokenize the indentation spaces like this (We'll highlight some tokens in yellow and green):

```
def add_numbers(a, b):  
    ...."""Add the two numbers `a` and `b`."""  
    ....return a + b
```

Which may be suboptimal for a code-focused model. Code-focused models instead tend to make different tokenization choices:

```
def add_numbers(a, b):  
    ...."""Add the two numbers `a` and `b`."""  
    ....return a + b
```

These tokenization choices make the model's job easier and thus its performance has a higher probability of improving.

A more detailed tutorial on training tokenizers can be found in the [Tokenizers section of the Hugging Face course](#), and in [Natural Language Processing with Transformers, Revised Edition](#).

A Language Model Holds Embeddings for the Vocabulary of its Tokenizer

After a tokenizer is initialized, it is then used in the training process of its associated language model. This is why a pre-trained language model is linked with its tokenizer and can't use a different tokenizer without training.

The language model holds an embedding vector for each token in the tokenizer's vocabulary as we can see in [Figure 6-6](#). In the beginning, these vectors are randomly initialized like the rest of the model's weights, but the training process assigns them the values that enable the useful behavior they're trained to perform.



Figure 6-6. A language model holds an embedding vector associated with each token in its tokenizer.

Creating Contextualized Word Embeddings with Language Models

Now that we've covered token embeddings as the input to a language model, let's look at how language models can *create* better token embeddings. This is one of the main ways of using language models for text representation that empowers applications like named-entity recognition or extractive text summarization (which summarizes a long text by highlighting to most important parts of it, instead of generating new text as a summary).

Have the bards who preceded me left any theme unsung?

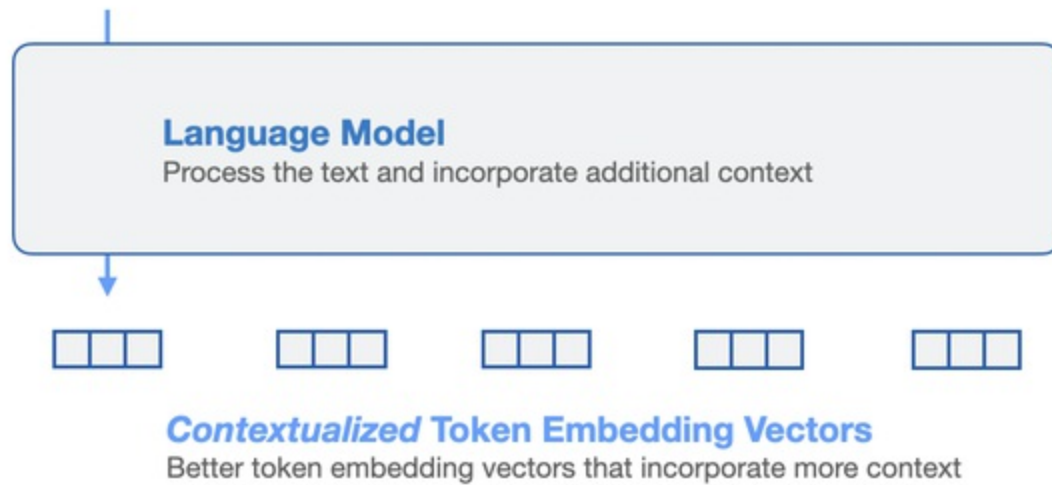


Figure 6-7. Language models produce contextualized token embeddings that improve on raw, static token embeddings

Instead of representing each token or word with a static vector, language models create contextualized word embeddings (shown in [Figure 6-7](#)) that represent a word with a different token based on its context. These vectors can then be used by other systems for a variety of tasks. In addition to the text applications we mentioned in the previous paragraph, these contextualized vectors, for example, are what powers AI image generation systems like Dall-E, Midjourney, and Stable Diffusion, for example.

Code Example: Contextualized Word Embeddings From a Language Model (Like BERT)

Let's look at how we can generate contextualized word

embeddings, the majority of this code should be familiar to you by now:

```
from transformers import AutoModel, AutoTokenizer
# Load a tokenizer
tokenizer = AutoTokenizer.from_pretrained("microsoft/d
# Load a language model
model = AutoModel.from_pretrained("microsoft/deberta-v
# Tokenize the sentence
tokens = tokenizer('Hello world', return_tensors='pt')
# Process the tokens
output = model(**tokens)[0]
```

This code downloads a pre-trained tokenizer and model, then uses them to process the string “Hello world”. The output of the model is then saved in the output variable. Let’s inspect that variable by first printing its dimensions (we expect it to be a multi-dimensional array).

The model we’re using here is called DeBERTA v3, which at the time of writing, is one of the best-performing language models for token embeddings while being small and highly efficient. It is described in the paper [DeBERTaV3: Improving DeBERTa using ELECTRA-Style Pre-Training with Gradient-Disentangled Embedding Sharing](#).

```
output.shape
```

This prints out:

```
torch.Size([1, 4, 384])
```

We can ignore the first dimension and read this as four tokens, each one embedded in 384 values.

But what are these four vectors? Did the tokenizer break the two words into four tokens, or is something else happening here? We can use what we've learned about tokenizers to inspect them:

```
for token in tokens['input_ids'][0]:  
    print(tokenizer.decode(token))
```

Which prints out:

```
[CLS]  
Hello  
world  
[SEP]
```

Which shows that this particular tokenizer and model operate by

adding the [CLS] and [SEP] tokens to the beginning and end of a string.

Our language model has now processed the text input. The result of its output is the following:

```
tensor([[
  [-3.3060, -0.0507, -0.1098, ..., -0.1704, -0.1618, 0.6
  [ 0.8918, 0.0740, -0.1583, ..., 0.1869, 1.4760, 0.0751
  [ 0.0871, 0.6364, -0.3050, ..., 0.4729, -0.1829, 1.015
  [-3.1624, -0.1436, -0.0941, ..., -0.0290, -0.1265, 0.7
  ]], grad_fn=<NativeLayerNormBackward0>)
```

This is the raw output of a language model. The applications of large language models build on top of outputs like this.

We can recap the input tokenization and resulting outputs of a language model in [Figure 6-8](#). Technically, the switch from token IDs into raw embeddings is the first step that happens inside a language model.

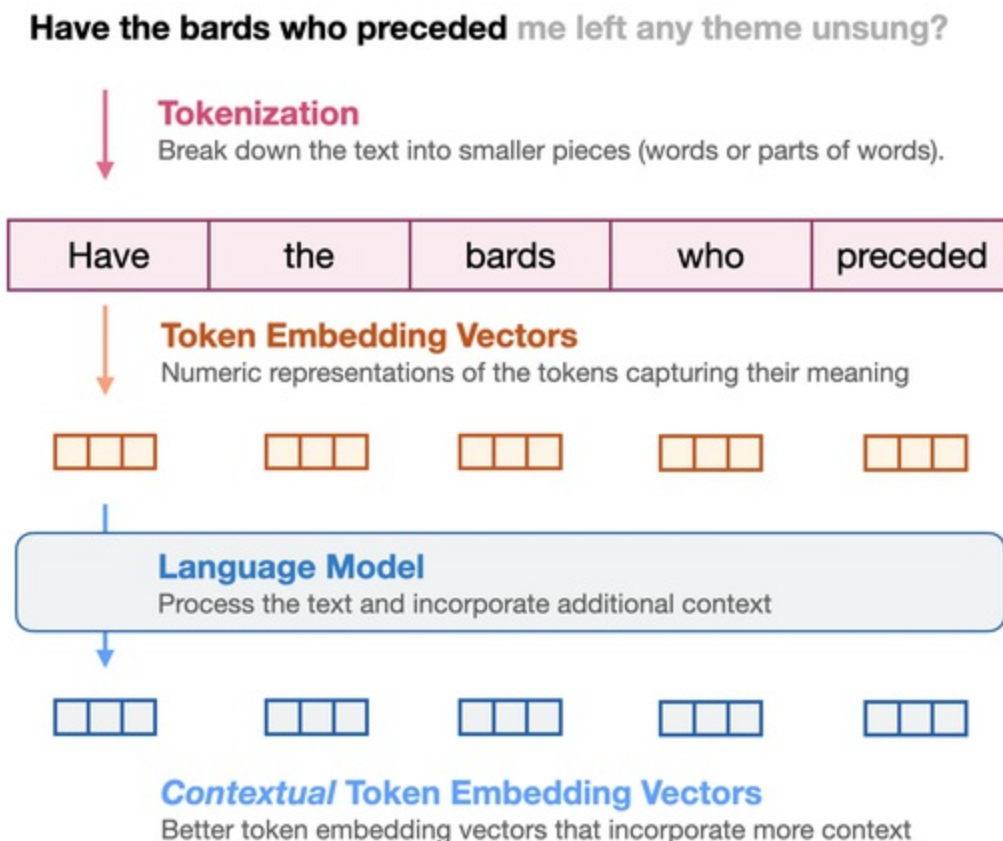


Figure 6-8. A language model operates on raw, static embeddings as its input and produces contextual text embeddings.

A visual like this is essential for the next chapter when we start to look at how Transformer-based LLMs work under the hood.

Word Embeddings

Token embeddings are useful even outside of large language models. Embeddings generated by pre-LLM methods like Word2Vec, Glove, and Fasttext still have uses in NLP and beyond NLP. In this section, we'll look at how to use pre-trained

Word2Vec embeddings and touch on how the method creates word embeddings. Seeing how Word2Vec is trained will prime you for the chapter on contrastive training. Then in the following section, we'll see how those embeddings can be used for recommendation systems.

Using Pre-trained Word Embeddings

Let's look at how we can download pre-trained word embeddings using the [Gensim](#) library

```
import gensim
import gensim.downloader as api
from sklearn.metrics.pairwise import cosine_similarity
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
# Download embeddings (66MB, glove, trained on wikipedia)
# Other options include "word2vec-google-news-300"
# More options at https://github.com/RaRe-Technologies
model = api.load("glove-wiki-gigaword-50")
```

Here, we've downloaded the embeddings of a large number of words trained on wikipedia. We can then explore the embedding

space by seeing the nearest neighbors of a specific word, 'king' for example:

```
model.most_similar([model['king']], topn=11)
```

Which outputs:

```
[('king', 1.0000001192092896),  
( 'prince', 0.8236179351806641),  
( 'queen', 0.7839043140411377),  
( 'ii', 0.7746230363845825),  
( 'emperor', 0.7736247777938843),  
( 'son', 0.766719400882721),  
( 'uncle', 0.7627150416374207),  
( 'kingdom', 0.7542161345481873),  
( 'throne', 0.7539914846420288),  
( 'brother', 0.7492411136627197),  
( 'ruler', 0.7434253692626953)]
```

The Word2vec Algorithm and Contrastive Training

The word2vec algorithm described in the paper [Efficient Estimation of Word Representations in Vector Space](#) is described in detail in [The Illustrated Word2vec](#). The central ideas are

condensed here as we build on them when discussing one method for creating embeddings for recommendation engines in the following section.

Just like LLMs, word2vec is trained on examples generated from text. Let's say for example, we have the text "*Thou shalt not make a machine in the likeness of a human mind*" from the *Dune* novels by Frank Herbert. The algorithm uses a sliding window to generate training examples. We can for example have a window size two, meaning that we consider two neighbors on each side of a central word.

The embeddings are generated from a classification task. This task is used to train a neural network to predict if words appear in the same context or not. We can think of this as a neural network that takes two words and outputs 1 if they tend to appear in the same context, and 0 if they do not.

In the first position for the sliding window, we can generate four training examples as we can see in [Figure 6-9](#).



Figure 6-9. A sliding window is used to generate training examples for the word2vec algorithm to later predict if two words are neighbors or not.

In each of the produced training examples, the word in the center is used as one input, and each of its neighbors is a distinct second input in each training example. We expect the final trained model to be able to classify this neighbor relationship and output 1 if the two input words it receives are indeed neighbors.

These training examples are visualized in [Figure 6-10](#).

Training examples	Word 1	Word 2	Target
	not	thou	1
	not	shalt	1
	not	make	1
	not	a	1

Figure 6-10. Each generated training example shows a pair of neighboring words.

If, however, we have a dataset of only a target value of 1, then a model can ace it by output 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 6-11](#).

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

Figure 6-11. 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.

It turns out that we don't have to be too scientific in how we choose the negative examples. A lot of useful models are result from simple ability to detect positive examples from randomly generated examples (inspired by an important idea called Noise Contrastive Estimation and described in [Noise-contrastive estimation: A new estimation principle for unnormalized statistical models](#)). So in this case, we get random words and add them to the dataset and indicate that they are not neighbors (and thus the model should output 0 when it sees them).

With this, we've seen two of the main concepts of word2vec

([Figure 6-12](#)): Skipgram - the method of selecting neighboring words and negative sampling - adding negative examples by random sampling from the dataset.

Skipgram					Negative Sampling		
shalt	not	make	a	machine	input word	output word	target
input		output					
make		shalt			make	shalt	1
make		not			make	aaron	0
make		a					
make		machine			make	taco	0

Figure 6-12. Skipgram and Negative Sampling are two of the main ideas behind the word2vec algorithm and are useful in many other problems that can be formulated as token sequence problems.

We can generate millions and even billions of training examples like this from running text. Before proceeding to train a neural network on this dataset, we need to make a couple of tokenization decisions, which, just like we've seen with LLM tokenizers, include how to deal with capitalization and punctuation and how many tokens we want in our vocabulary.

We then create an embedding vector for each token, and randomly initialize them, as can be seen in [Figure 6-13](#). In practice, this is a matrix of dimensions `vocab_size` x `embedding_dimensions`.

Token	Token Embedding
thou	
shalt	
make	
a	
not	
apothecary	
sublime	
def	
playback	

Figure 6-13. A vocabulary of words and their starting, random, uninitialized embedding vectors.

A model is then trained on each example to take in two embedding vectors and predict if they're related or not. We can see what this looks like in [Figure 6-14](#):



Figure 6-14. A neural network is trained to predict if two words are neighbors. It updates the embeddings in the training process to produce the final, trained embeddings.

Based on whether its prediction was correct or not, the typical machine learning training step updates the embeddings so that the next the model is presented with those two vectors, it has a better chance of being more correct. And by the end of the training process, we have better embeddings for all the tokens in our vocabulary.

This idea of a model that takes two vectors and predicts if they have a certain relation is one of the most powerful ideas in machine learning, and time after time has proven to work very well with language models. This is why we're dedicating chapter XXX to go over this concept and how it optimizes language models for specific tasks (like sentence embeddings and retrieval).

The same idea is also central to bridging modalities like text and images which is key to AI Image generation models. In that formulation, a model is presented with an image and a caption, and it should predict whether that caption describes this image or not.

Embeddings for Recommendation

Systems

The concept of token embeddings is useful in so many other domains. In industry, it's widely used for recommendation systems, for example.

Recommending songs by embeddings

In this section we'll use the Word2vec algorithm to embed songs using human-made music playlists. Imagine if we treated each song as we would a word or token, and we treated each playlist like a sentence. These embeddings can then be used to recommend similar songs which often appear together in playlists.

The [dataset](#) we'll use was collected by Shuo Chen from Cornell University. The dataset contains playlists from hundreds of radio stations around the US. [Figure 6-15](#) demonstrates this dataset.

Playlist #1: Song 1 Song 13 Song 2 Song 400

Playlist #2: Song 2 Song 81 Song 13 Song 82 Song 77

Playlist #3: Song 13 Song 2

Figure 6-15. For song embeddings that capture song similarity we'll use a dataset made up of a collection of playlists, each containing a list of songs.

Let's demonstrate the end product before we look at how it's built. So let's give it a few songs and see what it recommends in response.

Let's start by giving it Michael Jackson's *Billie Jean*, the song with ID #3822.

```
print_recommendations(3822)
title Billie Jean
artist Michael Jackson
Recommendations:
```

id	title	artist
4181	Kiss	Prince & The Revolution
12749	Wanna Be Startin' Somethin'	Michael Jackson
1506	The Way You Make Me Feel	Michael Jackson
3396	Holiday	Madonna

500	Don't Stop 'Til You Get Enough	Michael Jackson
-----	--------------------------------	-----------------

That looks reasonable. Madonna, Prince, and other Michael Jackson songs are the nearest neighbors.

Let's step away from Pop and into Rap, and see the neighbors of 2Pac's California Love:

```
print_recommendations(842)
```

id	title	artist
413	If I Ruled The World (Imagine That) (w/ Lauryn Hill)	Nas
196	I'll Be Missing You	Puff Daddy & The Family
330	Hate It Or Love It (w/ 50 Cent)	The Game

211	Hypnotize	The Notorious B.I.G.
5788	Drop It Like It's Hot (w/ Pharrell)	Snoop Dogg

Another quite reasonable list!

```
# Get the playlist dataset file
data = request.urlopen('https://storage.googleapis.com
# Parse the playlist dataset file. Skip the first two
# they only contain metadata
lines = data.read().decode("utf-8").split('\n')[2:]
# Remove playlists with only one song
playlists = [s.rstrip().split() for s in lines if len(
print( 'Playlist #1:\n ', playlists[0], '\n')
print( 'Playlist #2:\n ', playlists[1])
Playlist #1: ['0', '1', '2', '3', '4', '5', ..., '43']
Playlist #2: ['78', '79', '80', '3', '62', ..., '210']
Let's train the model:
model = Word2Vec(playlists, vector_size=32, window=20,
```

That takes a minute or two to train and results in embeddings being calculated for each song that we have. Now we can use

those embeddings to find similar songs exactly as we did earlier with words.

```
song_id = 2172
# Ask the model for songs similar to song #2172
model.wv.most_similar(positive=str(song_id))
```

Which outputs:

```
[('2976', 0.9977465271949768),
 ('3167', 0.9977430701255798),
 ('3094', 0.9975950717926025),
 ('2640', 0.9966474175453186),
 ('2849', 0.9963167905807495)]
```

And that is the list of the songs whose embeddings are most similar to song 2172. See the jupyter notebook for the code that links song ids to their names and artist names.

In this case, the song is:

```
title Fade To Black
artist Metallica
```

Resulting in recommendations that are all in the same heavy

metal and hard rock genre:

id	title	artist
11473	Little Guitars	Van Halen
3167	Unchained	Van Halen
5586	The Last In Line	Dio
5634	Mr. Brownstone	Guns N' Roses
3094	Breaking The Law	Judas Priest

Summary

In this chapter, we have covered LLM tokens, tokenizers, and useful approaches to use token embeddings beyond language models.

- Tokenizers are the first step in processing the input to a LLM -- turning text into a list of token IDs.
- Some of the common tokenization schemes include breaking text down into words, subword tokens, characters, or bytes

- A tour of real-world pre-trained tokenizers (from BERT to GPT2, GPT4, and other models) showed us areas where some tokenizers are better (e.g., preserving information like capitalization, new lines, or tokens in other languages) and other areas where tokenizers are just different from each other (e.g., how they break down certain words).
- Three of the major tokenizer design decisions are the tokenizer algorithm (e.g., BPE, WordPiece, SentencePiece), tokenization parameters (including vocabulary size, special tokens, capitalization, treatment of capitalization and different languages), and the dataset the tokenizer is trained on.
- Language models are also creators of high-quality contextualized token embeddings that improve on raw static embeddings. Those contextualized token embeddings are what's used for tasks including NER, extractive text summarization, and span classification.
- Before LLMs, word embedding methods like word2vec, Glove and Fasttext were popular. They still have some use cases within and outside of language processing.
- The Word2Vec algorithm relies on two main ideas: Skipgram and Negative Sampling. It also uses contrastive training similar to the one we'll see in the contrastive training chapter.
- Token embeddings are useful for creating and improving recommender systems as we've seen in the music

recommender we've built from curated song playlists.

Chapter 7. Creating Text Embedding Models

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. *In particular, some of the formatting may not match the description in the text: this will be resolved when the book is finalized.*

This will be the 13th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Text embedding models lie at the foundation of many powerful natural language processing applications. They lay the groundwork for empowering already impressive technologies such as text generation models. We have already used

embedding models throughout this book in a number of applications, such as supervised classification, unsupervised classification, semantic search, and even giving memory to text generation models like ChatGPT.

It is nearly impossible to overstate the importance of embedding models in the field as they are the driving power behind so many applications. As such, in this chapter, we will discuss a variety of ways that we can create and fine-tune an embedding model to increase its representative and semantic power.

Let's start by discovering what embedding models are and how they generally work.

Embedding Models

Embeddings and embedding models have already been discussed in quite a number of chapters before (Chapters X, X, and X) thereby demonstrating their usefulness. Before going into training such a model, let's recap what we have learned with embedding models before.

Unstructured textual data by itself is often quite hard to process. They are not values we can directly process, visualize and create actionable results from. We first have to convert this textual data

to something that we can easily process, numeric representations. This process is often referred to as **embedding** the input to output usable vectors, namely **embeddings** as shown in [Figure 7-1](#).

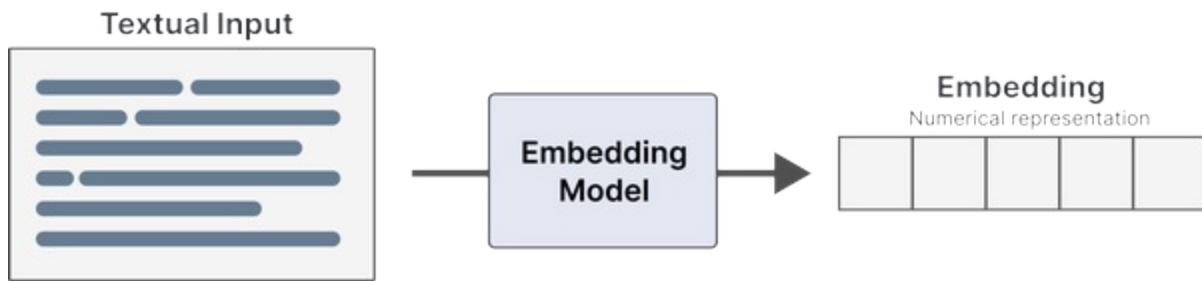


Figure 7-1. We use an embedding model to convert textual input, such as documents, sentences, and phrases to numerical representations, called embeddings.

This process of embedding the input is typically performed by an LLM, which we refer to as an *embedding model*. The main purpose of such a model is to be as accurate as possible in representing the textual data as an embedding.

However, what does it mean to be accurate in representation? Typically, we want to capture the *semantic nature*, the meaning, of documents. If we can capture the core of what the document communicates, we hope to have captured what the document is about. In practice, this means that we expect vectors of documents that are similar to one another to be similar, whereas the embeddings of documents that each discuss something entirely different should be dissimilar. This idea of semantic

similarity is visualized in [Figure 7-2](#).

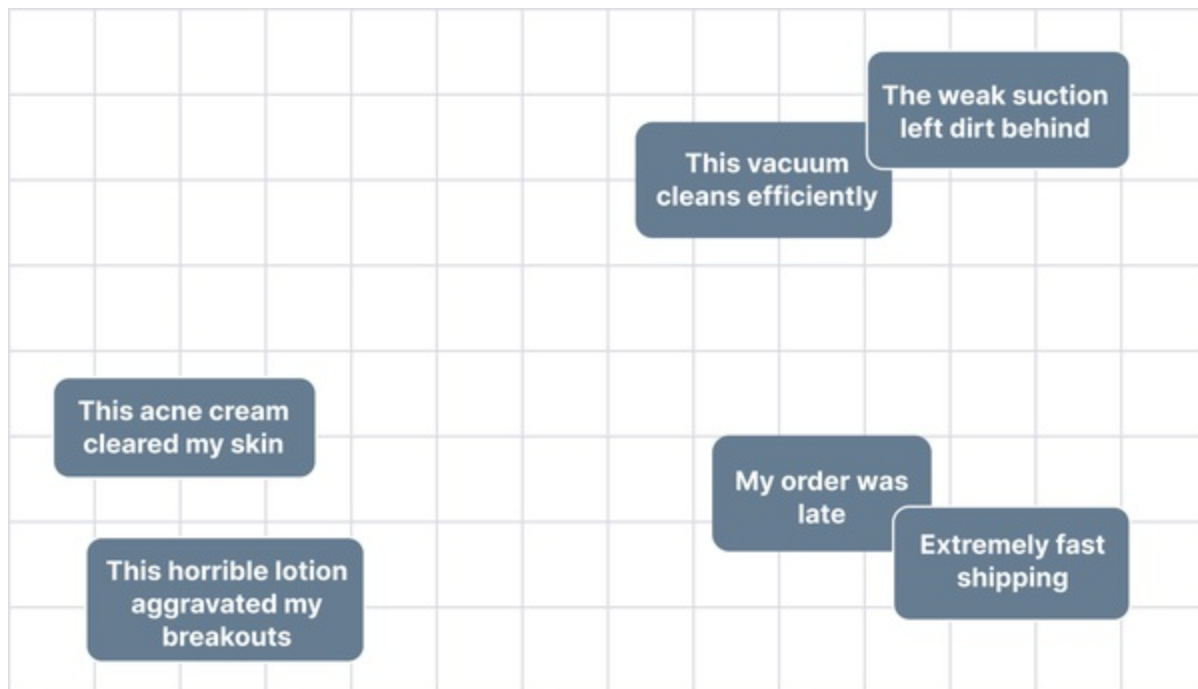


Figure 7-2. The idea of semantic similarity is that we expect textual data that have similar meaning are also closer to each other in n -dimensional space. As an example, it is illustrated here in 2-dimensional space. Do note that this is a simplified example. While 2-dimensional visualization helps illustrate the proximity and similarity of embeddings, these embeddings typically reside in high-dimensional spaces.

An embedding model, however, can be trained for a number of purposes. For example, when we are building a sentiment classifier, we are more interested in the sentiment of texts than their semantic similarity. As illustrated in [Figure 7-3](#), we can fine-tune the model such that documents are closer based on their sentiment than their semantic nature.

Either way, an embedding model aims to learn what makes certain documents similar to one another and we can guide this

process. By presenting the model with enough examples of semantically similar documents, we can steer towards semantics whereas using examples of sentiment would steer it in that direction.

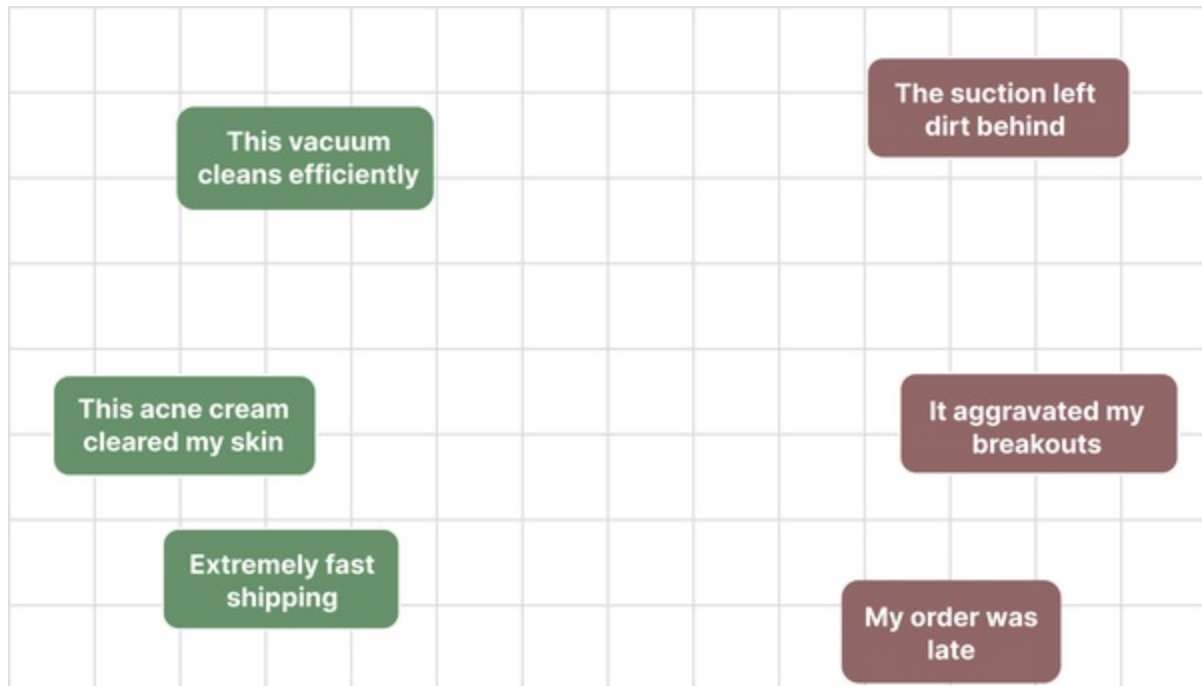


Figure 7-3. Similarity can be expressed as more than just semantically. An embedding model can be trained to focus on sentiment similarity, the idea that documents with similar sentiments are closer to each other in n-dimensional space than documents with dissimilar sentiments. In this figure, negative reviews (red) are close to one another and dissimilar to positive reviews (green).

There are many ways in which we can train, fine-tune, and guide embedding models but one of the strongest and widely-used techniques is called contrastive learning.

What is Contrastive Learning?

One major technique for both training and fine-tuning text embedding models is called contrastive learning. Contrastive learning is a technique that aims to train an embedding model such that similar documents are closer in vector space whilst dissimilar documents are further apart. We have seen this notion previously in Figures 13-X and Figure 13-X.

The underlying idea of contrastive learning is that the best way to learn and model similarity/dissimilarity between documents is by feeding a model examples of similar and dissimilar pairs. In order to accurately capture the semantic nature of a document, it often needs to be contrasted with another document for a model to learn what makes it different or similar. This contrasting procedure is quite powerful and relates to the context in which documents are written. This high-level procedure is demonstrated in [Figure 7-4](#).



Figure 7-4. Contrastive learning aims to teach an embedding model whether documents are similar or dissimilar. Contrastive learning does so by presenting groups of documents to a model that are similar or dissimilar to a certain degree.

Another way to look at contrastive learning is through the nature

of explanations. A nice example of this is an anecdotal story of a reporter asking a robber “Why did you rob a bank”, to which he answers “Because that is where the money is.”¹ Although a factually correct answer, the intent of the question was not why he robs banks specifically but why he robs at all. This is called contrastive explanation and refers to understanding a particular case, “Why P” in contrast to alternatives, “Why P and not Q?”² In the example, the question could be interpreted in a number of ways and may be best modeled by providing an alternative: “Why did you rob a bank (P) instead of obeying the law (Q)?”.

The importance of alternatives to the understanding of a question also applies to how an embedding learns through contrastive learning. By showing a model similar and dissimilar pairs of documents, it starts to learn what makes something similar/dissimilar and more importantly, why.

For example, you could learn a model to understand what a dog is by letting it find features such as “tail”, “nose”, “four legs”, etc. This learning process can be quite difficult since features are often not well-defined and can be interpreted in a number of ways. A being with a “tail”, “nose”, and “four legs” can also be a cat. To help the model steer toward what we are interested in, we essentially ask it “Why is this a dog and not a cat?”. By providing the contrast between two concepts, it starts to learn the features

that define the concept but also the features that are not related. We further illustrate this concept of contrastive explanation in [Figure 7-5](#).

Why is this a **horse**?

four legs tail fur long manes gallops ear length

Why is this a **horse** and not a **zebra**?

~~four legs~~ ~~tail~~ ~~fur~~ long manes ~~gallops~~ ear length
no stripes

Figure 7-5. Explanations are typically grounded by the contrast of other possibilities. As such, we get more information when we frame a question as a contrast. The same applies to an embedding model. When we feed it with different contrasts (degrees of similarity), it starts to learn what makes things different from one another and thereby the distinctive characteristics of concepts.

NOTE

One of the earliest and most popular examples of contrastive learning in NLP is actually Word2Vec. The model learns word representations by training on individual words in a sentence. A word close to a target word in a sentence will be constructed as a positive pair whereas randomly sampled words constitute dissimilar pairs. In other words, positive examples of neighboring words are contrasted with randomly selected words that are not neighbors. Although not widely known, it is one of the first major breakthroughs in NLP that leverages contrastive learning with neural networks.

There are many ways we can apply contrastive learning to create

text embedding models but the most well-known technique and framework is [sentence-transformers](#).

SBERT

Although there are many forms of contrastive learning, one framework that has popularized the technique within the Natural Language Processing community, is sentence-transformers. Its approach fixes a major problem with the original BERT implementation for creating sentence embeddings, namely its computational overhead. Before sentence-transformers, sentence embeddings were often used with BERT using an architectural structure called cross-encoders.

Cross-encoders allow two sentences to be passed to the transformer network simultaneously to predict the extent to which the two sentences are similar. It does so by adding a classification head to the original architecture that can output a similarity score. However, the number of computations rises quickly when you want to find the highest pair in a collection of 10,000 sentences. That would require $n \cdot (n-1) / 2 = 49\,995\,000$ inference computations and therefore generates significant overhead. Moreover, a cross-encoder generally does not generate embeddings, as shown in [Figure 7-6](#).

A solution to this overhead is by generating embeddings from a BERT model by averaging its output layer or using the [CLS] token. This, however, has shown to be worse than simply averaging word vectors, like GloVe.³

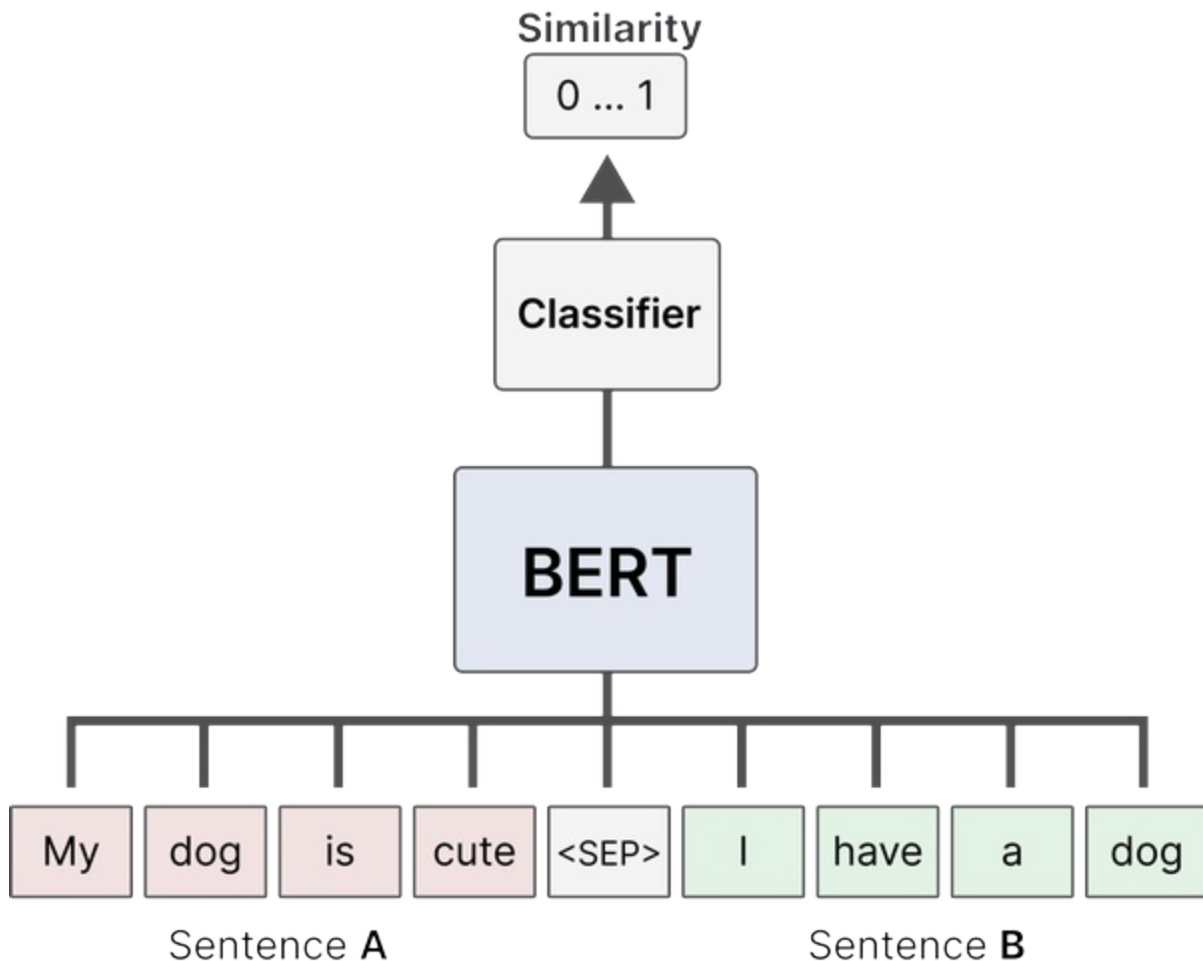


Figure 7-6. The architecture of a cross-encoder. Both sentences are concatenated, separated with a <SEP> token, and fed to the model simultaneously. Instead of outputting embeddings, it outputs a similarity score between the input sentences.

Instead, the authors of sentence-transformers approached the problem differently and searched for a method that is fast and creates embeddings that can be compared semantically. The

result is an elegant alternative to the original cross-encoder architecture. Unlike a cross-encoder, in sentence-transformers, the classification head is dropped, and instead mean pooling is used on the final output layer to generate an embedding.

Sentence-transformers are trained using a siamese architecture. In this architecture, as visualized in [Figure 7-7](#), we have two identical BERT models that share the same weights and neural architecture. Since the weights are identical for both BERT models, we can use a single model and feed it the sentences one after the other.

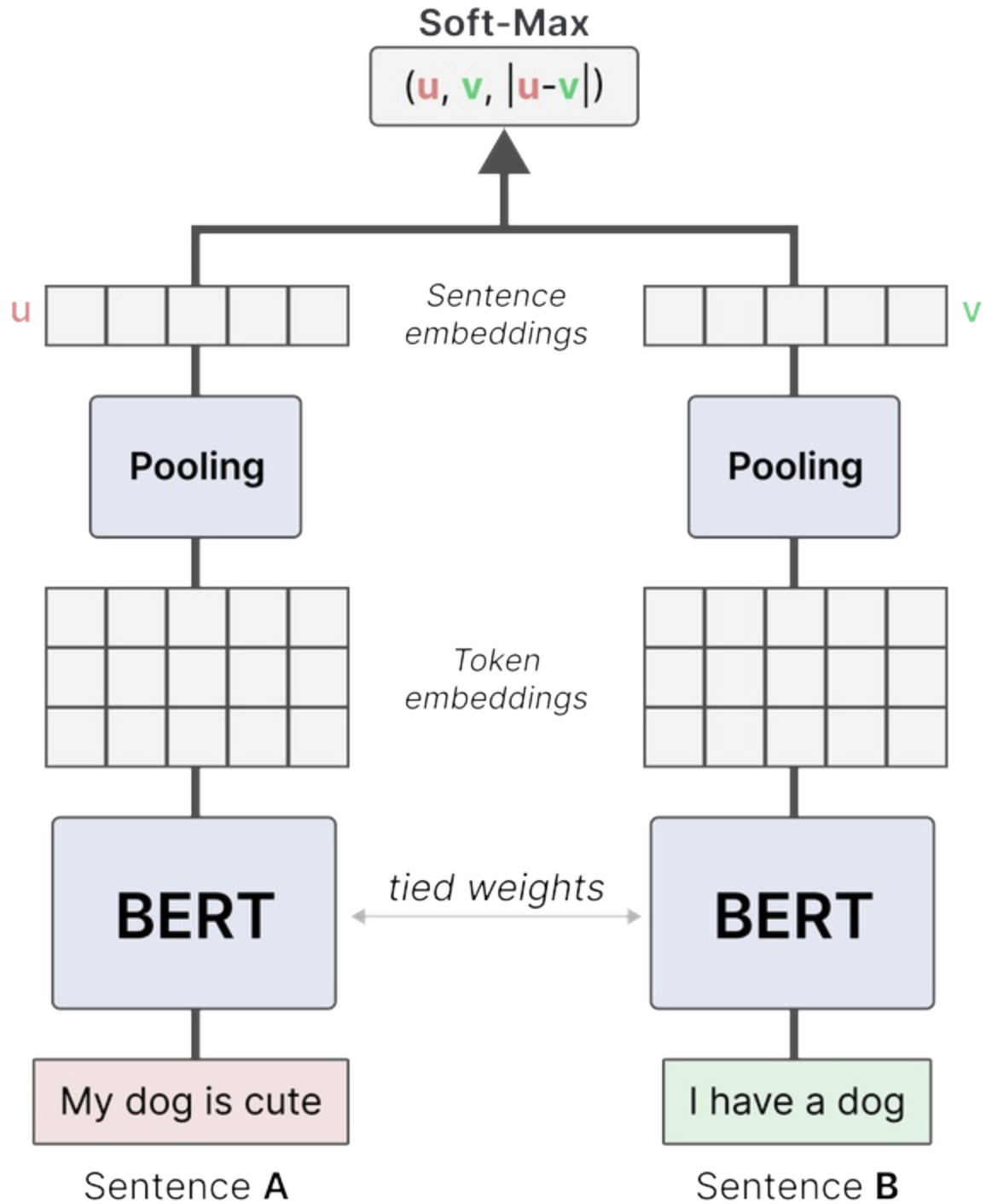


Figure 7-7. The architecture of the original sentence-transformers model which leverages a siamese network, also called a bi-encoder. BERT models with tied weights are fed the sentences from which embeddings are generated through the pooling of token embeddings. Then, models are optimized through the similarity of the sentence embeddings.

The optimization process of these pairs of sentences is done through the loss functions which can have a major impact on the model's performance. During training, the embeddings for each sentence are concatenated together with the difference between the embeddings. Then, this resulting embedding is optimized through a softmax classifier.

The resulting architecture is also referred to as a bi-encoder or SBERT for sentence-BERT. Although a bi-encoder is quite fast and creates accurate sentence representations, cross-encoders generally achieve better performance than a bi-encoder but do not generate embeddings.

The bi-encoder, like a cross-encoder, leverages contrastive learning; by optimizing the (dis)similarity between pairs of sentences, the model will eventually learn the things that make the sentences what they are.

To perform contrastive learning, we need two things. First, we need data that constitute similar/dissimilar pairs. Second, we will need to define how the model defines and optimizes similarity.

Creating an Embedding Model

There are many methods through which an embedding model

can be created but generally, we look towards contrastive learning. This is an important aspect of many embedding models as the process allows it to efficiently learn semantic representations.

However, this is not a free process. We will need to understand how to generate contrastive examples, how to train the model, and how to properly evaluate it.

Generating contrastive examples

When pre-training your embedding model, you will often see data being used from Natural Language Inference (NLI) datasets. As we described in Chapter 2, NLI refers to the task of investigating whether, for a given premise, it entails the hypothesis (entailment), contradicts it (contradiction), or neither (neutral).

For example, when the premise is “He is in the cinema watching *Coco*” and the hypothesis “He is watching *Frozen* at home”, then these statements are contradictions. In contrast, when the premise is “He is in the cinema watching *Coco*” and the hypothesis “In the movie theater he is watching the Disney movie *Coco*”, then these statements are considered entailment. This principle is illustrated in [Figure 7-8](#).

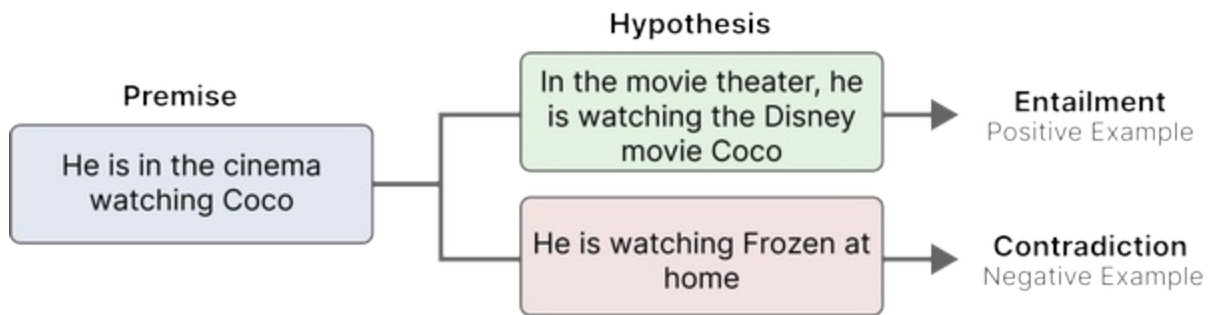


Figure 7-8. We can leverage the structure of NLI datasets to generate negative examples (contradiction) and positive examples (entailments) for contrastive learning.

If you look closely at entailment and contradiction, then they describe the extent to which two inputs are similar to one another. As such, we can use NLI datasets to generate negative examples (contradictions) and positive examples (entailments) for contrastive learning.

NOTE

Contrastive examples can also be generated if you have labeled data. In Chapter 2, we used SetFit to perform few-shot classification using sentence-transformers. In SetFit, contrastive examples were generated by comparing sentences within classes (positive examples) and between classes (negative examples).

The data that we are going to be using throughout creating and fine-tuning embedding models are derived from the General Language Understanding Evaluation benchmark ([GLUE](#)). This GLUE benchmark consists of nine language understanding tasks to evaluate and analyze model performance.

One of these tasks is the Multi-Genre Natural Language Inference (MNLI) corpus which is a collection of sentence pairs annotated with entailment (contradiction, neutral, entailment). We will be using this data to train our text embedding model. Let's load the dataset using the `datasets` package:

```
from datasets import load_dataset

# Load MNLI dataset from GLUE
# 0 = entailment, 1 = neutral, 2 = contradiction
dataset = load_dataset("glue", "mnli", split="train")
```

Next, we take a look at an example of entailment:

```
>>> dataset[2]
{'premise': 'One of our number will carry out your ins',
 'hypothesis': 'A member of my team will execute your',
 'label': 0,
 'idx': 2}
```

After having loaded the dataset, we will need to process it in such a way that it can be read with sentence-transformers:

```
from tqdm.auto import tqdm
```

```
from torch.utils.data import DataLoader
from sentence_transformers import InputExample

train_examples = [InputExample(texts=[row["premise"],
                                     label=row["label"]]) for
train_dataloader = DataLoader(train_examples, shuffle=
```

Train model

Now that we have our data loader with training examples, we will need to create our embedding model. We typically choose an existing sentence-transformer model and fine-tune that model but in this example, we are going to train an embedding from scratch.

This means that we will have to define two things. First, a pre-trained transformer model that serves as embedding individual words. As we have seen in Chapter 2, the “bert-base-uncased” model is often used for tutorials. However, many others exist that also have been evaluated using [sentence-transformers](#). Most notably, “microsoft/mpnet-base” often gives good results when used as word embedding models. Second, we will need to define the pooling strategy. Averaging the word embeddings is typically used throughout most embedding models.

```
from sentence_transformers import SentenceTransformer,

# Define a model that will embed individual words
word_embedding_model = models.Transformer('bert-base-u

# Define a model that will pool each individual words
# NOTE: This automatically uses average pooling but ot
# such as taking the maximum or mode of word embedding
pooling_model = models.Pooling(word_embedding_model.ge

# Create
model = SentenceTransformer(modules=[word_embedding_mo
```

NOTE

By default, all layers of an LLM in sentence-transformers are trainable. Although it is possible to freeze certain layers, it is generally not advised since the performance is often better when unfreezing all layers.

Next, we will need to define a loss function over which we will optimize the model. As mentioned at the beginning of this section, one of the first instances of sentence-transformers uses soft-max loss. For illustrative purposes, we are going to be using that for now but we will go into more performant losses later on:

```
from sentence_transformers import losses

# Define the loss function. In soft-max loss, we will
train_loss = losses.SoftmaxLoss(model=model, sentence_
```

Now that we have defined our data, embedding model, and loss we can start training our model. We can do that using the `fit` function:

```
# Train our model for a single epoch
model.fit(train_objectives=[(train_dataloader, train_l
```

We train our model for a single epoch which takes roughly an hour or so on a V100 GPU. And... that's it! We have now trained our own embedding model from scratch.

NOTE

The sentence-transformers framework allows for multi-task learning. The `train_objectives` parameter accepts a list of tuples which makes it possible to give it different datasets each with their own objective to optimize for. This means that we could give it the entire GLUE benchmark to train on.

We can perform a quick evaluation of the performance of our

model. There are many tasks for doing, which we will go in-depth later, but a good one to start with is the Semantic Textual Similarity Benchmark (STSB) that is found in the GLUE dataset as we have seen before.

It is a collection of sentence pairs labeled, through human annotation, with similarity scores between 1 and 5.

We can leverage this dataset to see how well our model scores on a semantic similarity task. First, we will need to process the STSB dataset:

```
import datasets
sts = datasets.load_dataset('glue', 'stsb', split='val')

# Make sure every value is between 0 and 1
sts = sts.map(lambda x: {'label': x['label'] / 5.0})

# Process the data to be used from sentence_transformers
samples = [InputExample(texts=[sample['sentence1'], sample['sentence2']],
                        label=sample['label']) for sample in sts.iter_instances()]
```

We can use these samples to generate an evaluator using sentence-transformers:

```
from sentence_transformers.evaluation import Embedding

# Create an embedding similarity evaluator for sts
evaluator = EmbeddingSimilarityEvaluator.from_input_ex
```

This evaluator allows us to evaluate any model, so let's compare our trained model with its untrained variant:

```
>>> # Evaluate the original model
>>> orig_model = SentenceTransformer('bert-base-uncase
>>> print("Baseline: ", evaluator(orig_model))
>>>
>>> # Evaluate our trained model
>>> print("Trained model: ", evaluator(model))

"Baseline: 0.6146254081453191"
"Trained model: 0.7376971430125273"
```

This training procedure improved the baseline score from 0.61 to 0.74!

In-depth Evaluation

A good embedding model is more than just a good score on the STSB benchmark! As we have seen before, the GLUE benchmark

has a number of tasks for which we can evaluate our embedding model. However, there exist many more benchmarks that allow for the evaluation of embedding models. To unify this evaluation procedure, the Massive Text Embedding Benchmark (MTEB)⁴ was developed. This MTEB spans 8 embedding tasks that cover 58 datasets and 112 languages.

To publicly compare the state-of-the-art embedding models, a [leaderboard](#) was created with the scores of each embedding model across all tasks.

```
from mteb import MTEB

# Choose evaluation task
evaluation = MTEB(tasks=["Banking77Classification"])

# Calculate results
results = evaluation.run(model)
```

When we inspect the results, we can see a number of evaluation metrics for this task:

```
>>> results

{'Banking77Classification': {'mteb_version': '1.0.2',
```

```
'dataset_revision': '0fd18e25b25c072e09e0d92ab615fda'
'mteb_dataset_name': 'Banking77Classification',
'test': {'accuracy': 0.7825324675324674,
'f1': 0.782082703333302,
'accuracy_stderr': 0.010099229383338676,
'f1_stderr': 0.010381981136492737,
'main_score': 0.7825324675324674,
'evaluation_time': 23.44}}}
```

The great thing about this evaluation benchmark is not only the diversity of the tasks and languages but that even the evaluation time is saved. Although many embedding models exist, we typically want those that are both accurate and have low latency. The tasks for which embedding models are used, like semantic search, often benefit from and require to have fast inference.

Loss Functions

We trained [our model](#) using SoftMaxLoss to illustrate how one of the first sentence-transformers models was trained. However, not only is there a large variety of loss functions to choose from, SoftMaxLoss is generally not advised as there are more performant losses.

Instead of going through every single loss function out there,

there are two loss functions that are typically used and seem to perform generally well, namely:

- Cosine Similarity
- Multiple Negatives Ranking Loss

NOTE

There are many more loss functions to choose from than just those discussed here. For example, a loss like MarginMSE works great for training or fine-tuning a cross-encoder. There are a number of interesting [loss functions](#) implemented in the sentence-transformers framework.

Cosine Similarity

The Cosine Similarity Loss is an intuitive and easy-to-use loss that works across many different use cases and datasets. However, this loss is typically used in semantic textual similarity tasks. In these tasks, a similarity score is assigned to the pairs of texts over which we optimize the model.

Instead of having strictly positive or negative pairs of sentences, we assume to have pairs of sentences that are similar or dissimilar to a certain degree. Typically, this value lies between 0 and 1 to indicate dissimilarity and similarity respectively ([Figure 7-9](#)).

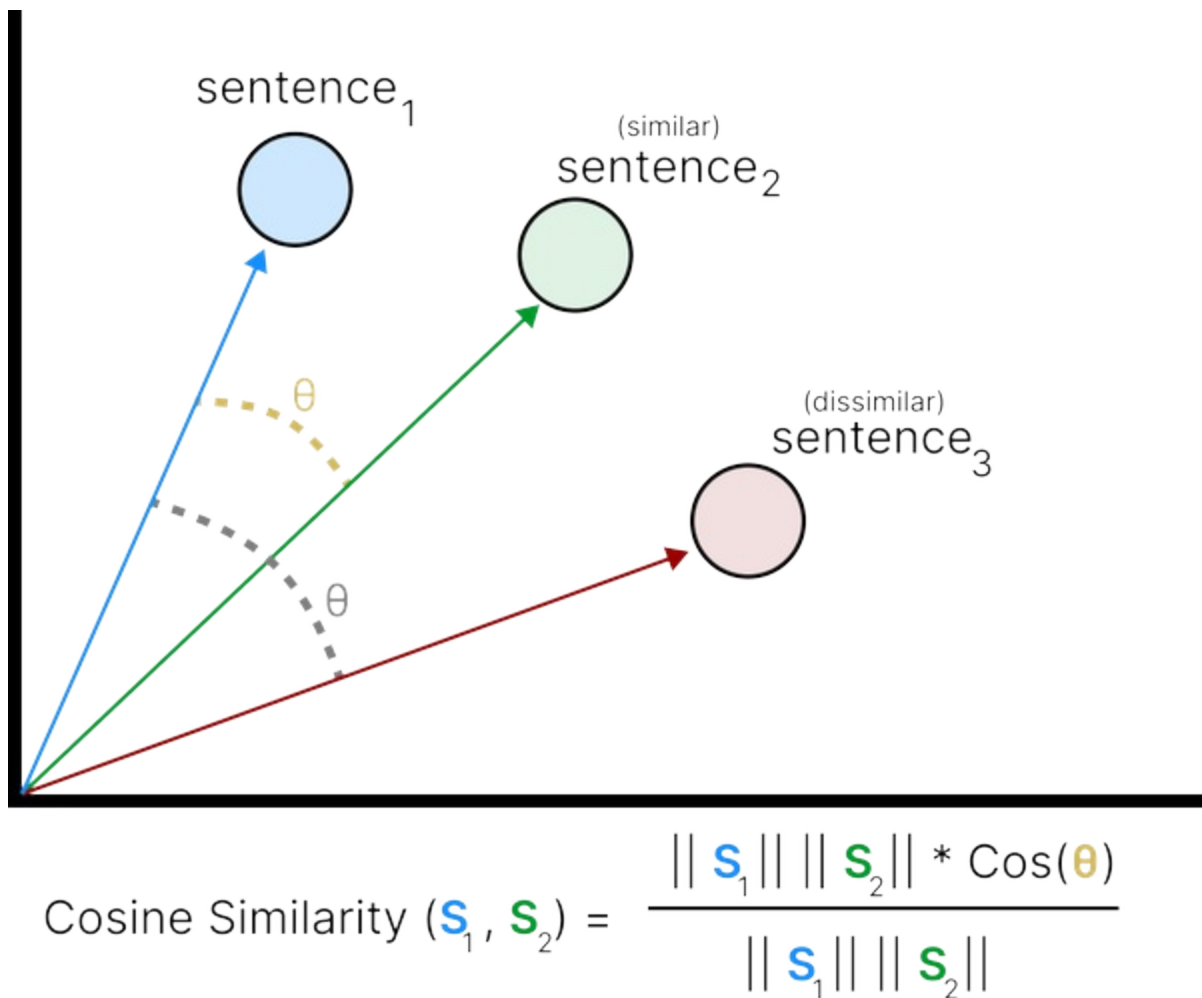


Figure 7-9. The Cosine Similarity Loss aims to minimize the cosine distance between semantically similar sentences and to maximize the distance between semantically dissimilar sentences.

The Cosine Similarity Loss is straightforward—it calculates the cosine similarity between the two embeddings of the two texts and compares that to the labeled similarity score. The model will learn to recognize the degree of similarity between sentences.

A minimal example of training with Cosine Similarity Loss would

be:

```
# Prepare data
sts = datasets.load_dataset('glue', 'stsb', split='train')
sts = sts.map(lambda x: {'label': x['label'] / 5.0})
train_examples = [InputExample(texts=[sample['sentence1'],
                                     sample['sentence2']],
                               label=sample['label']) for sample in sts]
train_dataloader = DataLoader(train_examples, shuffle=True)

# Define model
word_embedding_model = models.Transformer('bert-base-uncased')
pooling_model = models.Pooling(word_embedding_model.get_output_embeddings())
cosine_model = SentenceTransformer(modules=[word_embedding_model, pooling_model])

# Loss function
train_loss = losses.CosineSimilarityLoss(model=cosine_model)

# Fit model
cosine_model.fit(train_objectives=[(train_dataloader, train_loss)])
```

We are using the STSB dataset for this example. As we have seen before, they are pairs of sentences with annotated similarity scores that lend themselves naturally to the Cosine Similarity Loss.

```
>>> # Evaluate trained model with MNR loss
```

```
>>> print("Trained model + Cosine Similarity Loss: ",  
"Trained model + Cosine Similarity Loss: 0.84802799431")
```

A score of 0.848 is a big improvement compared to the SoftMaxLoss example. However, since our training and evaluation are both on the same tasks in contrast to the SoftMaxLoss example, comparing them would not be fair.

Multiple Negatives Ranking Loss

Multiple Negatives Ranking (MNR⁵) loss, often referred to as InfoNCE⁶ or NTXentLoss,⁷ is a loss that, in principle, only uses positive pairs of sentences.

For example, you might have pairs of question/answer, image/image caption, paper title/paper abstract, etc. The great thing about these pairs is that we can be confident they are hard positive pairs. In MNR Loss ([Figure 7-10](#)), negative pairs are constructed by mixing a positive pair with another positive pair. In the example of a paper title and abstract, you would generate a negative pair by combining the title of a paper with a completely different abstract. These negatives are called in-batch negatives.

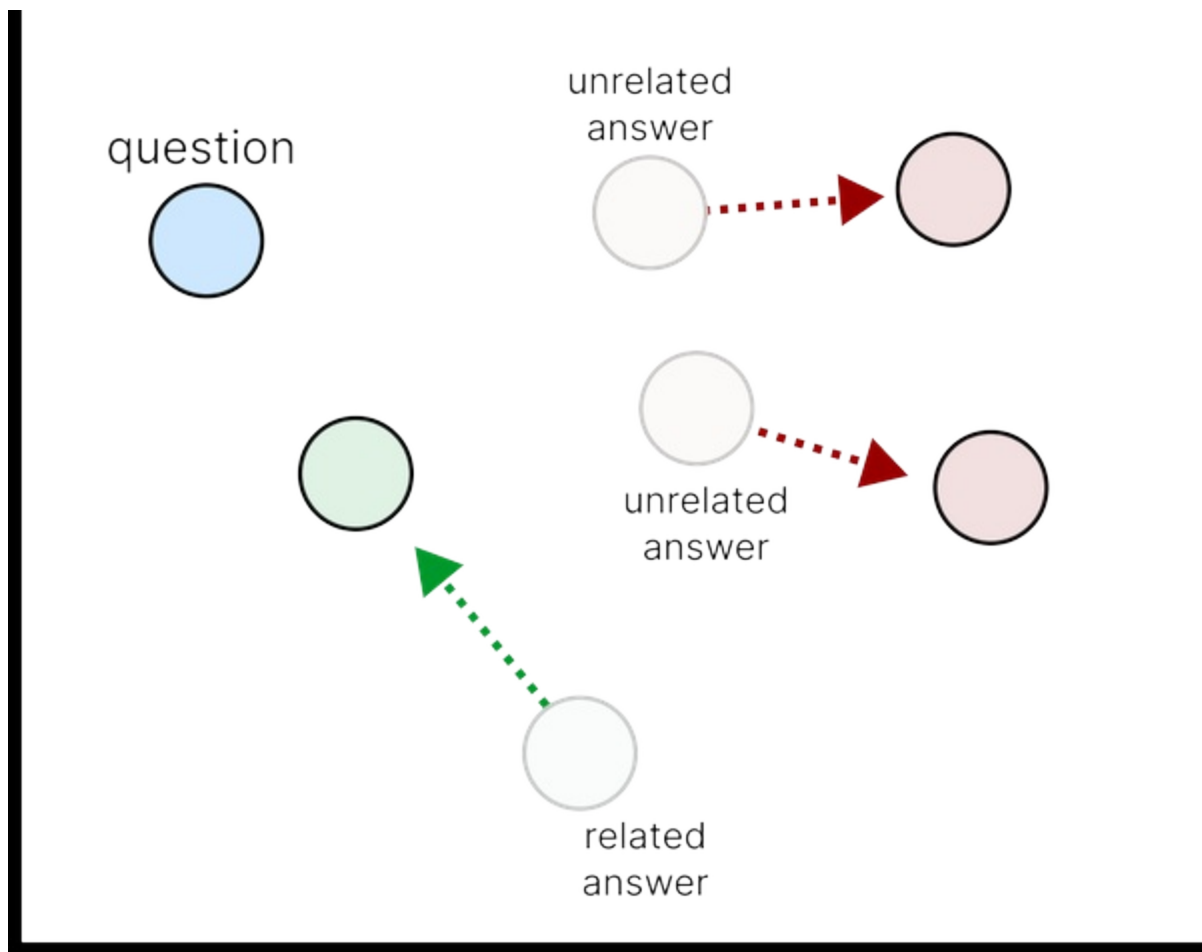


Figure 7-10. Multiple Negatives Ranking Loss aims to minimize the distance between related pairs of text, such as questions and answers, and maximize the distance between unrelated pairs, such as questions and unrelated answers.

After having generated these positive and negative pairs, we calculate their embeddings and apply cosine similarity. These similarity scores are then used to answer the question, are these pairs negative or positive? In other words, it is treated as a classification task and we can use cross-entropy loss to optimize the model.

The following is a minimal example of training with MNR loss:

```
# Prepare data and only keep positive pairs
dataset = load_dataset("glue", "mnli", split="train")
dataset = dataset.filter(lambda x: True if x['label']
train_examples = [InputExample(texts=[row["premise"],
train_dataloader = DataLoader(train_examples, shuffle=

# Define model
word_embedding_model = models.Transformer('bert-base-u
pooling_model = models.Pooling(word_embedding_model.ge
mnr_model = SentenceTransformer(modules=[word_embeddin

# Loss function
train_loss = losses.MultipleNegativesRankingLoss(model

# Fit model
mnr_model.fit(train_objectives=[(train_dataloader, tra
```

Note that we used the same data as in our SoftMaxLoss example. Let's compare the performance of this model with that of our previously trained models:

```
>>> # Evaluate trained model with MNR loss
>>> print("Trained model + MNR Loss: ", evaluator(mnr_
```

```
"Trained model + MNR Loss: 0.8183052945831789"
```

Compared to our previously trained model with SoftMaxLoss (0.74), our model with MNR Loss (0.82) seems to be much more accurate!

TIP

Larger batch sizes tend to be better with MNR Loss as a larger batch makes the task more difficult. The reason for this is that the model needs to find the best matching sentence from a larger set of potential pairs of sentences. You can adapt the code to try out different batch sizes and get a feeling of its effects.

There is a downside to how we used this loss function. Since negatives are sampled from other question/answer pairs, these in-batch or “easy” negatives that we used could potentially be completely unrelated to the question. As a result, the embedding model’s task of then finding the right answer to a question becomes quite easy. Instead, we would like to have negatives that are very related to the question but not the right answer. These negatives are called hard negatives. Since this makes the task more difficult for the embedding model as it has to learn more nuanced representations, the embedding model’s performance generally improves quite a bit.

NOTE

In MNR Loss, cosine similarity is often used but could be replaced with the dot product instead. An advantage of using the dot product is that it tends to work better for longer texts as the dot product will increase. A disadvantage, however, is that it generally works less well for clustering tasks.

A good example of a hard negative is the following. Let's assume we have the following question: "How many people live in Amsterdam?". A related answer to this question would be: "Almost a million people live in Amsterdam". To generate a good hard negative, we ideally want the answer to contain something about Amsterdam and the number of people living in this city. For example: "More than a million people live in Utrecht, which is more than Amsterdam." This answer is unrelated to the question but very similar, so this would be a good hard negative. [Figure 7-11](#) illustrates the differences between easy and hard negatives.

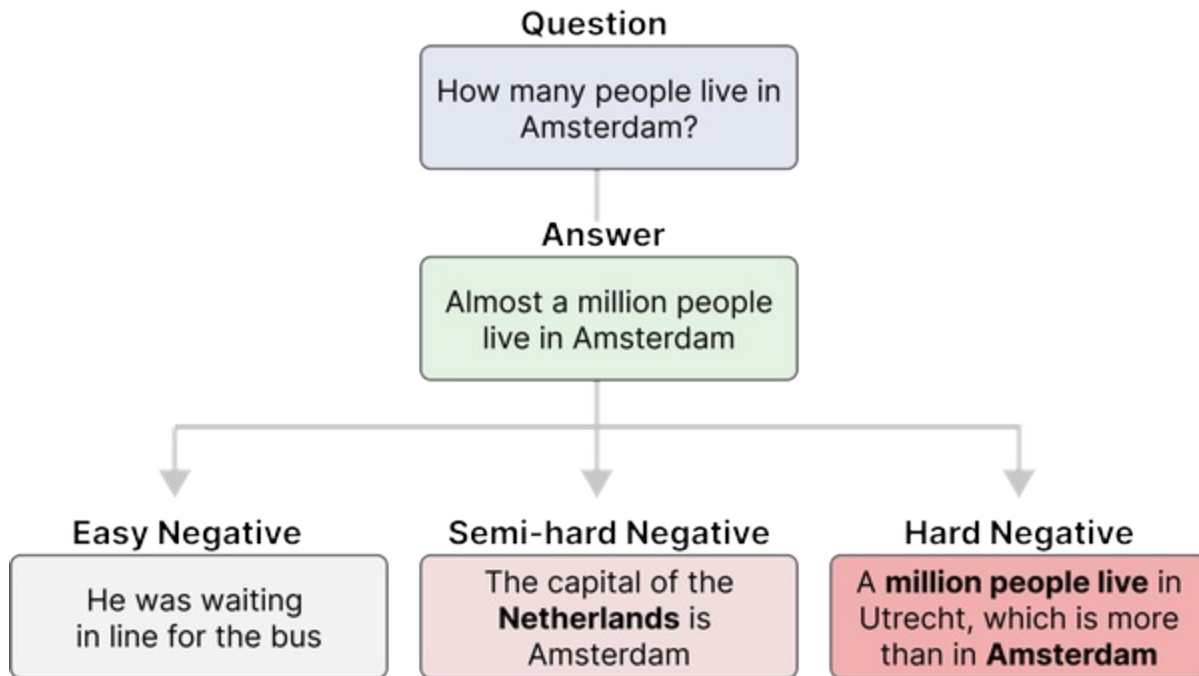


Figure 7-11. [negatives.png] An easy negative is typically unrelated to both the question and answer. A semi-hard negative has some similarities to the topic of the question and answer but is somewhat unrelated. A hard negative is very similar to the question but is generally the wrong answer.

Using hard negatives with this loss is rather straightforward, instead of passing two related texts to the `InputExample`, we pass three texts, two related texts, and the last text a hard-negative.

Fine-tuning an Embedding Model

In the previous section, we went through the basics of training an embedding model from scratch and saw how we could leverage loss functions to further optimize its performance. This method, although quite powerful, requires creating an

embedding model from scratch. This process can be quite costly and time-consuming.

Instead, sentence-transformers have a number of pre-trained embedding models that we can use as a base for fine-tuning. They are trained on large amounts of data and already generate great performance out-of-the-box.

There are a number of ways to fine-tune your model, depending on the data availability and domain. We will go through a number of them and demonstrate the strength of leveraging pre-trained embedding models.

Supervised

The most straightforward way to fine-tune an embedding model is to repeat the process of training our model as we did before, but replace the `'bert-base-uncased'` with a pre-trained sentence-transformers model. There are many to choose from but generally, the `'all-mpnet-base-v2'` (https://www.sbert.net/docs/pretrained_models.html) performs well across many use cases (see <https://huggingface.co/spaces/mteb/leaderboard>).

In practice, we would only need to run the following to fine-tune our model:

```
from sentence_transformers import SentenceTransformer

# Prepare data and only keep positive pairs
dataset = load_dataset("glue", "mnli", split="train")
dataset = dataset.filter(lambda x: True if x['label']
train_examples = [InputExample(texts=[row["premise"],
train_dataloader = DataLoader(train_examples, shuffle=

# Load a pre-trained model
model = SentenceTransformer('all-mpnet-base-v2')

# Loss function
train_loss = losses.MultipleNegativesRankingLoss(model

# Fine-tune our model for a single epoch
model.fit(train_objectives=[(train_dataloader, train_l
```

Here, we use the same data as we used to train our model in the Multiple Negatives Ranking Loss example. The code for fine-tuning your model is rather straightforward thanks to the incredibly well-built sentence-transformers package.

TIP

Instead of using a pre-trained BERT model like 'bert-base-uncased' or a possible out-of-domain model like 'all-mpnet-base-v2', you can also perform Masked Language Modeling on the pre-trained BERT model to first adapt it to your domain.

Then, you can use this fine-tuned BERT model as the base for training your embedding model. This is a form of domain adaptation. You can find more about Masked Language Modeling in Chapter X.

The main difficulty of fine-tuning/training your model is finding the right data. With these models, we not only want to have very large datasets, the data in itself needs to be of high quality.

Developing positive pairs is generally straightforward but adding hard negative pairs significantly increases the difficulty of creating quality data.

NOTE

Training on the data, as shown above, is a bit redundant since the model was already trained on a very similar NLI dataset. However, the procedure remains the same for fine-tuning it on your domain-specific data.

Augmented SBERT

A disadvantage of training or fine-tuning these embedding models is that they often require substantial training data. Many of these models are trained with more than a billion sentence pairs. Extracting such a high number of sentence pairs for your use case is generally not possible as in many cases, there is only little data available.

Fortunately, there is a way to augment your data such that an embedding model can be fine-tuned when there is only a little labeled data available. This procedure is referred to as Augmented SBERT.⁸

In this procedure, we aim to augment the small amount of labeled data such that they can be used for regular training. It makes use of the slow and more accurate cross-encoder architecture (BERT) to augment and label a larger set of input pairs. These newly labeled pairs are then used for fine-tuning a bi-encoder (SBERT).

As shown in [Figure 7-12](#), Augmented SBERT involves the following steps:

1. Fine-tune a cross-encoder (BERT) using a small annotated dataset (gold dataset)
2. Create new sentence pairs
3. Label new sentence pairs with the fine-tuned cross-encoder (silver dataset)
4. Train a bi-encoder (SBERT) on the extended dataset (gold + silver dataset)

Here, a gold dataset is a small but fully annotated dataset that holds the ground truth. A silver dataset is also fully annotated

but is not necessarily the ground truth as it was generated through predictions of the cross-encoder.

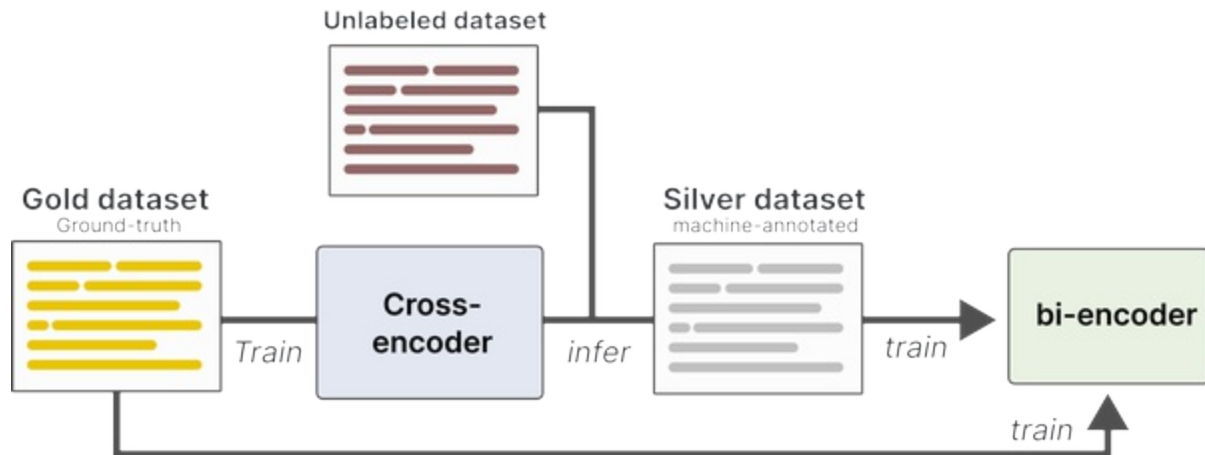


Figure 7-12. Augmented SBERT works through training a high-performant cross-encoder on a small gold dataset. Then, the trained cross-encoder can be used to label an unlabeled dataset to generate the silver dataset which is much bigger than the gold dataset. Finally, both the gold and silver datasets are used to train the bi-encoder.

Before we get into the steps above, let us first prepare the data:

```
# Make sure every value is between 0 and 1
sts = datasets.load_dataset('glue', 'stsb', split='train')
sts = sts.map(lambda x: {'label': x['label'] / 5.0})

# Process the data to be used in sentence_transformers
gold_examples = [InputExample(texts=[sample['sentence1'], sample['sentence2']], label=sample['label']) for sample in sts]
gold_dataloader = DataLoader(gold_examples, shuffle=True)

# Fully labeled gold dataset
gold = pd.DataFrame({'sentence1': sts['sentence1'], 'sentence2': sts['sentence2'], 'label': sts['label']})
```

We use the train split of the STSB corpus as our gold dataset and use it to train our cross-encoder (step 1):

```
from sentence_transformers.cross_encoder import CrossEncoder

# Train a cross-encoder on the gold dataset
cross_encoder = CrossEncoder('bert-base-uncased', num_epochs=100)
cross_encoder.fit(train_data_loader=gold_data_loader, dev_data_loader=silver_data_loader)
```

After having trained our cross-encoder, we can generate new candidate sentence pairs by simply random sampling 10 sentences for each input sentence (step 2):

```
# Prepare unlabeled to-be silver dataset
silver = pd.DataFrame(columns=["sentence1", "sentence2"])
for sentence in gold.sentence1:
    sampled = gold[gold['sentence1'] != sentence].sample(n=10)
    sampled.sentence1 = sentence
    silver = pd.concat([silver, sampled], ignore_index=True)
silver = silver.drop_duplicates()
```

TIP

Instead of randomly sampling the silver sentence pairs, we can also use pre-trained sentence-transformers. By retrieving the top-k sentences from the dataset using semantic search, the silver sentence pairs that we create tend to be more accurate.

Although the sentence pairs are still chosen based on an approximation it is much better than random sampling.

The cross-encoder that we trained in step 1 can then be used to label the candidate sentence pairs generated in the previous step to build up the silver dataset:

```
# Predict labels for the unlabeled silver data
pairs = list(zip(silver['sentence1'], silver['sentence2']))
silver['label'] = cross_encoder.predict(pairs)
```

Now that we have a silver and gold dataset, we simply combine them and train our embedding model as we did before:

```
# Combine gold + silver
data = pd.concat([gold, silver], ignore_index=True, axis=0)
data = data.drop_duplicates(subset=['sentence1', 'sentence2'])

# initialize dataloader
examples = [InputExample(texts=[sample['sentence1'], sample['sentence2']], labels=[sample['label']]) for sample in data.to_dict('records')]
dataloader = DataLoader(examples, shuffle=True, batch_size=16)

# Initialize bi-encoder
word_embedding_model = models.Transformer('bert-base-uncased')
pooling_model = models.Pooling(word_embedding_model.get_word_embeddings())
```

```
model = SentenceTransformer(modules=[word_embedding_mo

# Loss function
loss = losses.CosineSimilarityLoss(model=model)

# Fine-tune our model for a single epoch
model.fit(train_objectives=[(dataloader, loss)], epoch
```

We can run the above again with only the gold dataset and see how adding this silver dataset influences the performance. Training on only the gold dataset results in a performance of 0.804 whereas adding the silver dataset ups the performance to 0.830!

This method allows for increasing the datasets that you already have available without the need to manually label hundreds of thousands of sentence pairs.

Unsupervised Learning

To create an embedding model, we typically need labeled data. However, not all real-world datasets come with a nice set of labels that we can use. We would instead look for techniques to train the model without any pre-determined labels—unsupervised learning.

Unsupervised techniques for creating or fine-tuning an embedding model generally perform worse than their supervised alter-egos. Many approaches exist, like Simple Contrastive Learning of Sentence Embeddings (SimCSE)⁹, Contrastive Tension (CT)¹⁰, Transformer-based Denoising AutoEncoder (TSDAE)¹¹, and Generative Pseudo-Labeling (GPL)¹².

We will be going through two methods, TSDAE and GPL which we can even combine later on.

Transformer-based Denoising AutoEncoder

TSDAE is a very elegant approach to creating an embedding model with unsupervised learning. The method assumes that we have no labeled data at all and does not require us to artificially create labels.

The underlying idea of TSDAE is that we add noise to the input sentence by removing a certain percentage of words from it. This “damaged” sentence is put through an encoder, with a pooling layer on top of it, to map it to a sentence embedding.

From this sentence embedding, a decoder tries to reconstruct the original sentence from the “damaged” sentence but without the

artificial noise.

This method is very similar to Masked Language Modeling, where we try to reconstruct and learn certain masked words. Here, instead of reconstructing masked words, we try to reconstruct the entire sentence.

After training, we can use the encoder to generate embeddings from text since the decoder is only used for judging whether the embeddings can accurately reconstruct the original sentence ([Figure 7-13](#)).

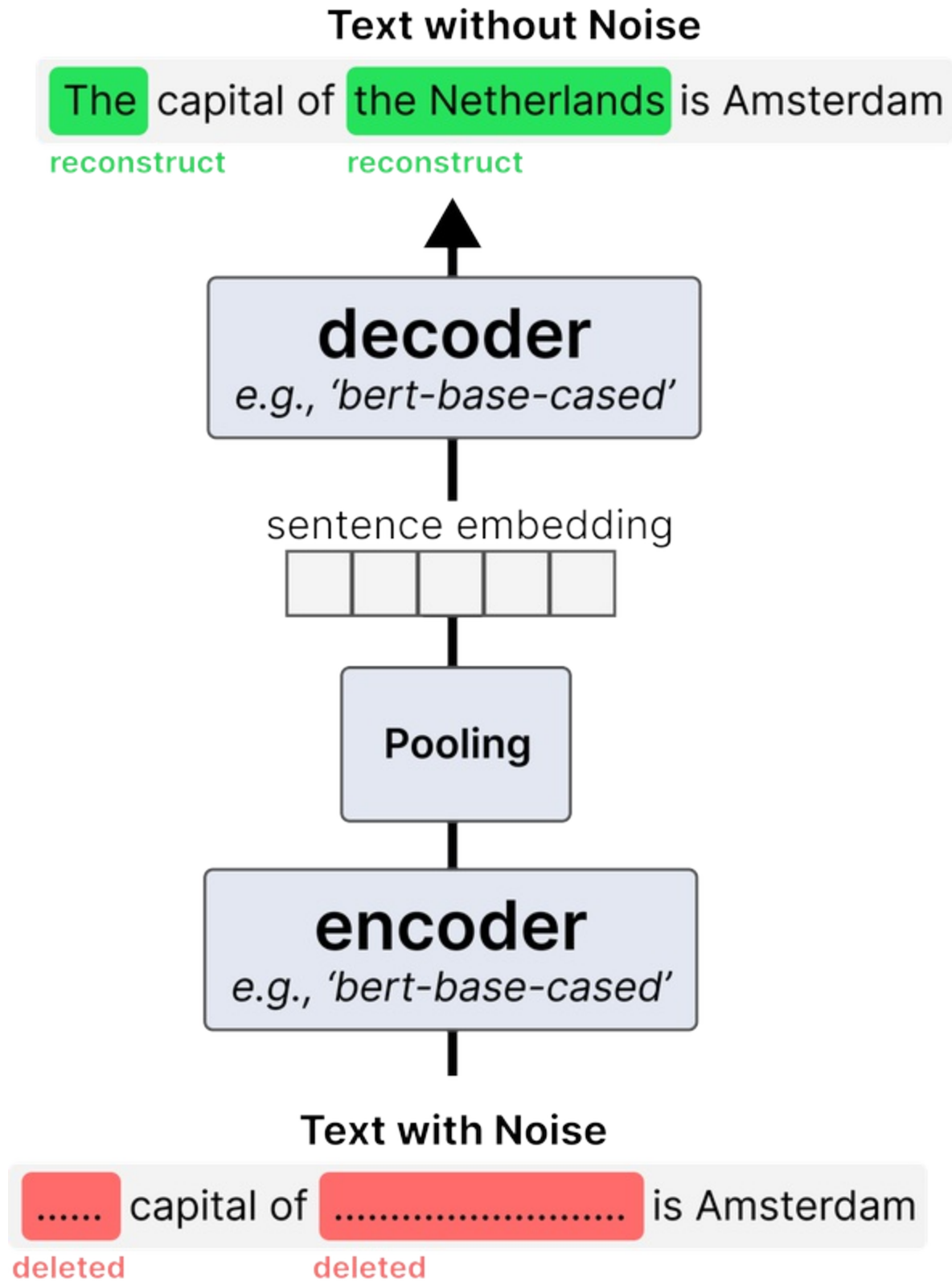


Figure 7-13. TSDAE randomly removes words from an input sentence which is passed through an encoder to generate a sentence embedding. From this sentence embedding,

the original sentence is reconstructed. The main idea here is that the more accurate the sentence embedding is, the more accurate the reconstructed sentence will be.

Since we only need a bunch of sentences without any labels, training this model is straightforward. We start by defining our model as we did before:

```
# Create your embedding model
word_embedding_model = models.Transformer('bert-base-u
pooling_model = models.Pooling(word_embedding_model.ge
model = SentenceTransformer(modules=[word_embedding_mo
```

NOTE

Note that we are using CLS pooling instead of mean pooling. The authors found that there was little difference between them and since mean pooling loses positional information, they choose CLS pooling.

Next, we create a bunch of sentences to be used for our model using the STSB dataset that we used before:

```
from sentence_transformers.datasets import DenoisingAu

# Extract training data, we ignore the labels that we
train_data = datasets.load_dataset('glue', 'stsb', spl
train_data = train_data["sentence1"] + train_data["sen
```

```
# Add noise to our input data
train_dataset = DenoisingAutoEncoderDataset(list(set(t
train_dataloader = DataLoader(train_dataset, batch_siz

# Use the denoising auto-encoder loss
train_loss = losses.DenoisingAutoEncoderLoss(model, 'b
```

The main difference between this and how we performed supervised modeling before is that we pass our data to the `DenoisingAutoEncoderDataset`. This will generate noise to the input during training. We also defined our encoder loss using the same base model as we used for our embedding model.

Lastly, we only need to fit and evaluate our model:

```
# Train our embedding model
model.fit(train_objectives=[(train_dataloader, train_l

# Evaluate
evaluator(model)
```

After fitting our model, we get a score of 0.74 which is quite impressive considering we did all this training with unlabeled data.

Domain Adaptation

When you have very little or no labeled data available, you typically use unsupervised learning to create your text embedding model. However, unsupervised techniques are generally outperformed by supervised techniques and have difficulty learning domain-specific concepts.

This is where *domain adaptation* comes in. Its goal is to update existing embedding models to a specific textual domain that contains different subjects from the source domain. [Figure 7-14](#) demonstrates how domains can differ in content.

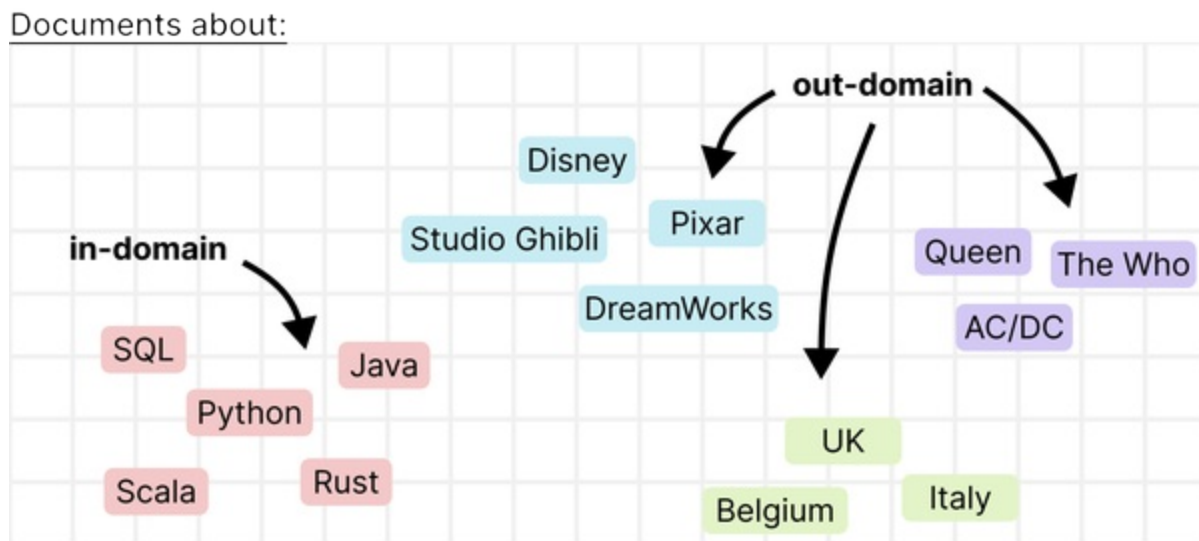


Figure 7-14. In domain adaptation, the aim is to create and generalize an embedding model from one domain to another. The target domain, or out-domain, generally contains words and subjects that were not found in the source domain or in-domain.

One method for domain adaptation is called *adaptive pre-*

training. You start by pre-training your domain-specific corpus using an unsupervised technique, such as the previously discussed TSDAE or even Masked Language Modeling. Then, as illustrated in [Figure 7-15](#), you fine-tune that model using a training dataset in your target domain.

This procedure leverages a pipeline that we have seen before, first using TSDAE to fine-tune an LLM or existing embedding model which is further fine-tuned using supervised or even augmented SBERT training.

This, however, can be computationally expensive since we must first pre-train our data on a large corpus and then use supervised learning with a labeled dataset. Typically, the labeled datasets need to be large and can require millions of training pairs.

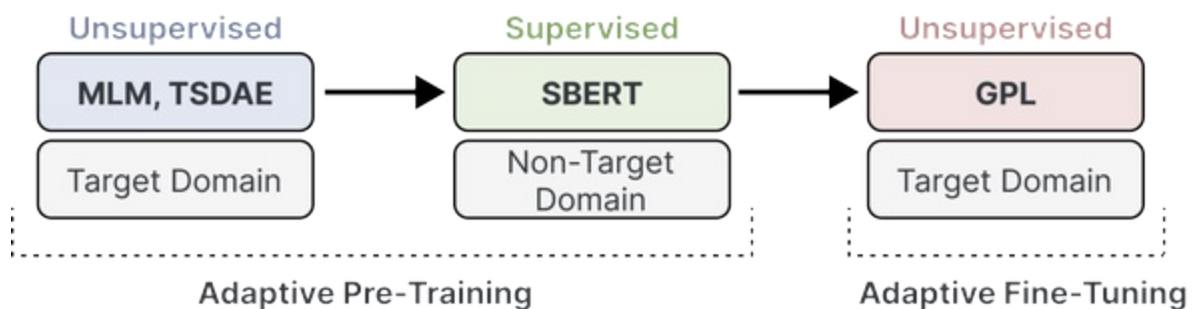


Figure 7-15. Domain adaptation can be performed with adaptive pre-training and adaptive fine-tuning. With adaptive pre-training, we first apply unsupervised tuning of an LLM on the target domain after which we fine-tune that model on labeled non-domain specific data. With adaptive fine-tuning, we take any existing pre-training embedding model and fine-tune it using unlabeled domain-specific data.

Instead, we are going to be using a method that can be run on top of a pre-trained embedding model instead, namely *Generative Pseudo-Labeling*.

Generative Pseudo-Labeling

Generative Pseudo-Labeling assumes that although we have data, none of it is labeled. It consists of three steps for generating labeled data that we can use for training an embedding model.

First, for each unlabeled text that you have in your domain-specific data, we use a generative model, like T5, to generate a number of queries. These queries are generally questions that can be answered with part(s) of the input text. For example, when your text is “Coco is a movie produced by Pixar”, the model might generate a query like “Who produced the movie Coco?”. These are the positive examples that we generate as we illustrate in [Figure 7-16](#).

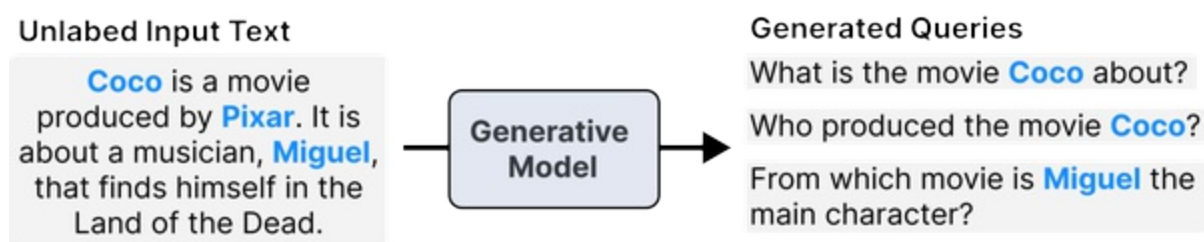


Figure 7-16. In the first step of GPL’s pipeline, queries are generated with a generative model, like T5, for each unlabeled input text. These queries can be used as sentence pairs for labeling later on in the GPL pipeline.

Second, we will also need to have negative examples for our model to learn. Preferably, a negative example is related to the query but is not the relevant answer. For example, a negative example for the query “Who produced the movie *Coco*?” would be “*The Lion King* was produced by Disney”.

To extract these negative examples, we use a pre-training embedding model to retrieve all texts that are relevant to the query. In [Figure 7-17](#), this second step shows how relevant texts are mined.

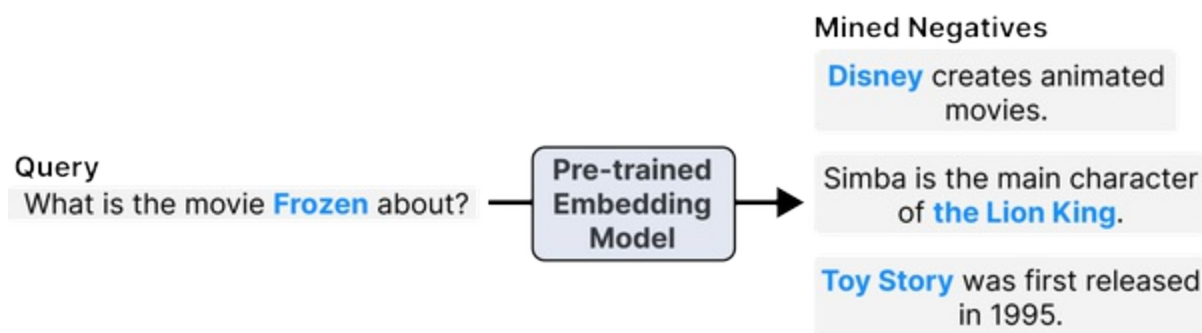


Figure 7-17. In the second step of GPL’s pipeline, negatives are mined using a pre-trained embedding model. This model generates embeddings for the input query and the corpus and finds those that relate to the input query.

Third, after we generate the negative examples, we will need to score them. Some of the negative examples might end up being the actual answer or they might not be relevant, both of which we want to prevent. As demonstrated in [Figure 7-18](#), we can use a pre-trained cross-encoder to score the query/passage pairs that

we have created through the previous two steps.

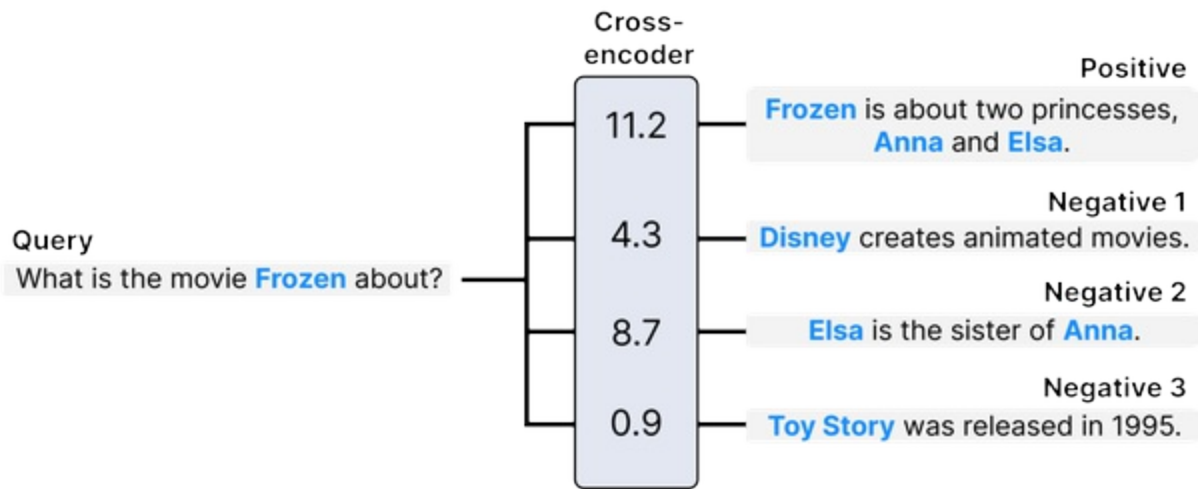


Figure 7-18. In the third step of GPL's pipeline, we use a cross-encoder to score all query/passage pairs. The goal of this procedure is to filter out any false negatives. Some negatives might accidentally turn out to be positives. Instead of a negative vs. positive labeling procedure, using a scoring procedure allows for more nuance in the representations.

After this final step, for each query, we now have a positive example (step 1) and a mined negative (step 2). In other words, we have triplets that we can use for our training procedure.

Running each step individually is quite a hassle but fortunately, there is a GPL package that abstracts that difficulty away.

However, we first need to format our data such that the package can read it.

```
import json
```

```
# Convert training data to the right format for GPL
```

```
train_data = datasets.load_dataset('glue', 'stsb', split=train_data_split)
with open('corpus.jsonl', 'w') as jsonl:
    for index, sentence in enumerate(train_data):
        line = {'_id': str(index), 'title': "", 'text': sentence[0]}
        jsonl.write(json.dumps(line)+'\n')
```

We are again using the STSB dataset and we extract just a small portion of the data for training. The reason for doing so is that training can still be quite expensive. The authors of GPL had to train for roughly a day on a V100 GPU, so trying out with a smaller dataset is advised before scaling up.

Next, we can import GPL and only need to run `train` to start:

```
import gpl

# Train an embedding model with GPL
gpl.train(
    path_to_generated_data=".",
    batch_size_gpl=8,
    batch_size_generation=8,
    gpl_steps=10000,
    queries_per_passage=1,
    output_dir="output",
    do_evaluation=False,
    use_amp=True)
```

```
# The model that we will fine-tune as our embeddin
base_ckpt="distilbert-base-uncased",

# The model used to generate queries
generator="BeIR/query-gen-msmarco-t5-base-v1",

# The model used for mining negatives
retrievers=["msmarco-distilbert-base-v3", "msmarco

# The model used for rating query/passage pairs
cross_encoder="cross-encoder/ms-marco-MiniLM-L-6-v
)
```

There are a few sub-models worth noting that describe the steps as we have seen them before

- `generator` refers to *step 1* where we generate queries for our data
- `retriever` refers to *step 2* where we generate queries for our data
- `cross_encoder` refers to *step 3* where we generate queries for our data
- `base_ckpt` refers to the final step of training our embedding model

After training, we can load and evaluate the model as follows:

```
>>> # Load our new model and evaluate it
>>> model = SentenceTransformer('output')
>>> evaluator(model)

0.8246360833250379
```

With GPL, we managed to get a score of 0.82 without any labeled data at all!

Summary

In this chapter, we have taken a look at creating and fine-tuning embedding models through contrastive learning, one of the most important components of training such models. Through both unsupervised and supervised techniques, we were able to create embedding models tuned to our datasets.

¹ Alan Garfinkel. “Forms of Explanation: Rethinking the Questions in Social Theory.” (1981).

² Tim Miller. “Contrastive explanation: A structural-model approach”.*The Knowledge Engineering Review* 36 (2021): e14.

3 Jeffrey Pennington, Richard Socher, and Christopher D, Manning. “Glove: Global vectors for word representation.” In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532–1543). 2014.

4 Muennighoff, Niklas, Nouamane, Tazi, Loïc, Magne, and Nils, Reimers. “MTEB: Massive text embedding benchmark”.*arXiv preprint arXiv:2210.07316* (2022).

5 Matthew Henderson, Rami, Al-Rfou, Brian, Strope, Yun-Hsuan, Sung, László, Lukacs, Ruiqi, Guo, Sanjiv, Kumar, Balint, Miklos, and Ray, Kurzweil. “Efficient natural language response suggestion for smart reply.” *arXiv preprint arXiv:1705.00652* (2017).

6 Oord, Aaron van den, Yazhe, Li, and Oriol, Vinyals. “Representation learning with contrastive predictive coding”.*arXiv preprint arXiv:1807.03748* (2018).

7 Ting Chn, Simon, Kornblith, Mohammad, Norouzi, and Geoffrey, Hinton. “A simple framework for contrastive learning of visual representations.” . In *International conference on machine learning* (pp. 1597–1607).2020.

8 Thakur, Nandan, Nils, Reimers, Johannes, Daxenberger, and Iryna, Gurevych. “Augmented sbert: Data augmentation method

for improving bi-encoders for pairwise sentence scoring tasks”.*arXiv preprint arXiv:2010.08240* (2020).

[9](#) Gao, Tianyu, Xingcheng, Yao, and Danqi, Chen. “Simcse: Simple contrastive learning of sentence embeddings”.*arXiv preprint arXiv:2104.08821* (2021).

[10](#) Janson, Sverker, Evangelina, Gogoulou, Erik, Ylipää, Amaru, Cuba Gyllensten, and Magnus, Sahlgren. “Semantic re-tuning with contrastive tension.” In *International Conference on Learning Representations, 2021*.2021.

[11](#) Kexin Wang, Nils, Reimers, and Iryna, Gurevych. “Tsdae: Using transformer-based sequential denoising auto-encoder for unsupervised sentence embedding learning”.*arXiv preprint arXiv:2104.06979* (2021).

[12](#) Kexin Wang, Nandan, Thakur, Nils, Reimers, and Iryna, Gurevych. “Gpl: Generative pseudo labeling for unsupervised domain adaptation of dense retrieval”.*arXiv preprint arXiv:2112.07577* (2021).

About the Authors

Jay Alamm is Director and Engineering Fellow at Cohere (pioneering provider of large language models as an API). In this role, he advises and educates enterprises and the developer community on using language models for practical use cases). Through his popular AI/ML blog, Jay has helped millions of researchers and engineers visually understand machine learning tools and concepts from the basic (ending up in the documentation of packages like NumPy and pandas) to the cutting-edge (Transformers, BERT, GPT-3, Stable Diffusion). Jay is also a co-creator of popular machine learning and natural language processing courses on DeepLearning.ai and Udacity.

Maarten Grootendorst is a Senior Clinical Data Scientist at IKNL (Netherlands Comprehensive Cancer Organization). He holds master's degrees in organizational psychology, clinical psychology, and data science which he leverages to communicate complex Machine Learning concepts to a wide audience. With his popular blogs, he has reached millions of readers by explaining the fundamentals of Artificial Intelligence—often from a psychological point of view. He is the author and maintainer of several open-source packages that rely on the strength of Large Language Models, such as BERTopic, PolyFuzz, and KeyBERT. His

packages are downloaded millions of times and used by data professionals and organizations worldwide.