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 Alammar and Maarten Grootendorst

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Hands-On Large Language Models

by Jay Alammar and Maarten Grootendorst

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

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

978-1-098-15090-7

[TO COME]

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.

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.

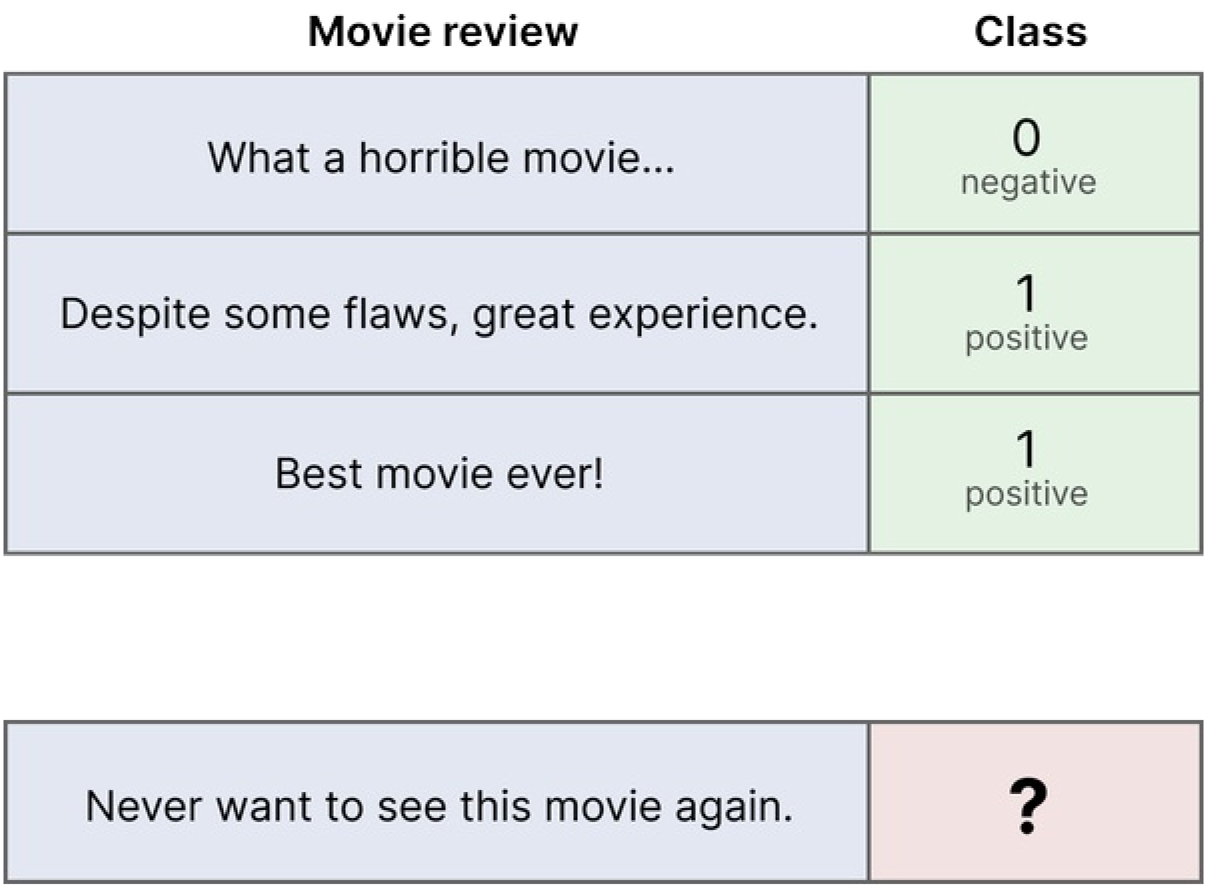

Figure 1-1. An example of supervised classification. Can we predict whether a movie review is either positive or negative?

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.

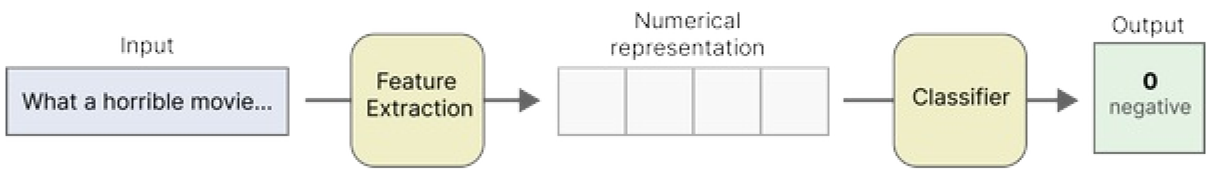

Figure 1-2. A common procedure for supervised text classification. We convert our textual input data to numerical representations through feature extraction. Then, a classifier is trained to predict labels.

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, architecture, 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

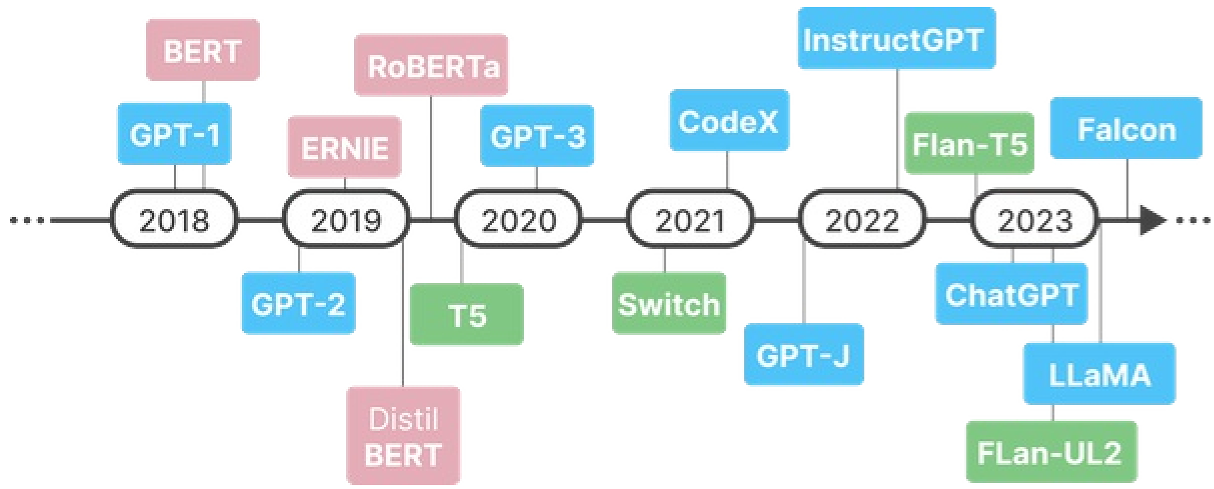

Figure 1-3. A timeline of common Large Language Model releases.

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.

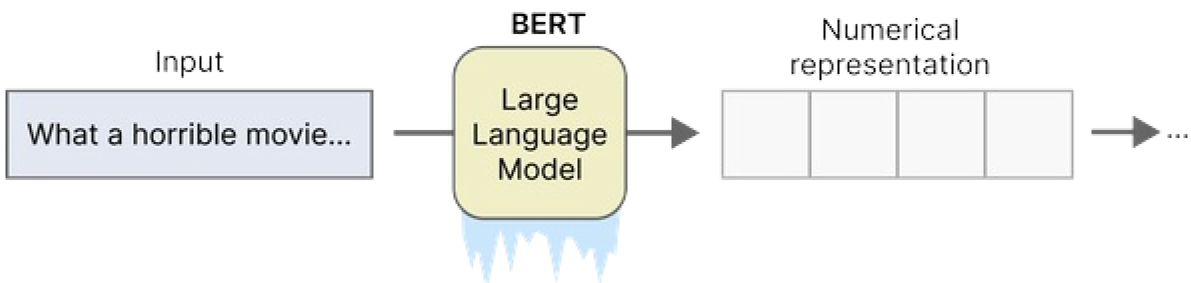

Figure 1-4. First, we start by using a generic pre-trained LLM (e.g., BERT) to convert our textual data into more numerical representations. During training, we will "freeze" the model such that its weights will not be updated. This speeds up training significantly but is generally less accurate.

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.

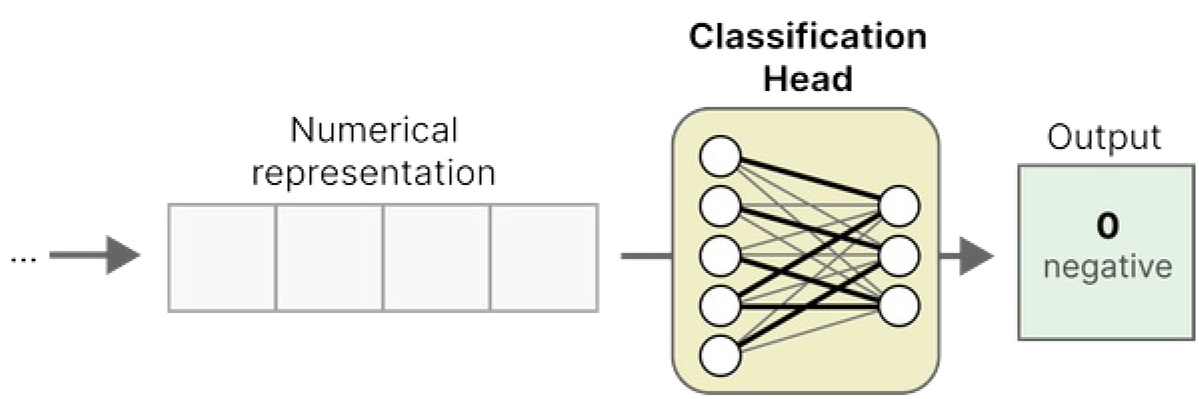

Figure 1-5. After fine-tuning our LLM, we train a classifier on the numerical representations and labels. Typically, a Feed Forward Neural Network is chosen as the classifier.

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.

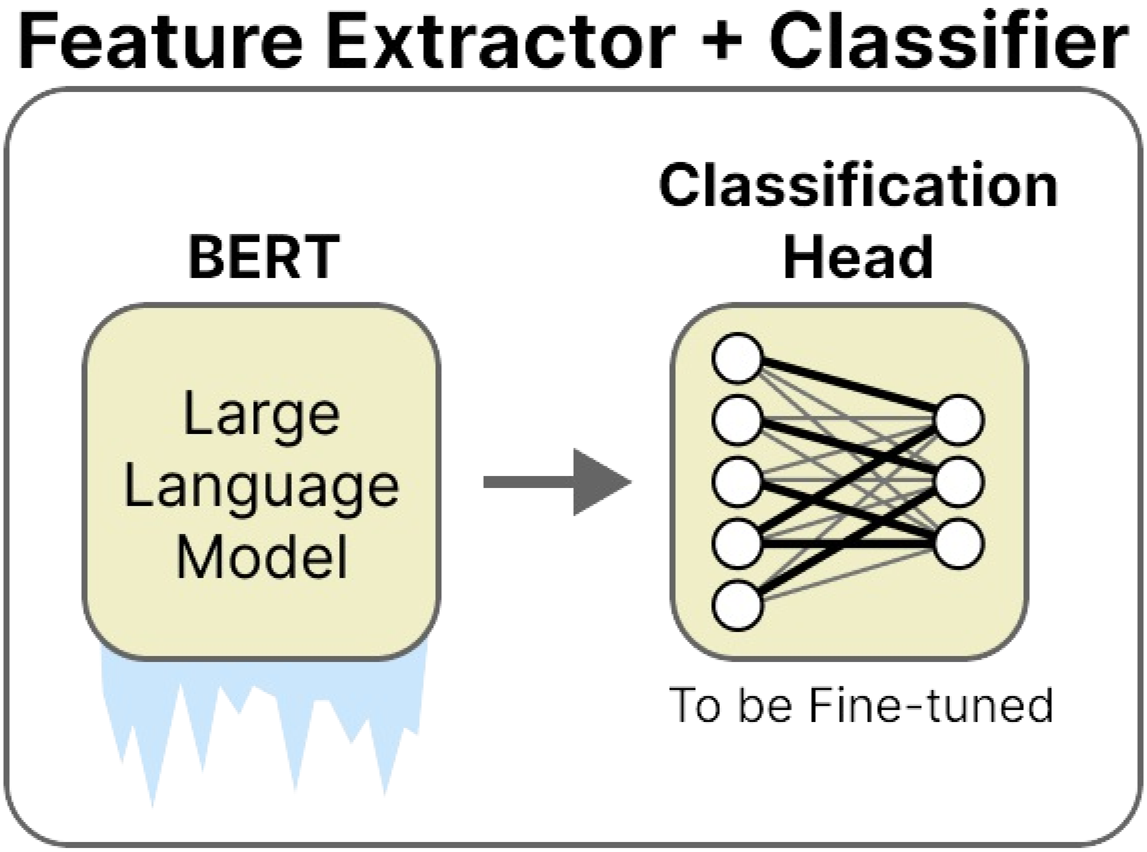

Figure 1-6. We adopt the BERT model such that its output embeddings are fed into a classification head. This head generally consists of a linear layer but might include dropout beforehand.

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_repor
>>> 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.

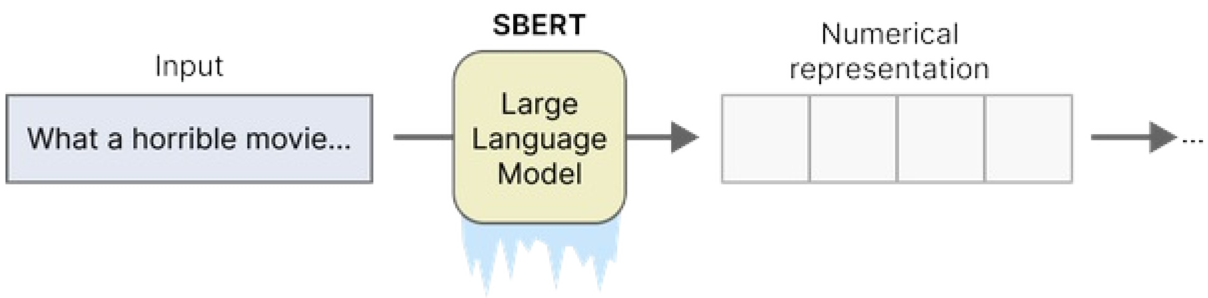

Figure 1-7. First, we use an LLM that was trained specifically to generate accurate numerical representations. These tend to be better representative vectors than we receive from a general Transformer-based model like BERT.

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.

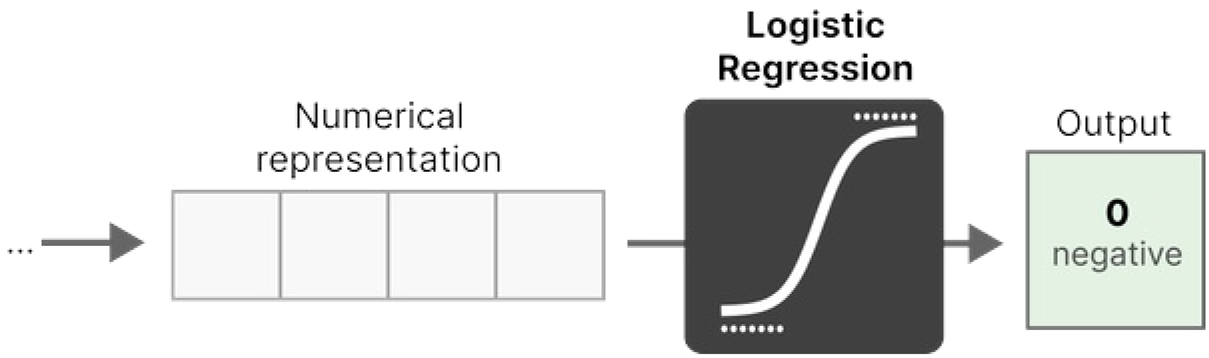

Figure 1-8. Using the embeddings as our features, we train a logistic regression model on our training data.

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.

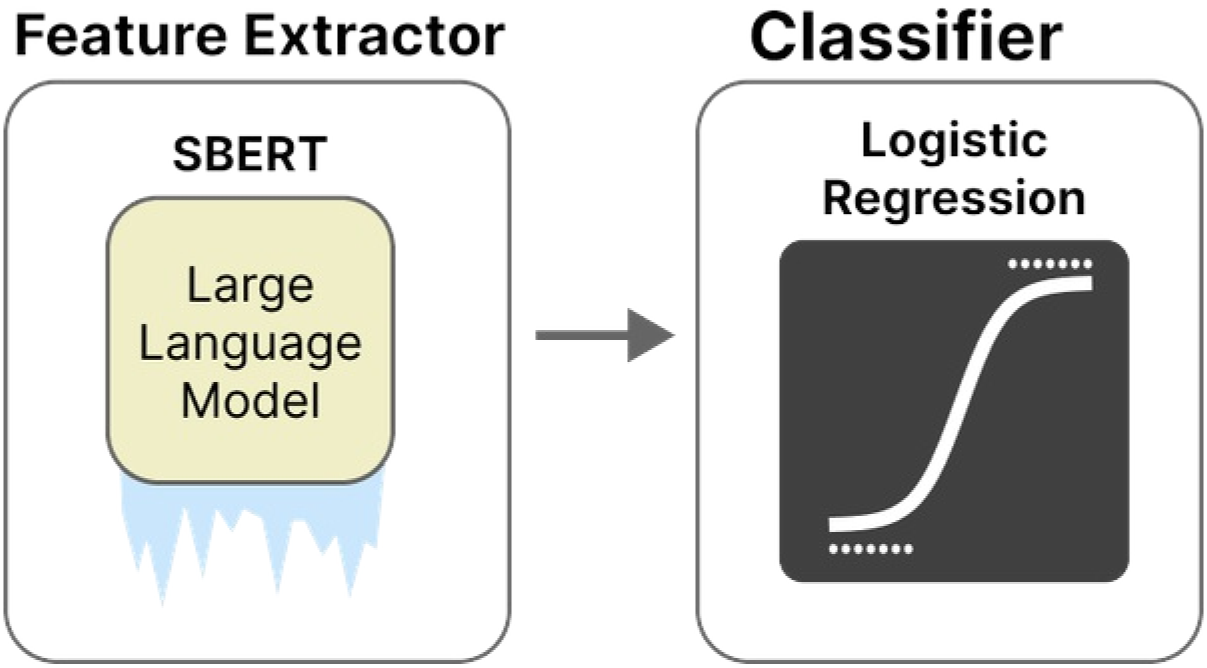

Figure 1-9. The classifier is a separate model that leverages the embeddings from SBERT to learn from.

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_repor
>>> 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.

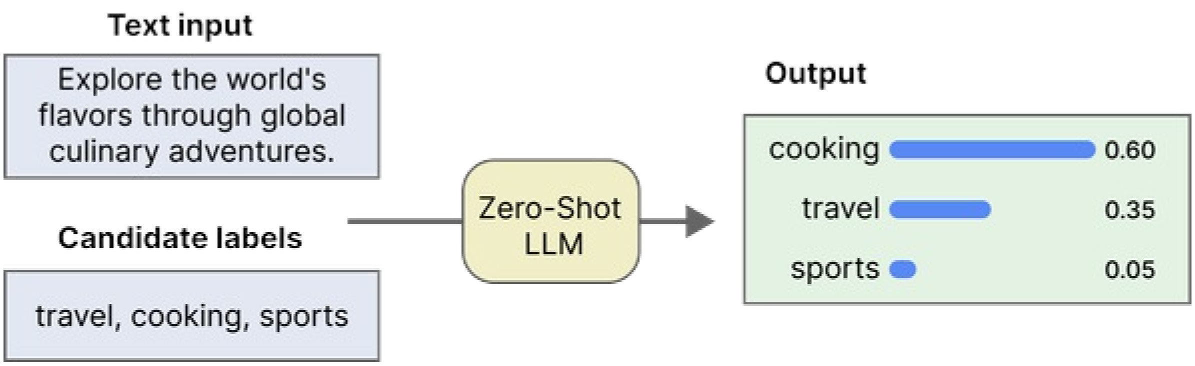

Figure 1-10. Figure 2-11. In zero-shot classification, the LLM is not trained on any of the candidate labels. It learned from different labels and generalized that information to the candidate labels.

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.

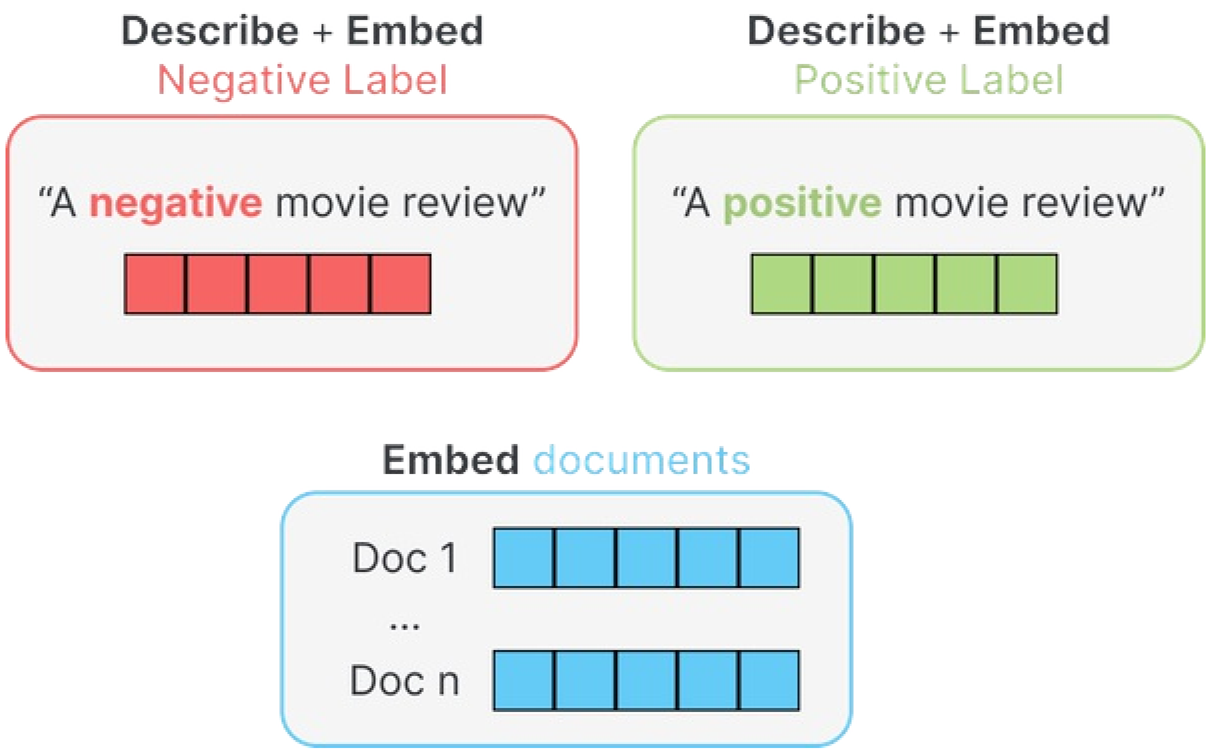

Figure 1-11. To embed the labels, we first need to give them a description. For example, the description of a negative label could be “A negative movie review”. This description can then be embedded through sentence-transformers. In the end, both labels as well as all the documents are embedded.

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.

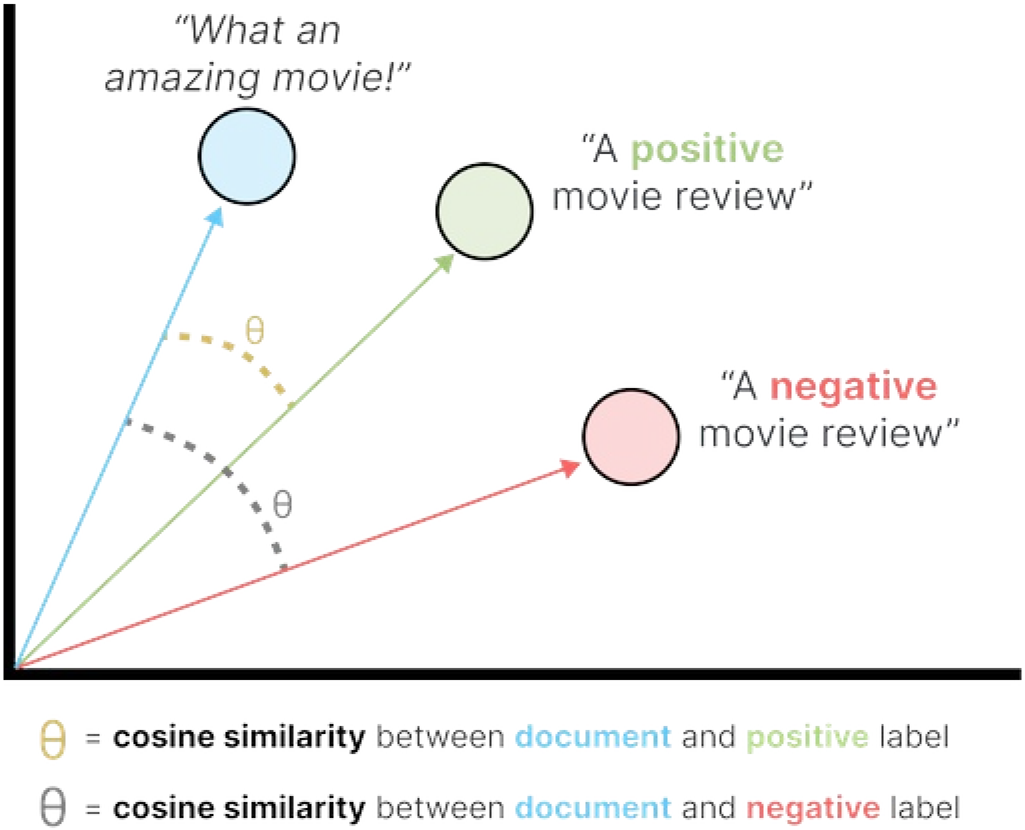

Figure 1-12. The cosine similarity is the angle between two vectors or embeddings. In this example, we calculate the similarity between a document and the two possible labels, positive and negative.

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.

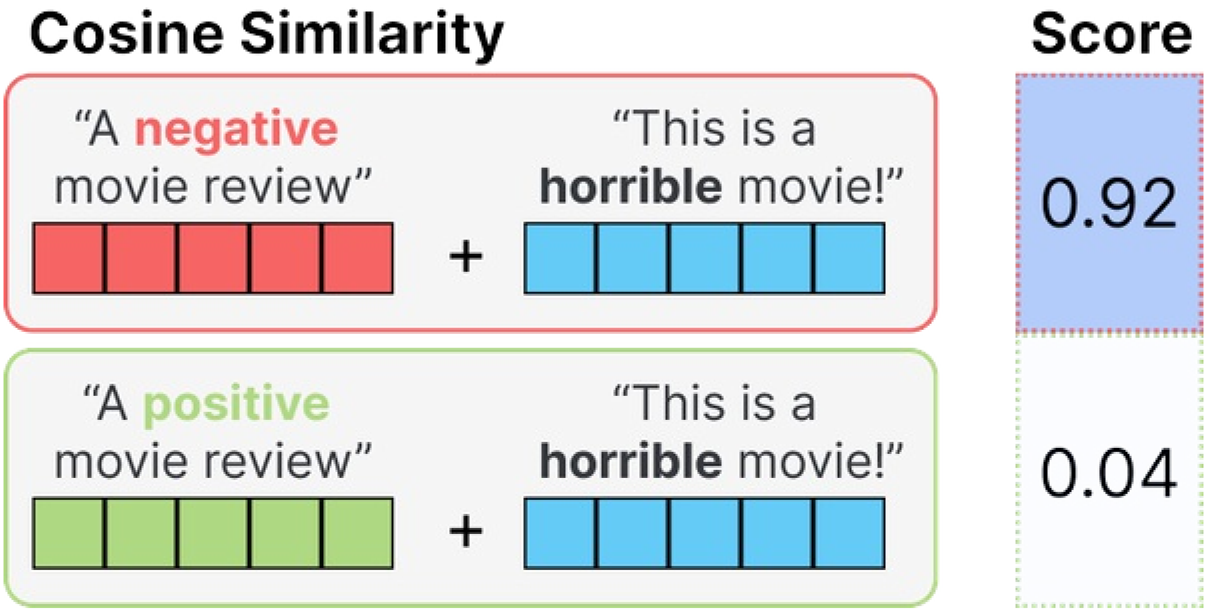

Figure 1-13. After embedding the label descriptions and the documents, we can use cosine similarity for each label document pair. For each document, the label with the highest similarity to the document is chosen.

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_
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.

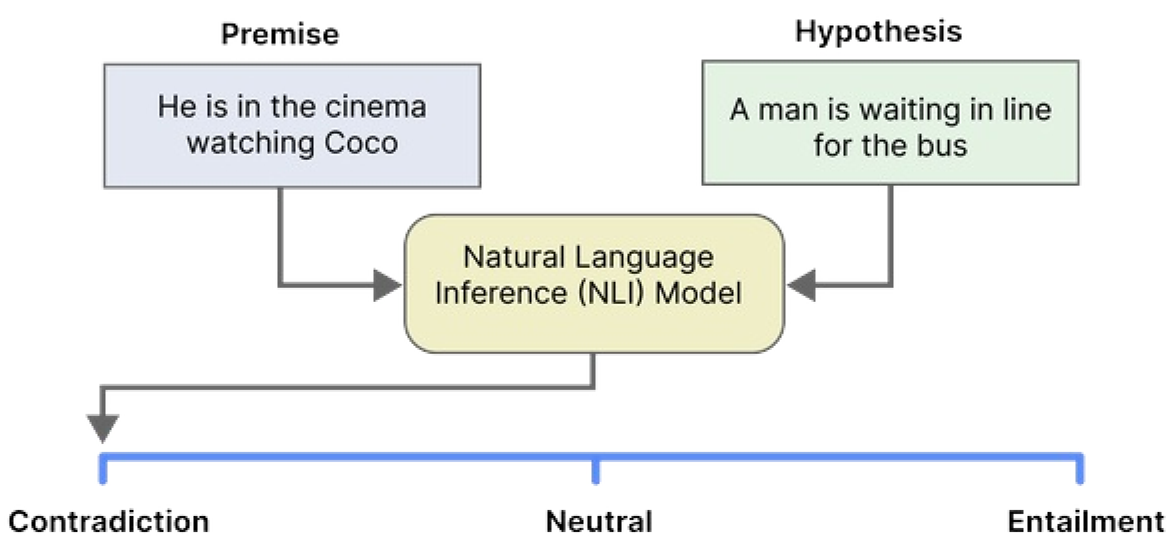

Figure 1-14. An example of natural language inference (NLI). The hypothesis is contradicted by the premise and is not relevant to one another.

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.

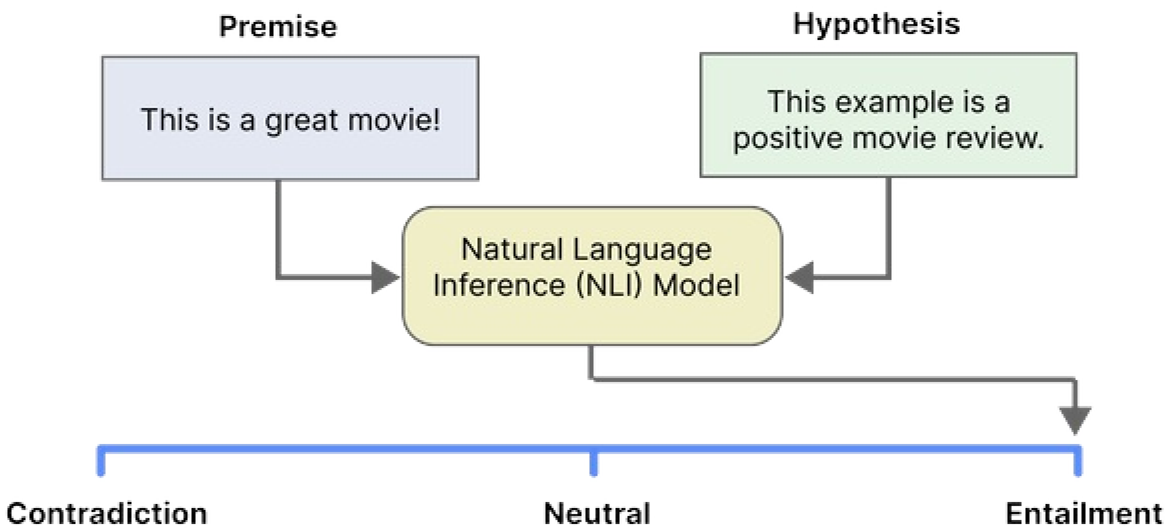

Figure 1-15. An example of zero-shot classification with natural language inference (NLI). The hypothesis is supported by the premise and the model will return that the review is indeed a positive movie review.

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_repor
>>> 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.

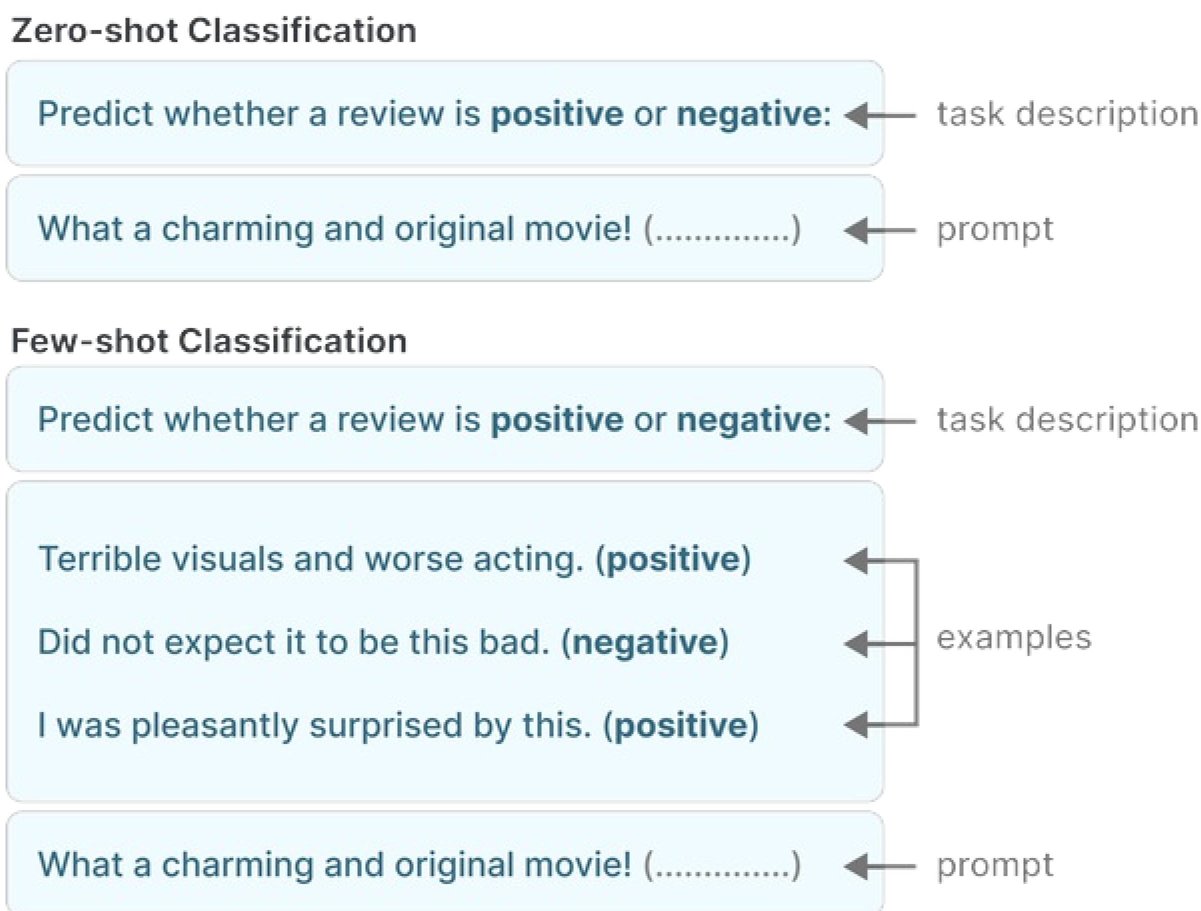

Figure 1-16. Zero-shot and few-shot classification through prompt engineering with generative models.

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.

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))
```

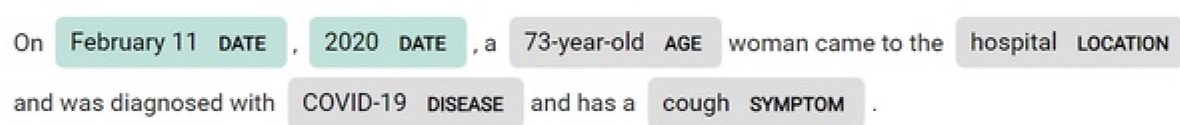

Figure 1-18. The output of SpaCy using OpenAI's GPT-3.5 model. Without any training, it correctly identifies our custom entities.

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.

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.

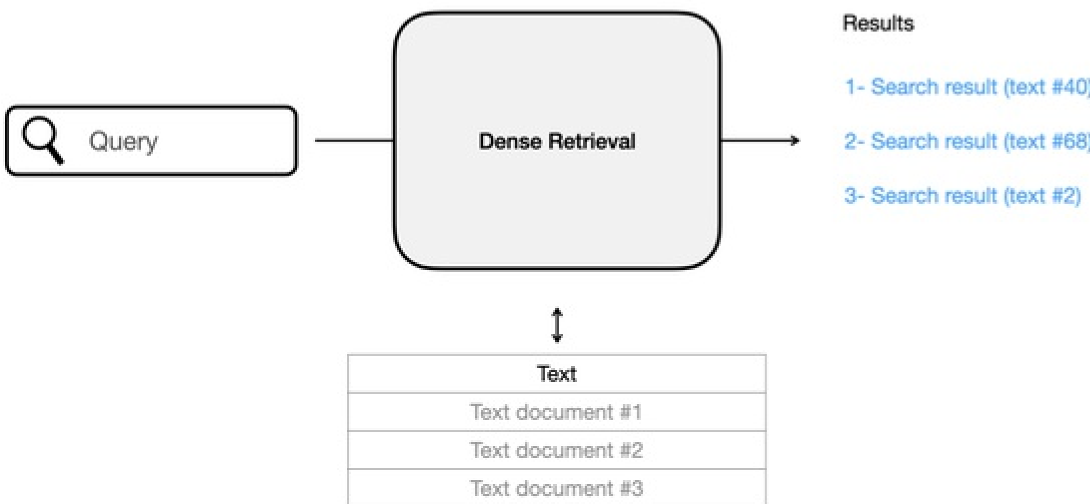

Figure 2-1. Dense retrieval is one of the key types of semantic search, relying on the similarity of text embeddings to retrieve relevant results

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.

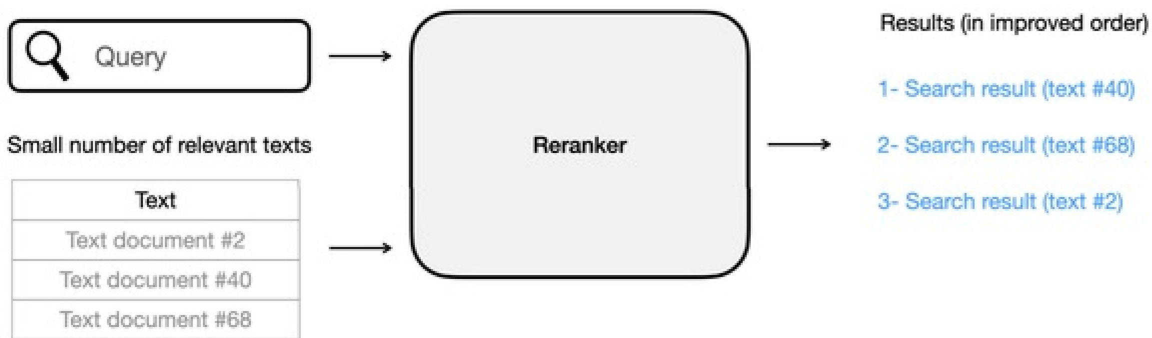

Figure 2-2. Rerankers, the second key type of semantic search, take a search query and a collection of results, and re-order them by relevance, often resulting in vastly improved results.

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.

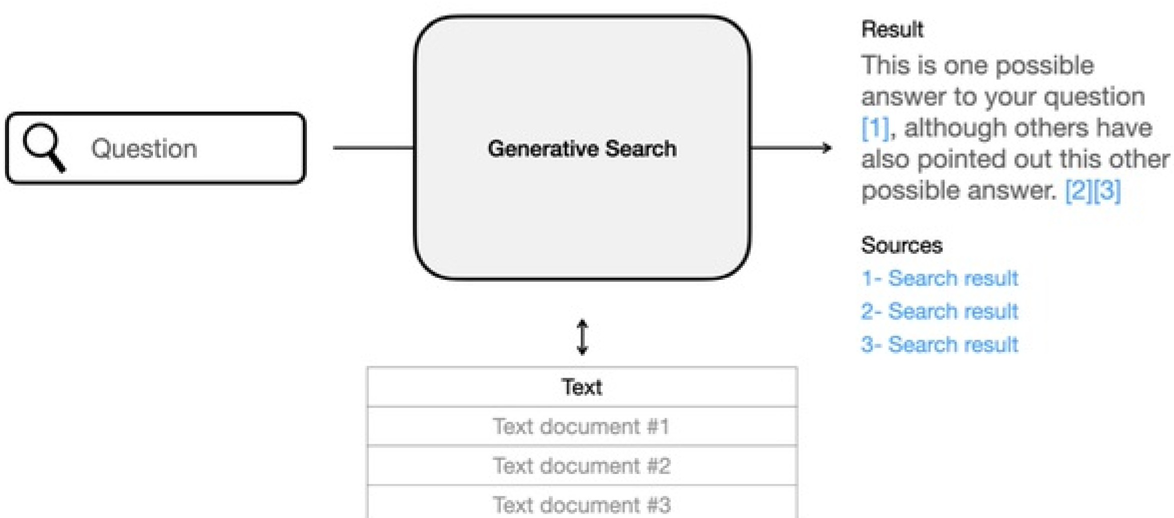

Figure 2-3. Generative search formulates an answer to a question and cites its information sources.

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).

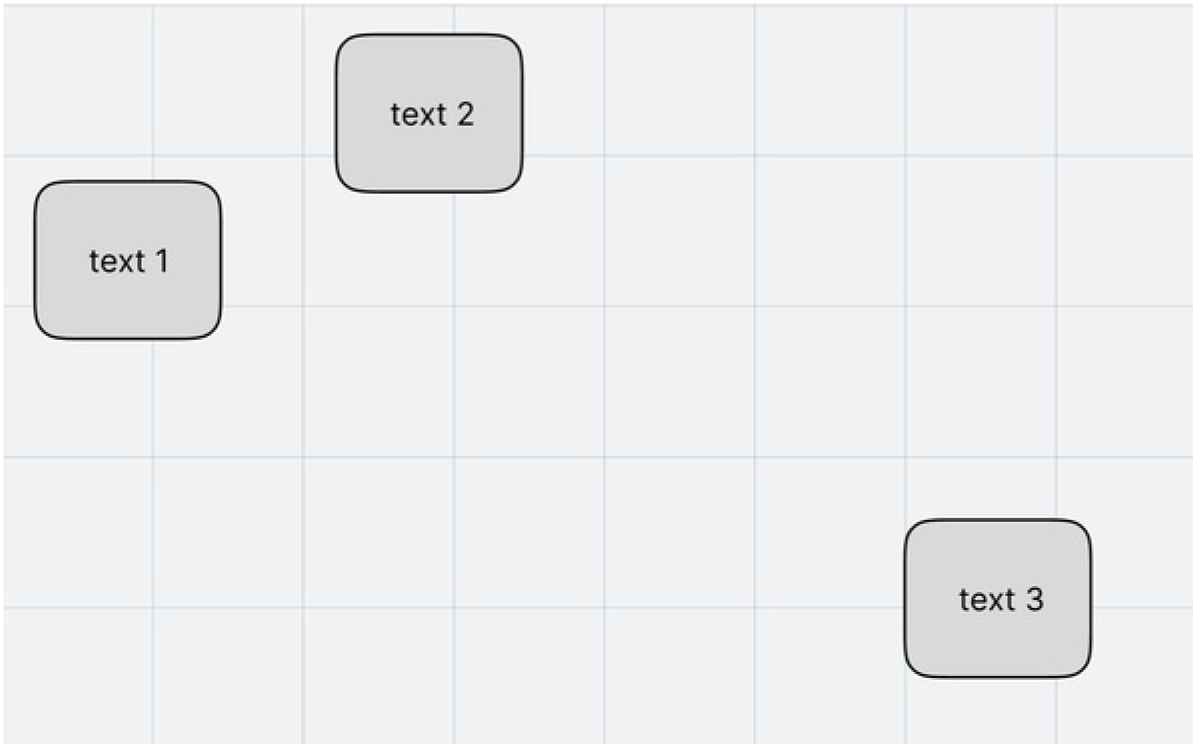

Figure 2-4. The intuition of embeddings: each text is a point, texts with similar meaning are close to each other.

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.

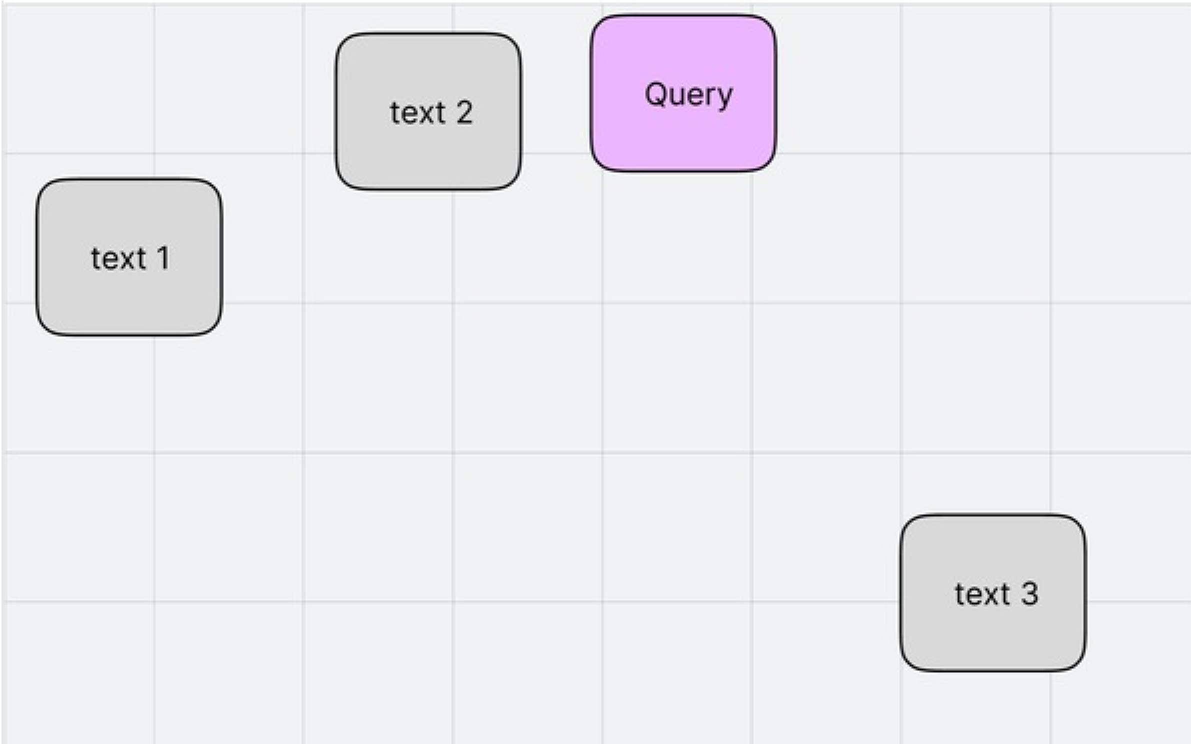

Figure 2-5. Dense retrieval relies on the property that search queries will be close to their relevant results.

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 publi
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). 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 laureat
   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 embeddin
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(
                                                    in
    # 3. Format the results
    results = pd.DataFrame(data={'texts': texts[simila
                                'distance': similar_it


    # 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.

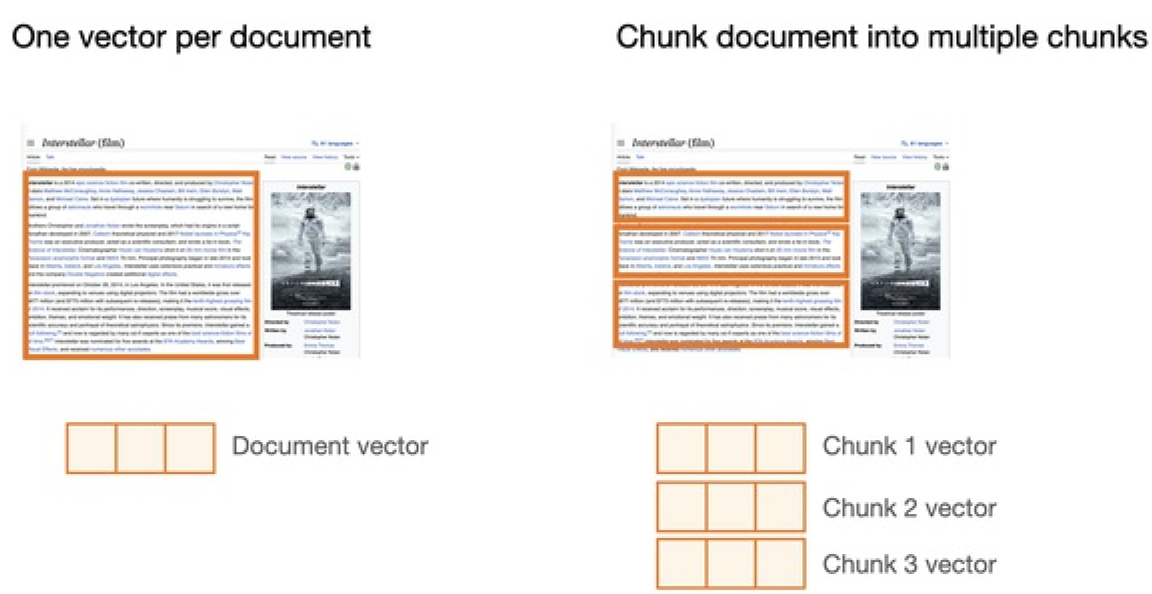

Figure 2-6. It’s possible to create one vector representing an entire document, but it’s better for longer documents to be split into smaller chunks that get their own embeddings.

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.

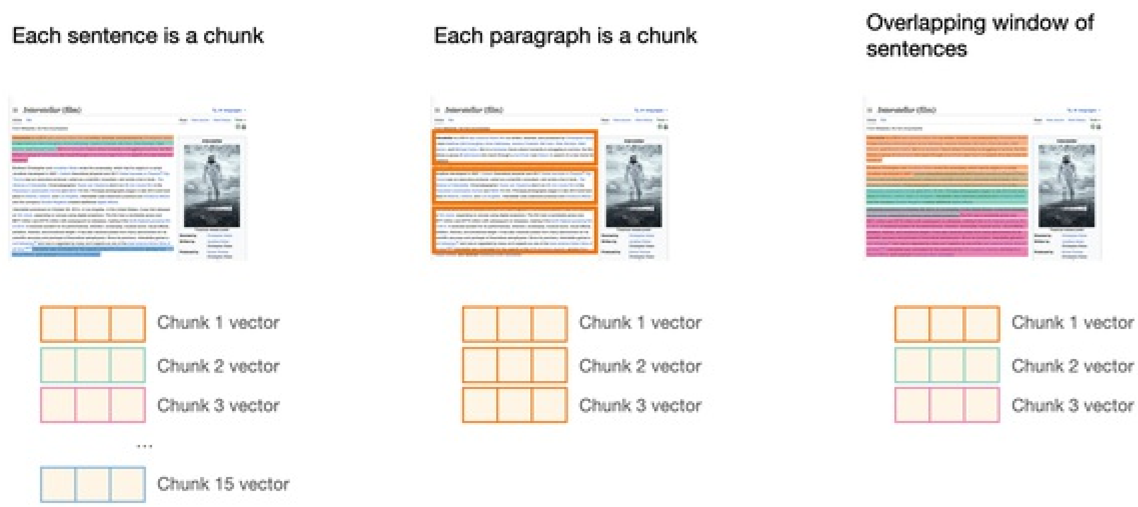

Figure 2-7. A number of possible options for chunking a document for embedding.

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.

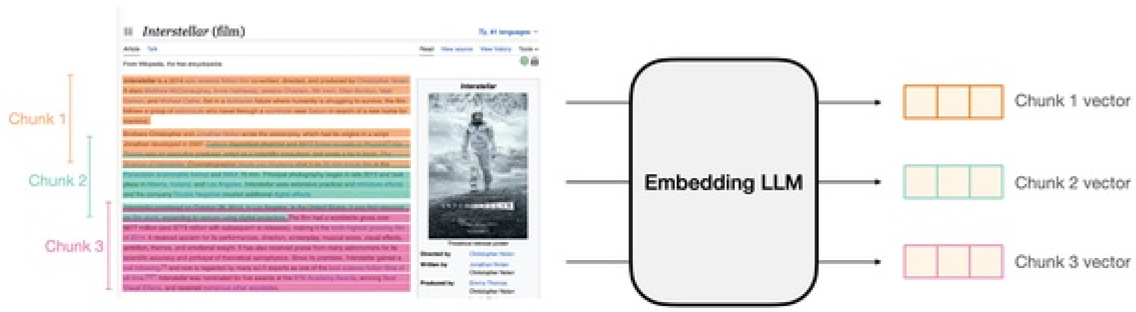

Figure 2-8. Chunking the text into overlapping segments is one strategy to retain more of the context around different segments.

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.

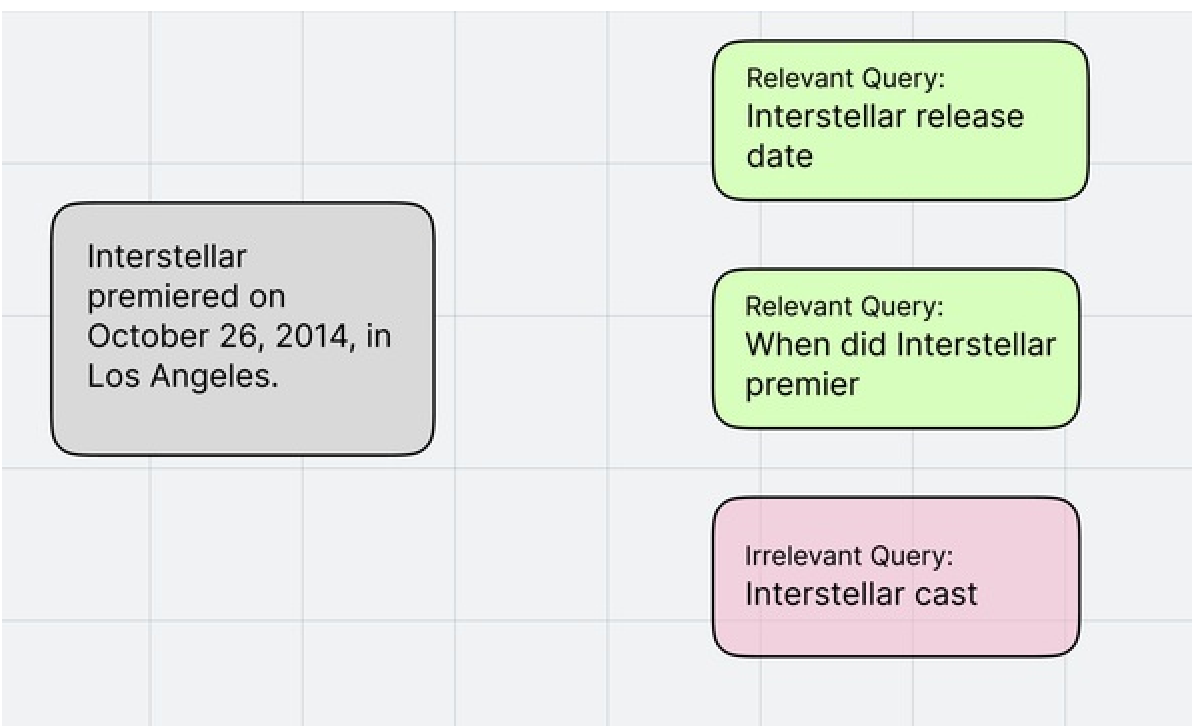

Figure 2-9. Before fine-tuning, the embeddings of both relevant and irrelevant queries may be close to a particular document.

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.

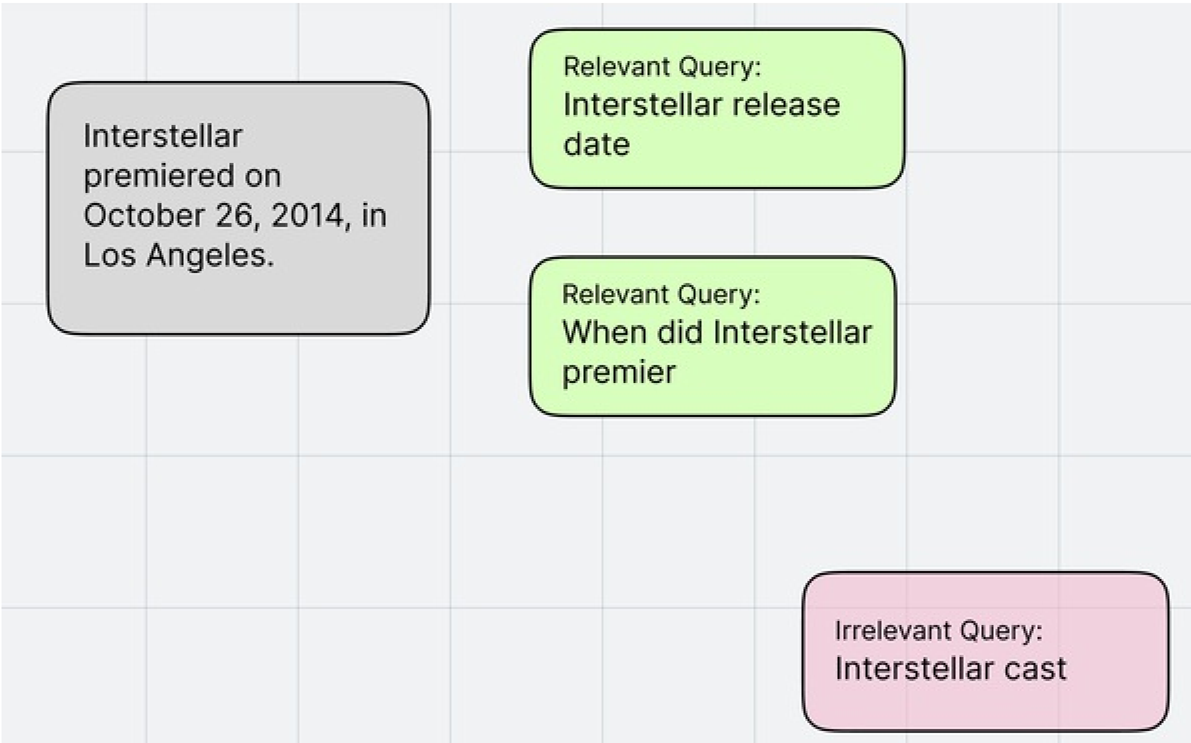

Figure 2-10. After the fine-tuning process, the text embedding model becomes better at this search task by incorporating how we define relevance on our dataset using the examples we provided of relevant and irrelevant documents.

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.

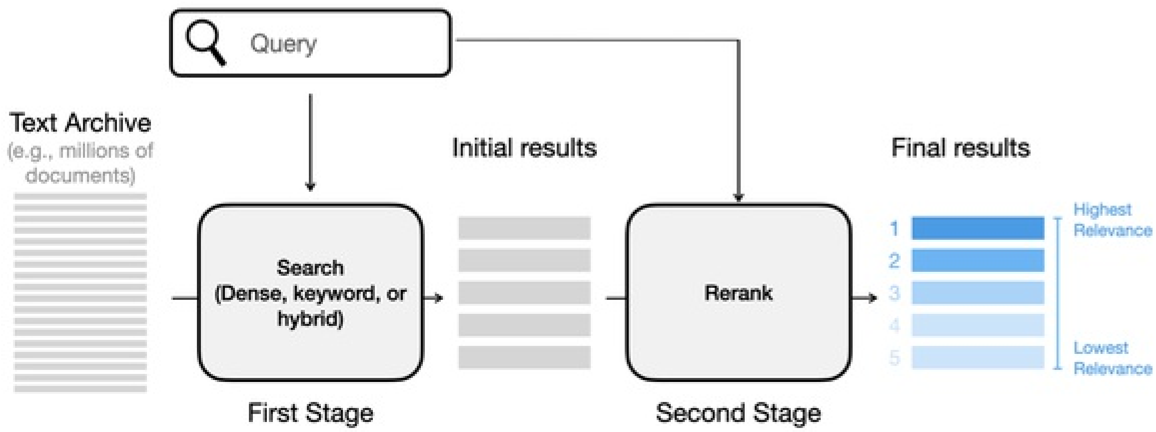

Figure 2-11. LLM Rerankers operate as a part of a search pipeline with the goal of re-ordering a number of shortlisted search results by relevance

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.

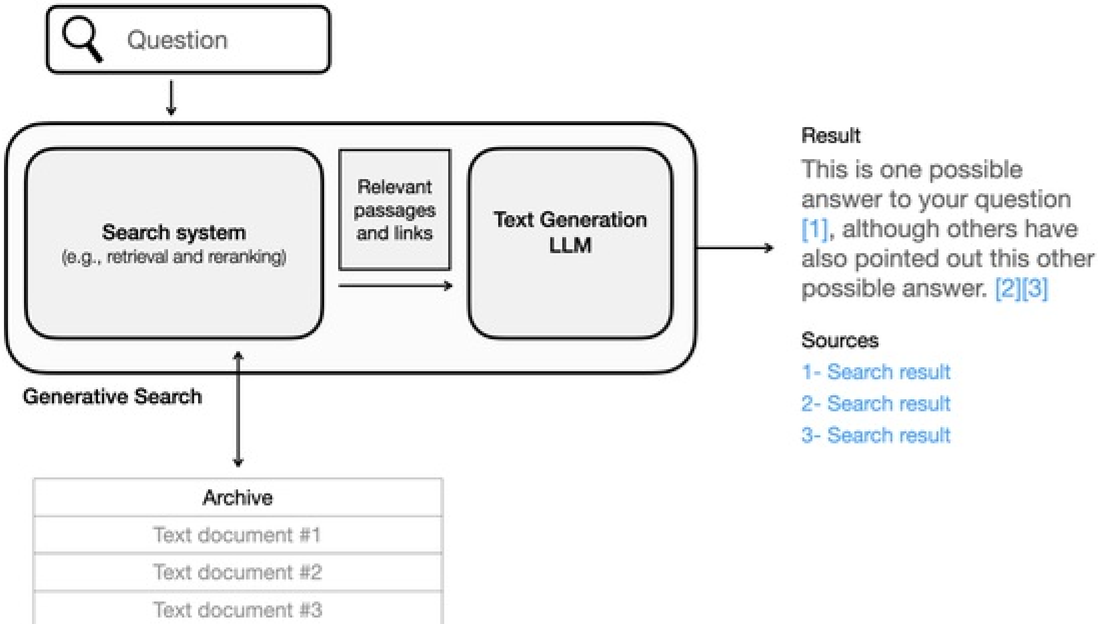

Figure 2-12. Generative search formulates answers and summaries at the end of a search pipeline while citing its sources (returned by the previous steps in the search system).

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.

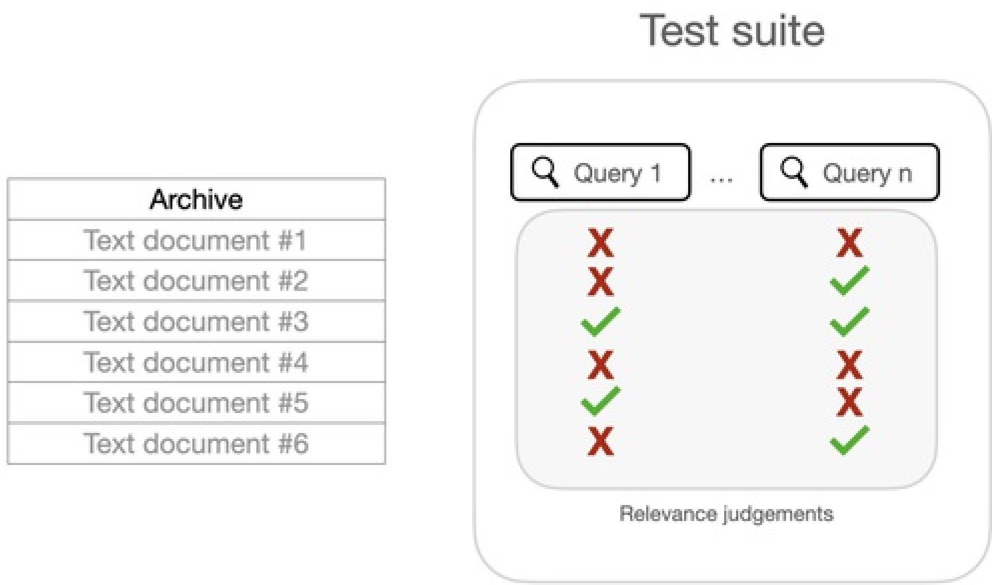

Figure 2-13. To evaluate search systems, we need a test suite including queries and relevance judgements indicating which documents in our archive are relevant for each query.

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.

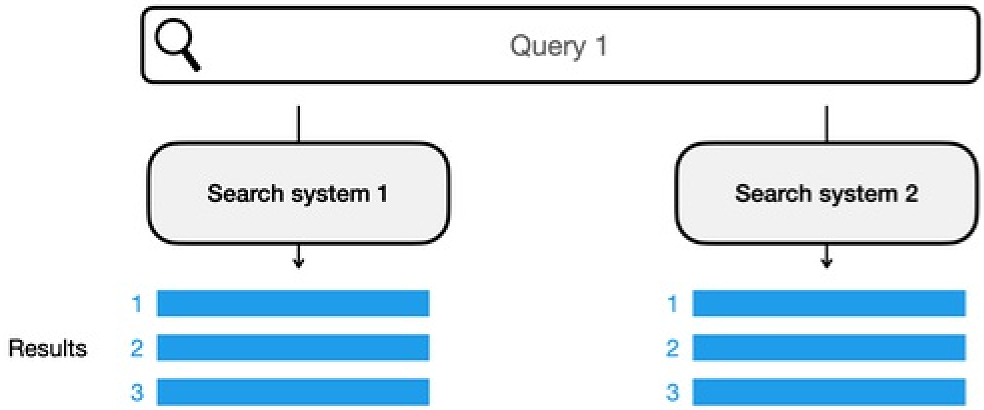

Figure 2-14. To compare two search systems, we pass the same query from our test suite to both systems and look at their top results

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 Figure3-16 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

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.

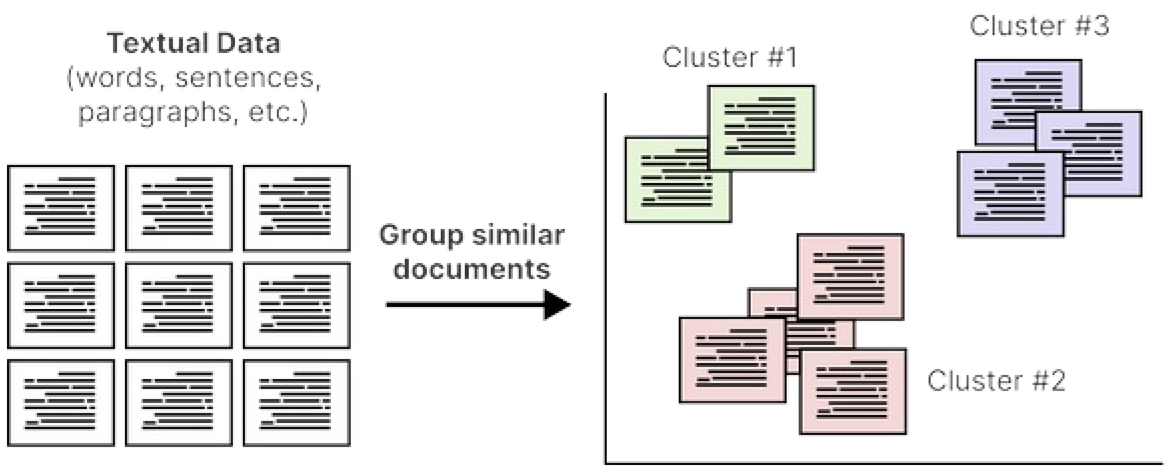

Figure 3-1. Clustering unstructured textual data.

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.

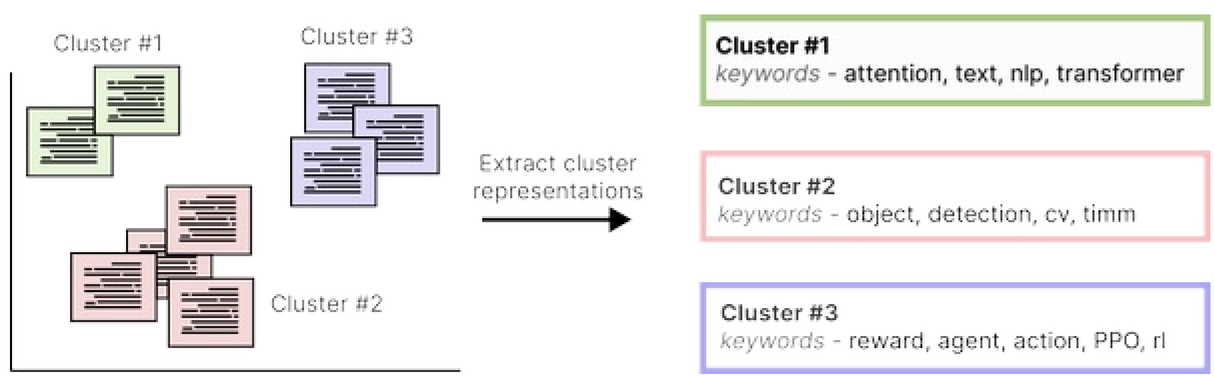

Figure 3-2. Topic modeling is a way to give meaning to clusters of textual documents.

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.

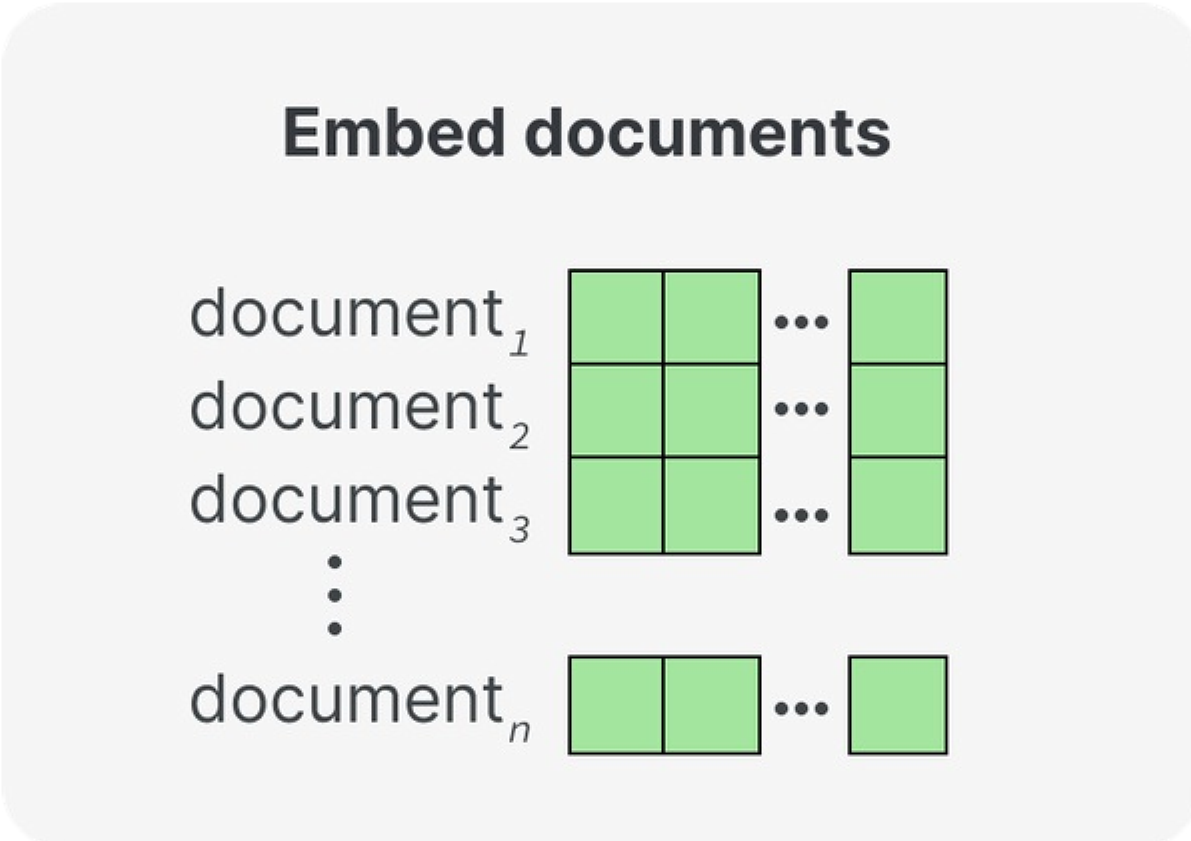

Figure 3-3. Step 1: We convert documents to numerical representations, namely embeddings.

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-v

# The abstracts are converted to vector representation
embeddings = 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.

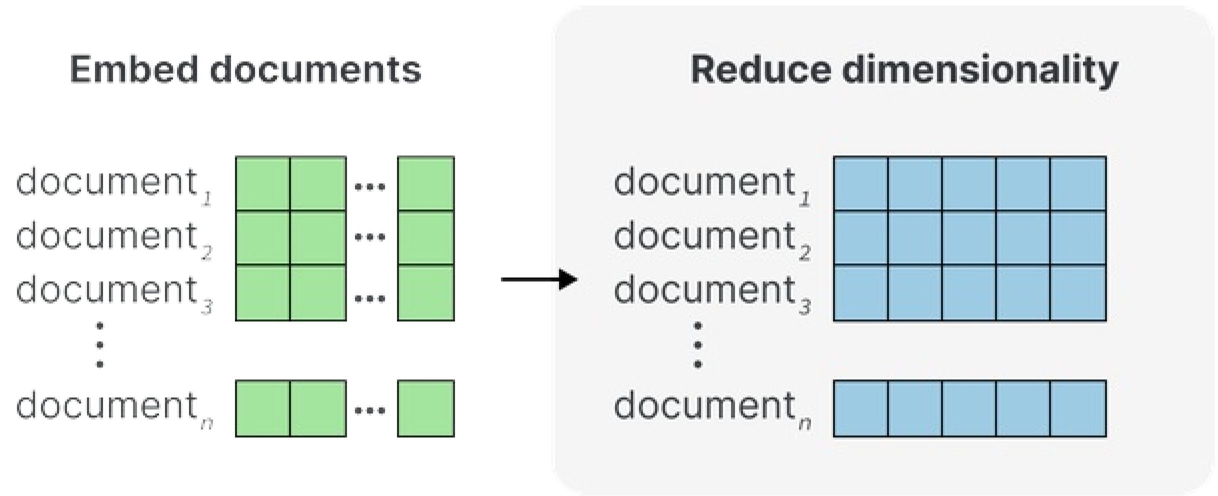

Figure 3-4. Step 2: The embeddings are reduced to a lower dimensional space using dimensionality reduction.

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.

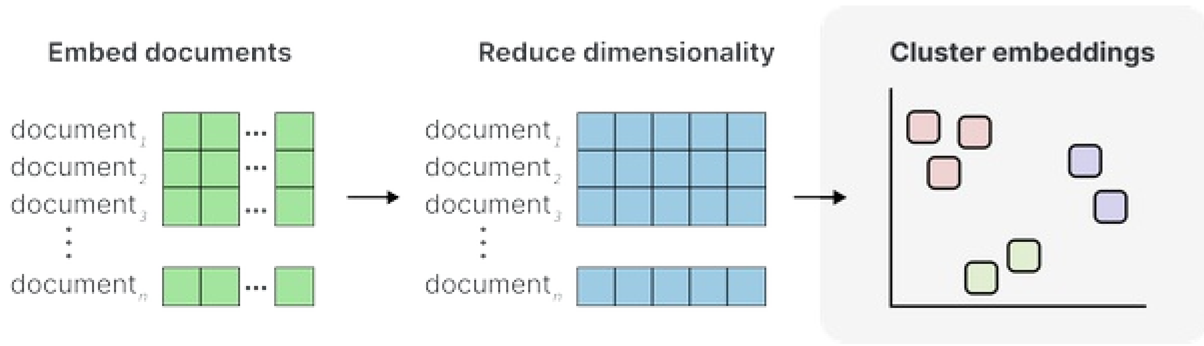

Figure 3-5. Step 3: We cluster the documents using the embeddings that were reduced in their dimensionality.

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.

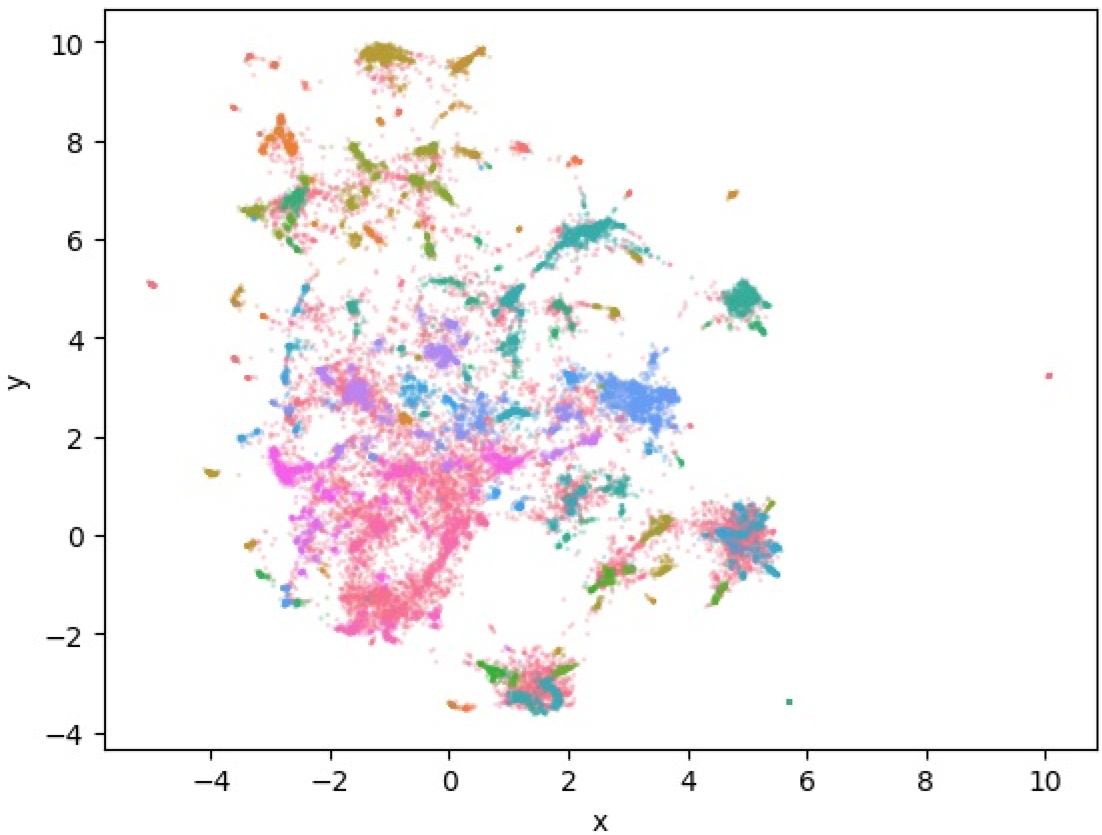

Figure 3-6. The generated clusters (colored) and outliers (grey) are represented as a 2D visualization.

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 sar
work has emphasized the need for models to capitalize 
beyond lexical and syntactic cues present in utterance
speakers will tend to employ sarcasm regarding differe
```

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".

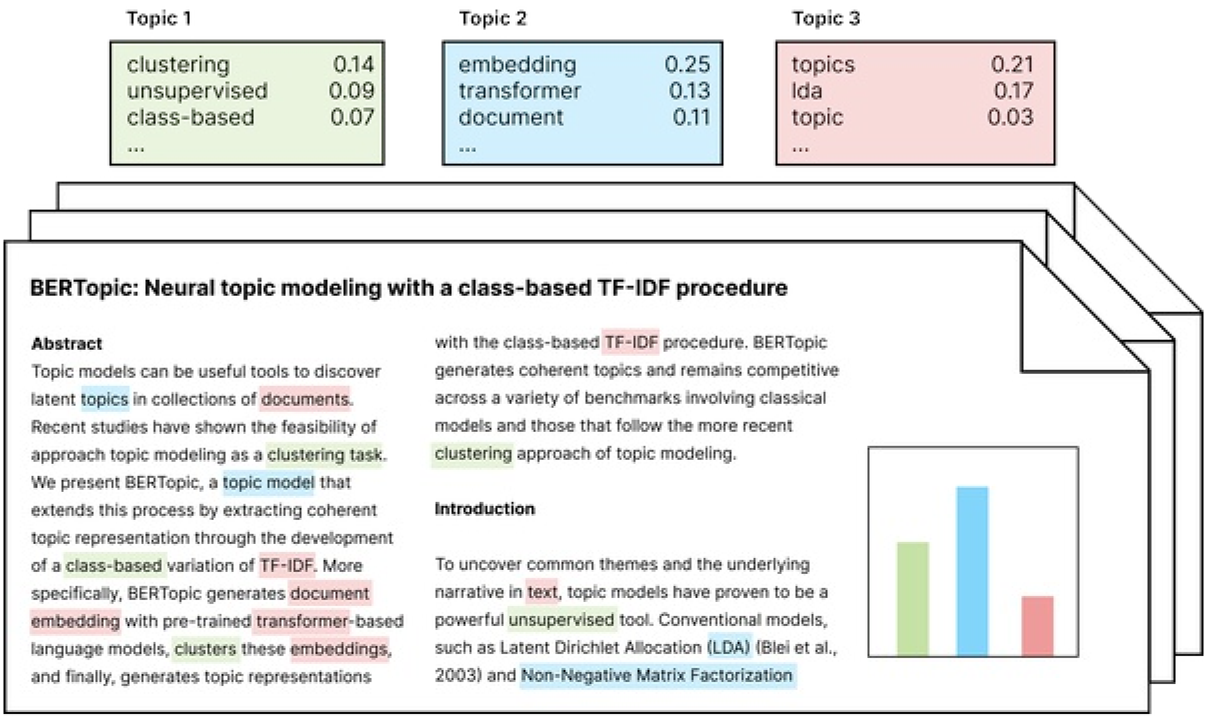

Figure 3-7. An overview of traditional topic modeling.

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.

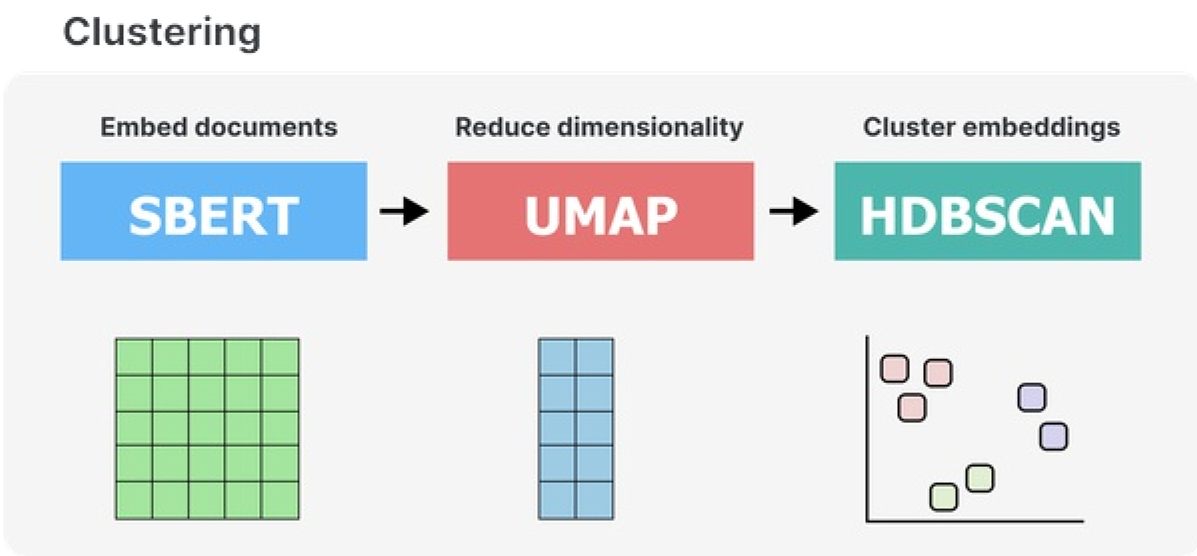

Figure 3-8. The first part of BERTopic's pipeline is clustering textual data.

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.

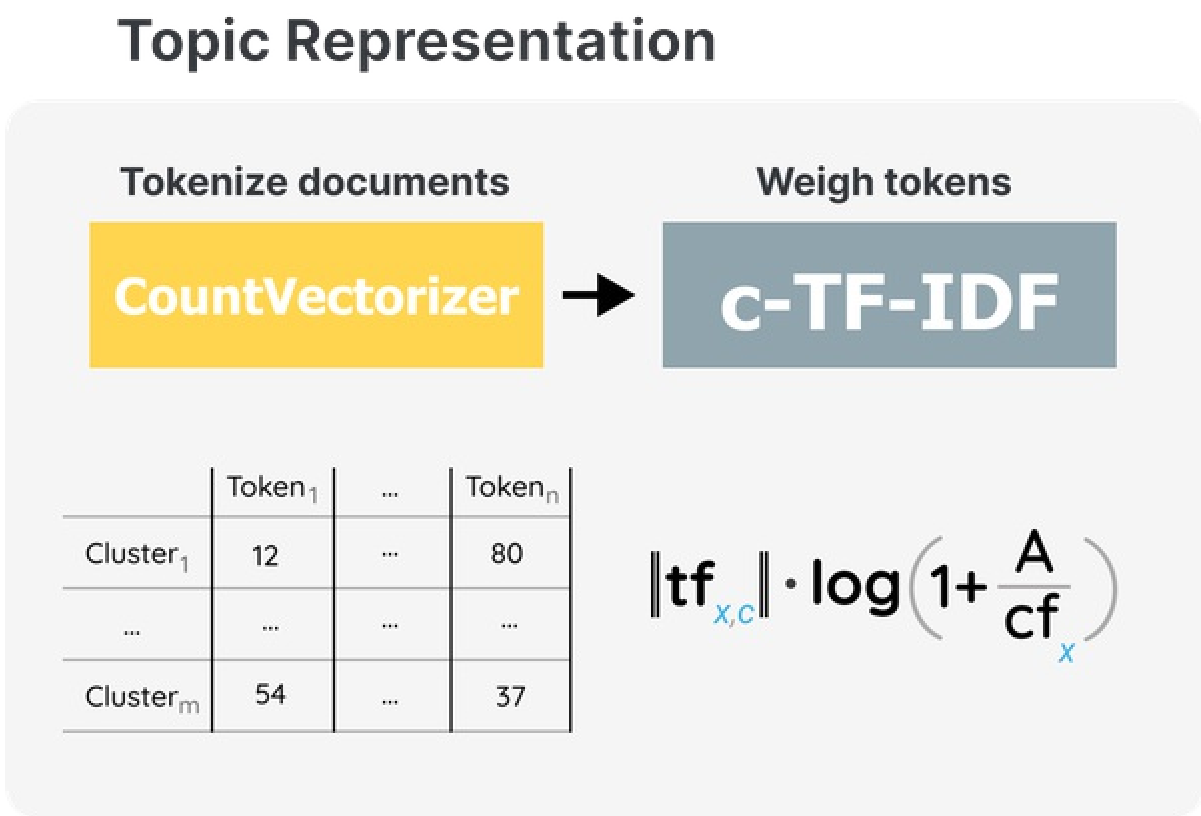

Figure 3-9. The second part of BERTopic’s pipeline is representing the topics. The calculation of the weight of term *x* in a class *c*.

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.

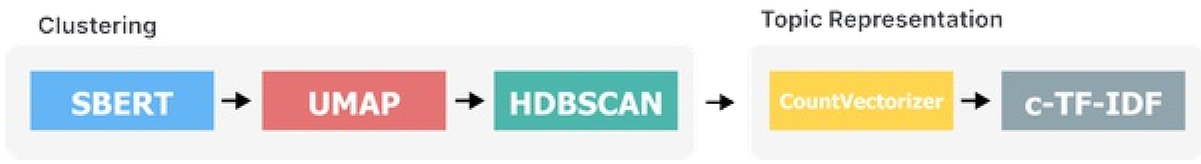

Figure 3-10. The full pipeline of BERTopic, roughly, consists of two steps, clustering and topic representation.

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.

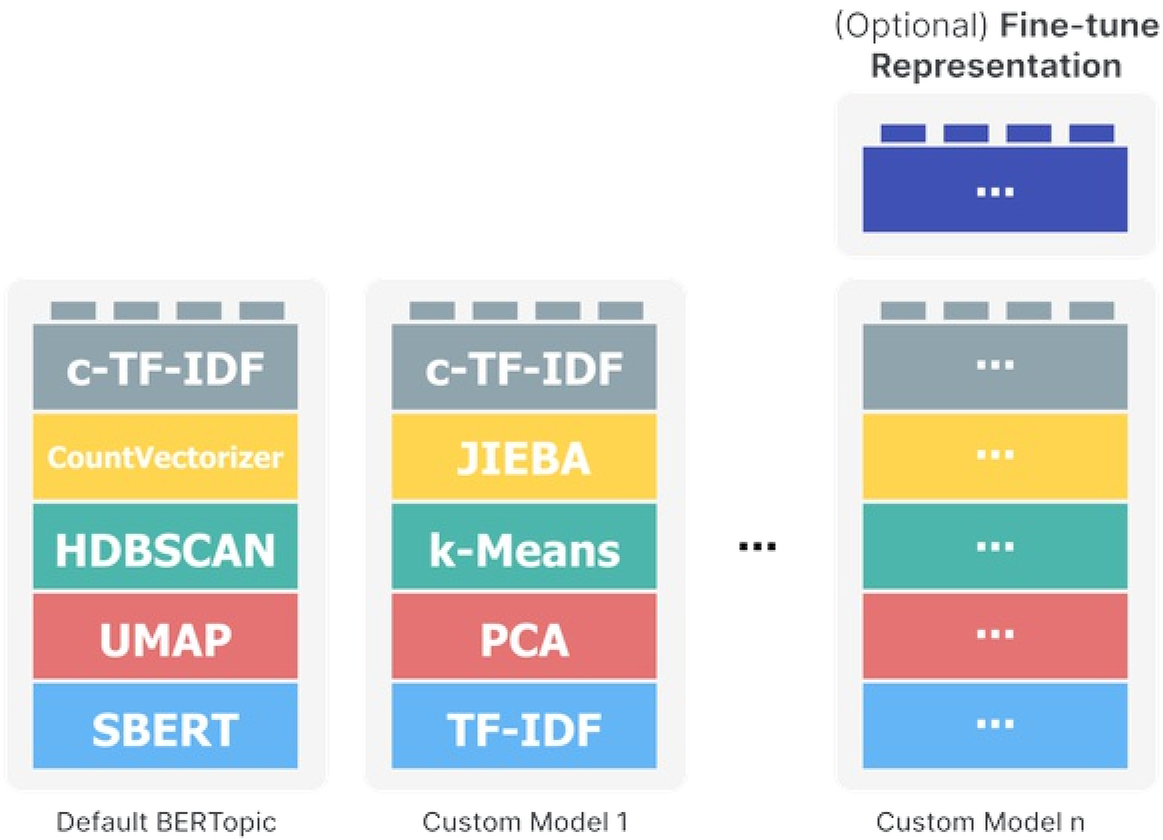

Figure 3-11. The modularity of BERTopic is a key component and allows you to build your own topic model whoever you want.

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 CountVecto

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 each individual step:

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

# 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', 0.009363770589253441),
 ('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 his 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.approxima
df = topic_model.visualize_approximate_distribution(ab
df
```

```
 Topic models can be useful tools to discover latent topics in collections of
documents.
```

	Topic	models	can	be	useful	tools	to	discover	latent	topics	in	collections	of	documents
19_topic_topics_lda_latent	0.605	0.605	0.605	0.605		0.102	0.381	0.660	0.941	1.082	0.917	0.637	0.356	0.115
55_tweets_hashtag_twitter_social	0.110	0.110	0.110	0.110										
88_layout_document_documents_extraction											0.117	0.117	0.117	0.117
98_languages_african_resources_technologies			0.113	0.224	0.224	0.224	0.111							
100_latent_variational_vae_variables						0.188	0.385	0.581	0.758	0.570	0.373	0.177		
187_annotation_tools_nerd_manufacturing			0.190	0.376	0.503	0.616	0.427	0.241	0.113					
244_latent_variational_variables_responses						0.154	0.314	0.473	0.620	0.465	0.306	0.146		

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.

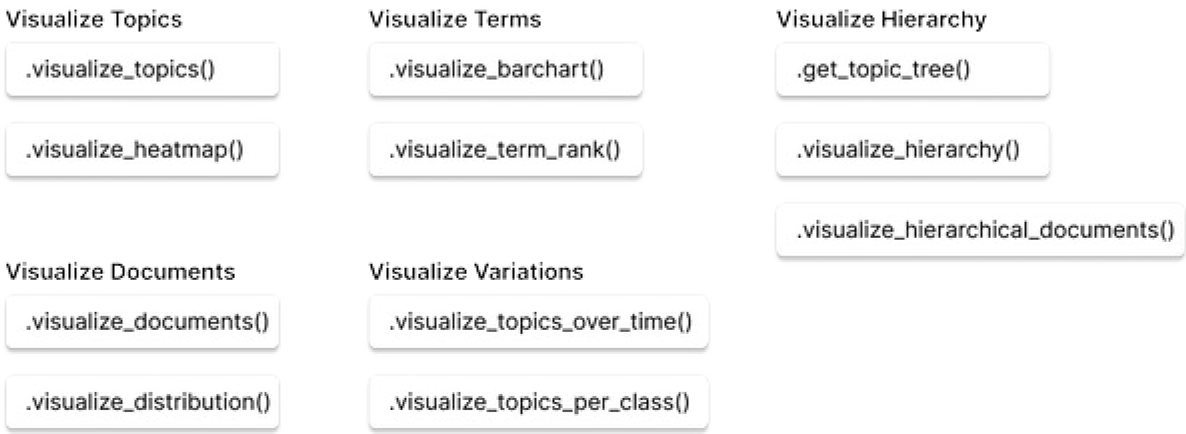

Figure 3-13. A wide range of visualization options are available in BERTopic.

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()
```

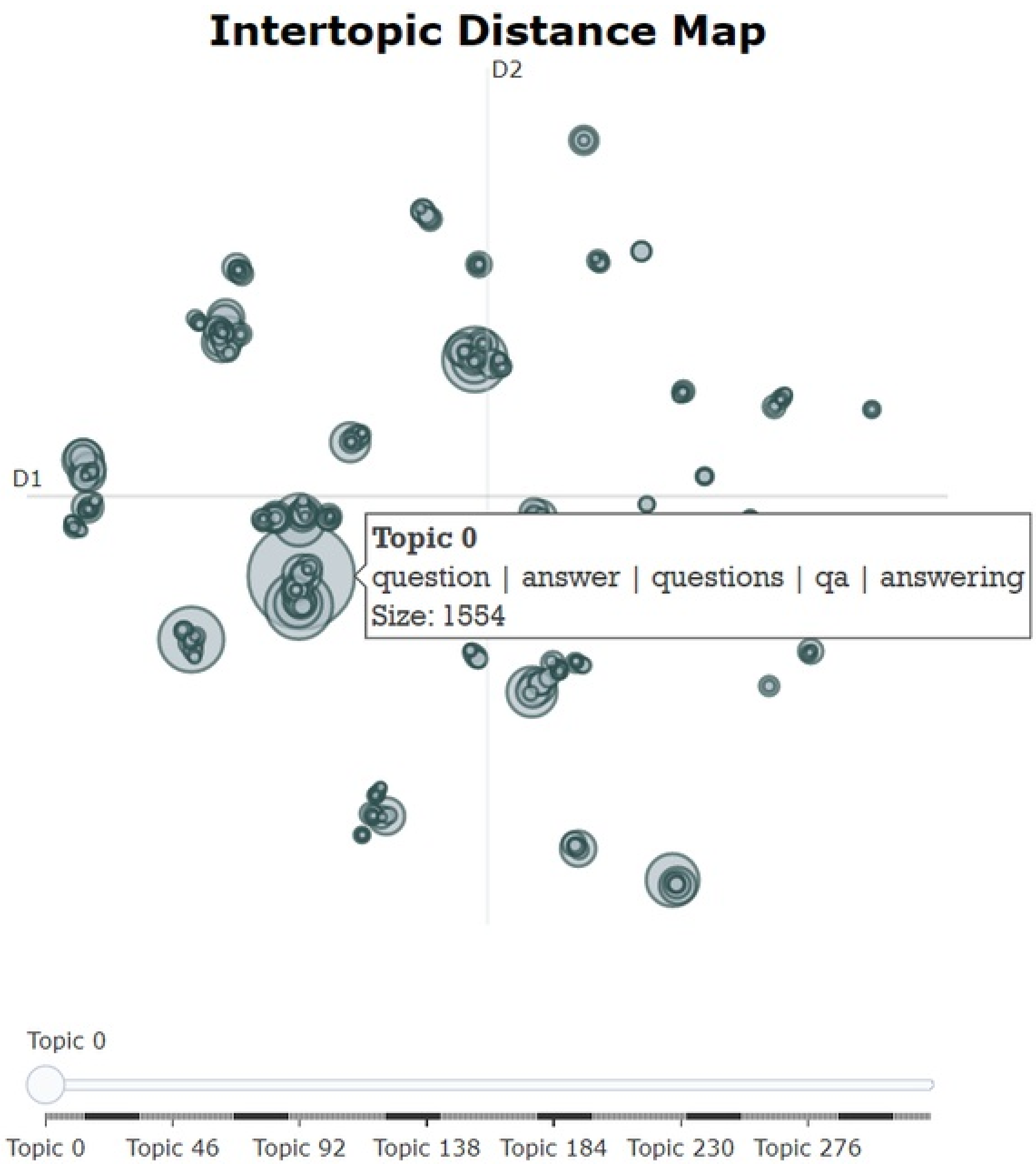

Figure 3-14. The intertopic distance map of topics represented in 2D space.

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])
```

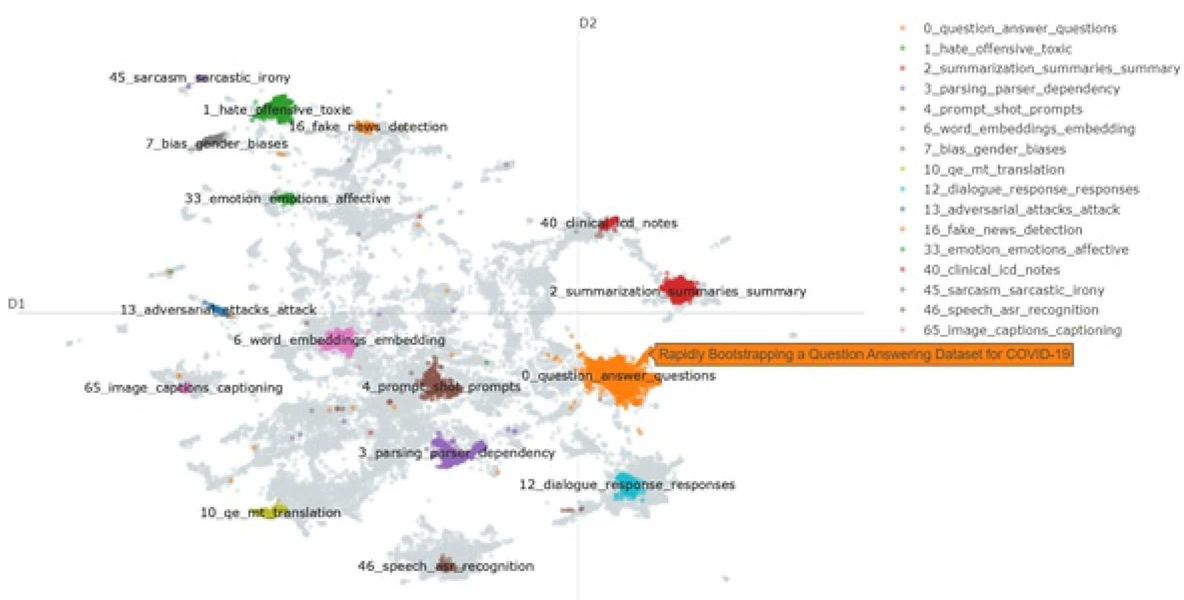

Figure 3-15. Abstracts and their topics are represented in a 2D visualization.

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.

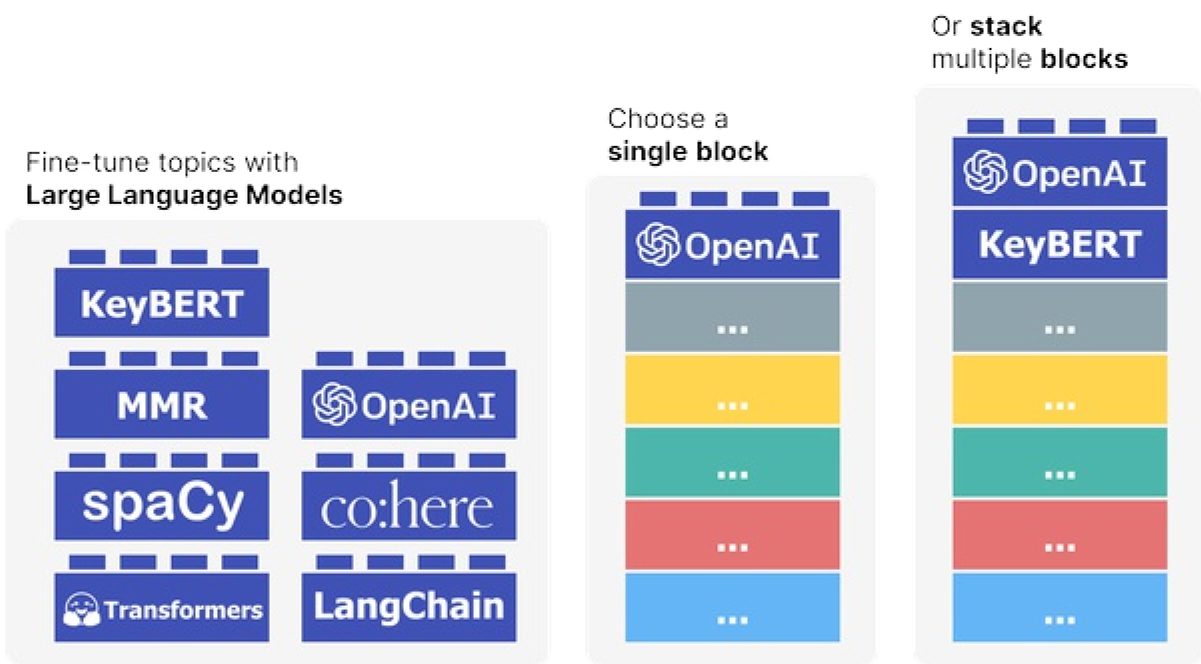

Figure 3-17. After applying the c-TF-IDF weighting, topics can be fine-tuned with a wide variety of representation models. Many of which are Large Language Models.

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_represent
```

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_leng
  """ 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[to
    new_words = " | ".join(list(zip(*model.get_topic(t

    # 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.

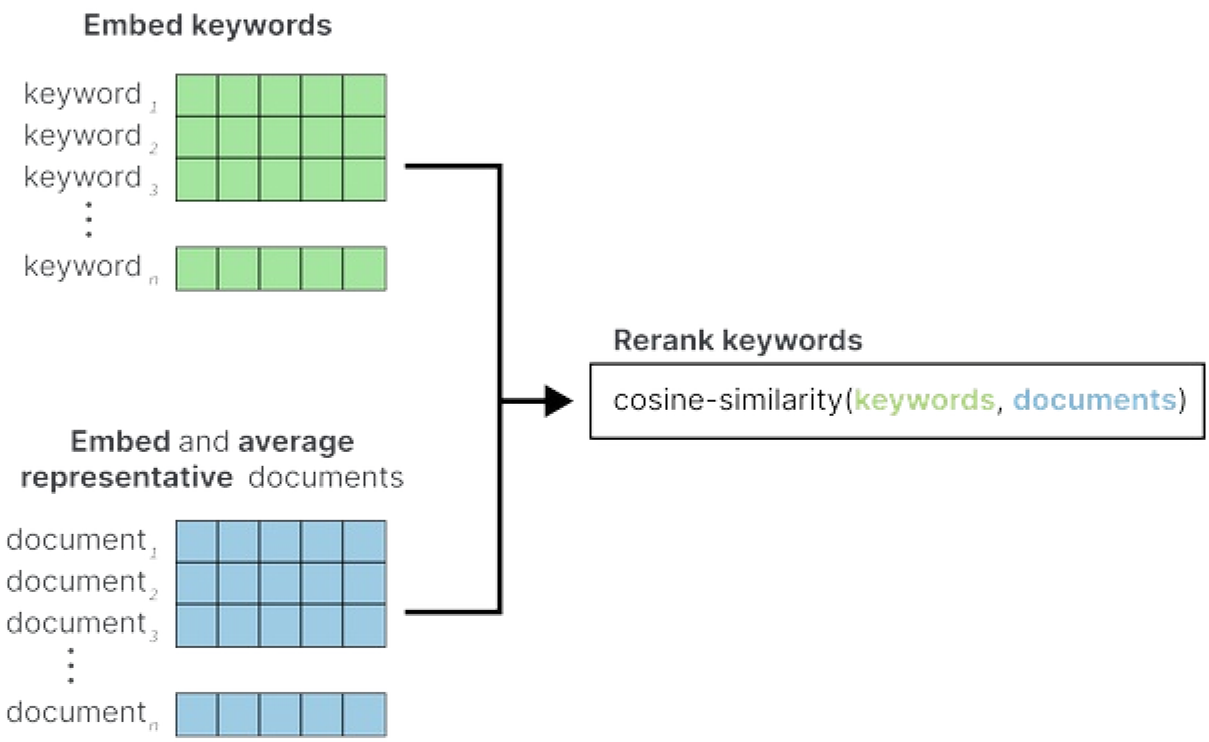

Figure 3-18. The procedure of the KeyBERTInspired representation model

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

# Update our topic representations
new_topic_model.update_topics(abstracts, representatio

# 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.

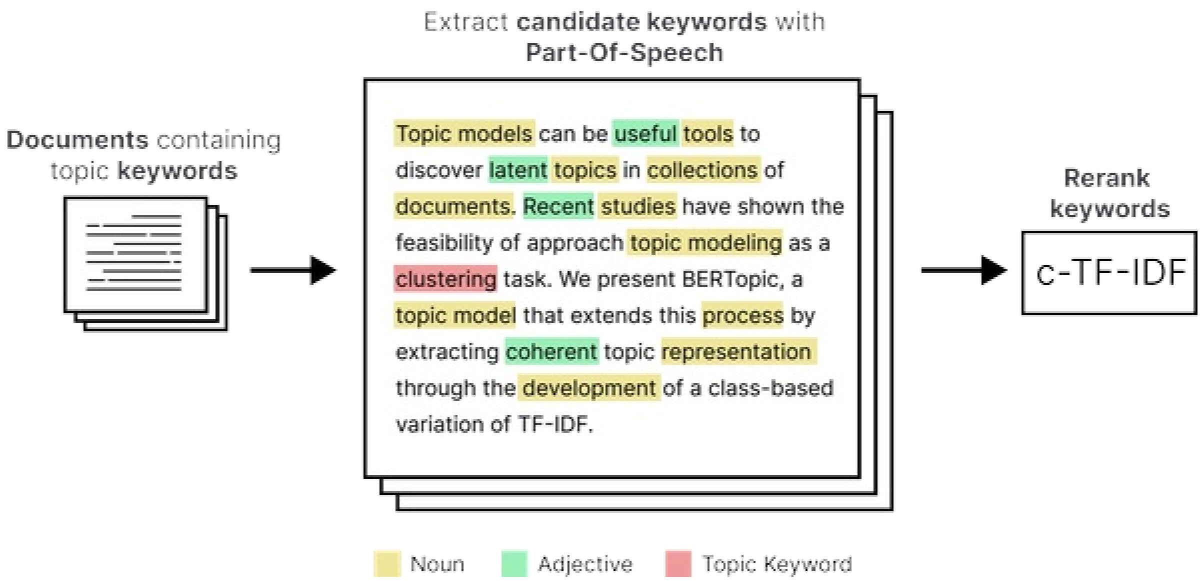

Figure 3-19. The procedure of the PartOfSpeech representation model

```
# 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)

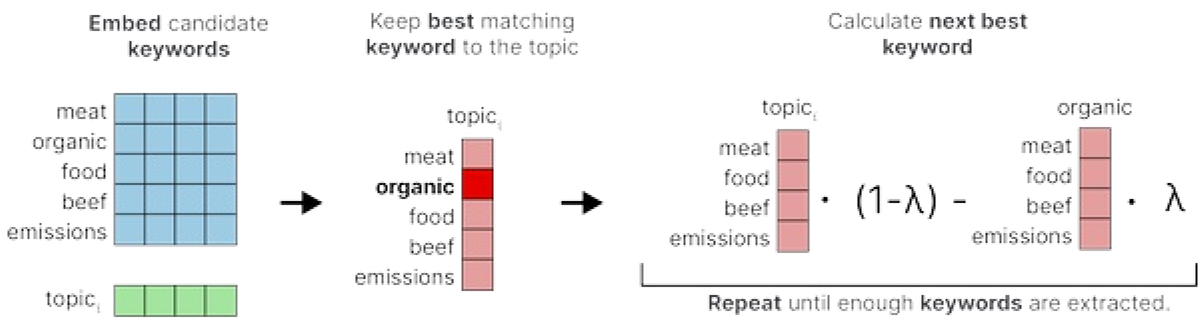

Figure 3-20. The procedure of the Maximal Marginal Relevance representation model. The diversity of the resulting keywords is represented by lambda (λ).

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(divers

# 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
--> 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.

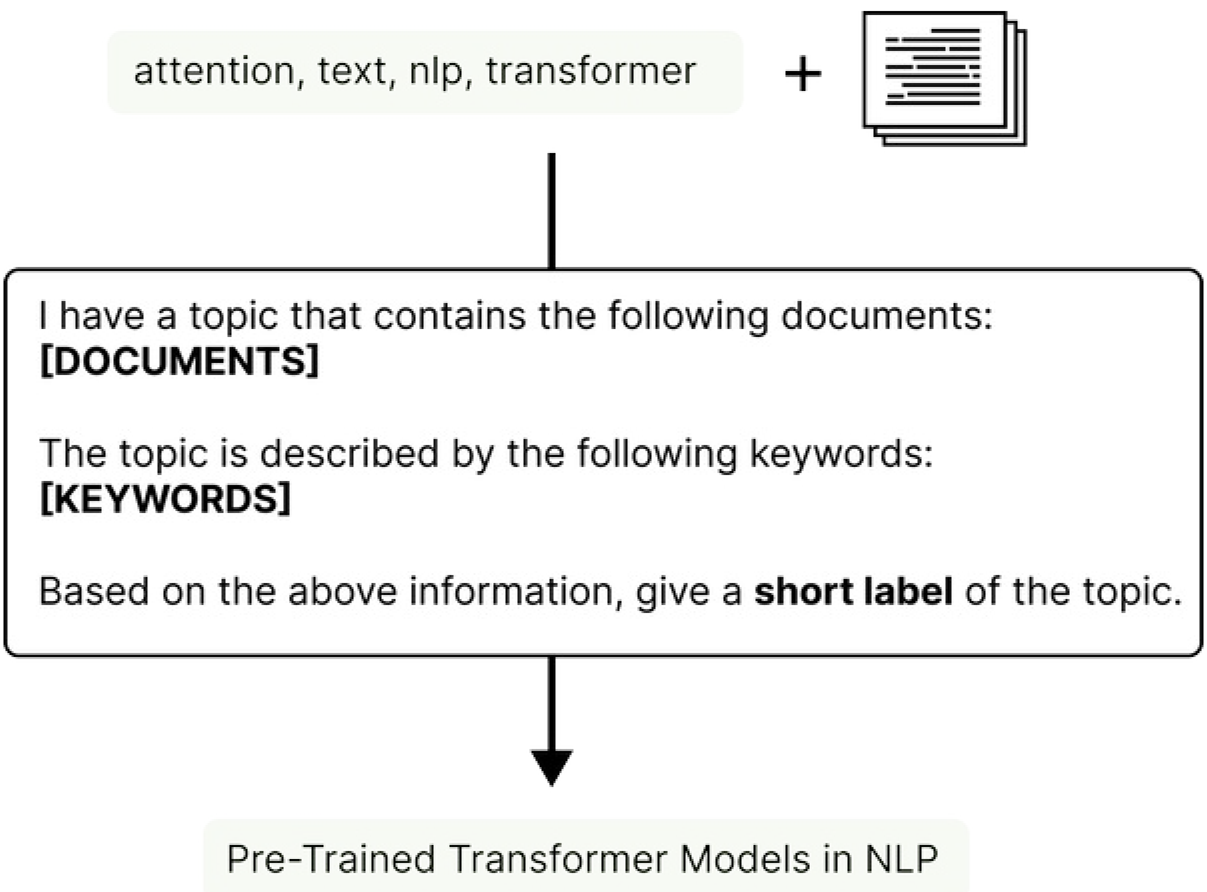

Figure 3-21. Use text generative LLMs and prompt engineering to create labels for topics from keywords and documents related to each topic.

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: [KEY

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

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: [KEY
```

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: [KEY

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.

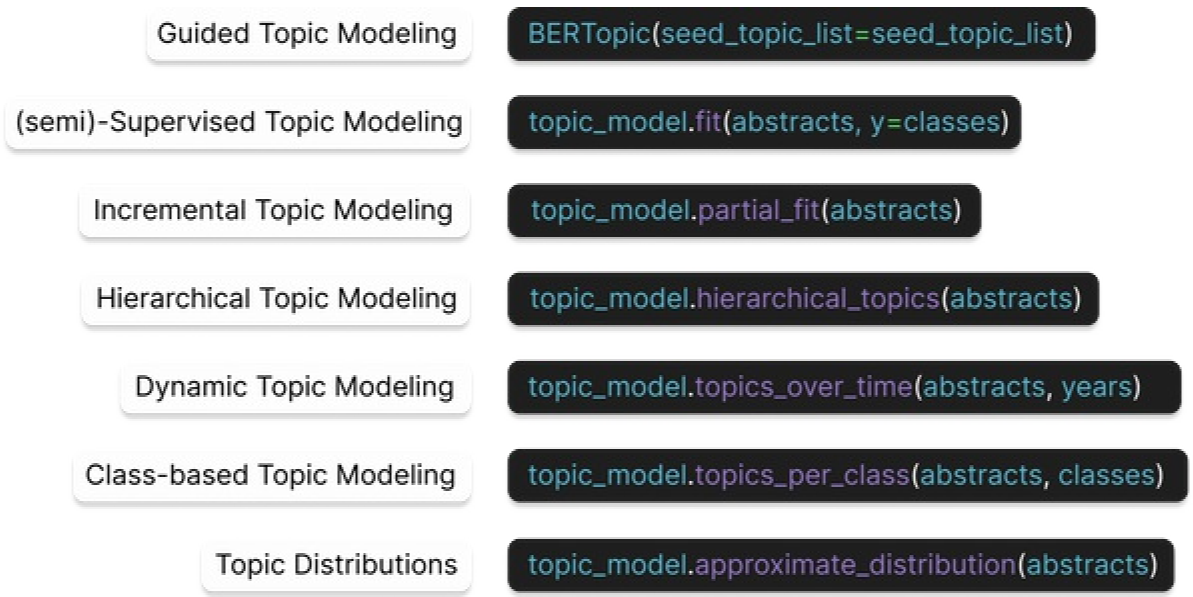

Figure 3-22. -X Topic Modeling Variations in BERTopic

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. 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.

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 4-1 shows this distinction.

Figure 4-1. 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 4-2.

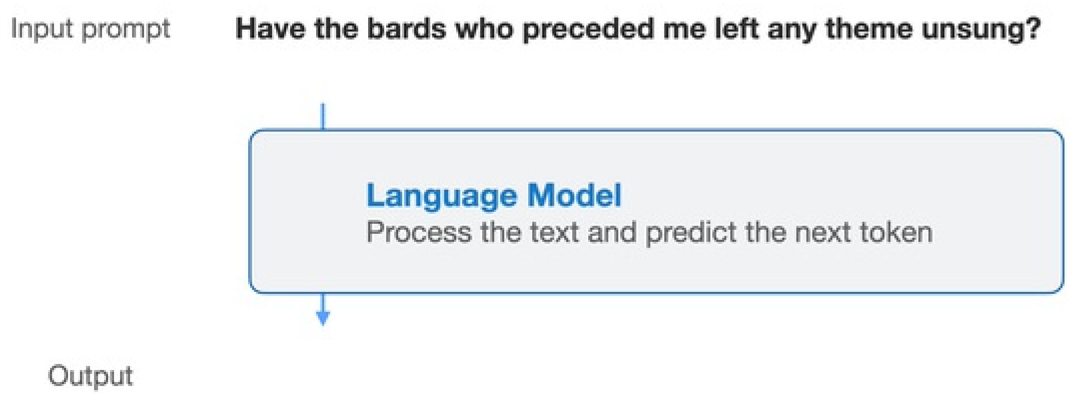

Figure 4-2. High-level view of a language model and its input prompt.

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 r
# 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_nam
```

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").inp
# 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 4-3. 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.

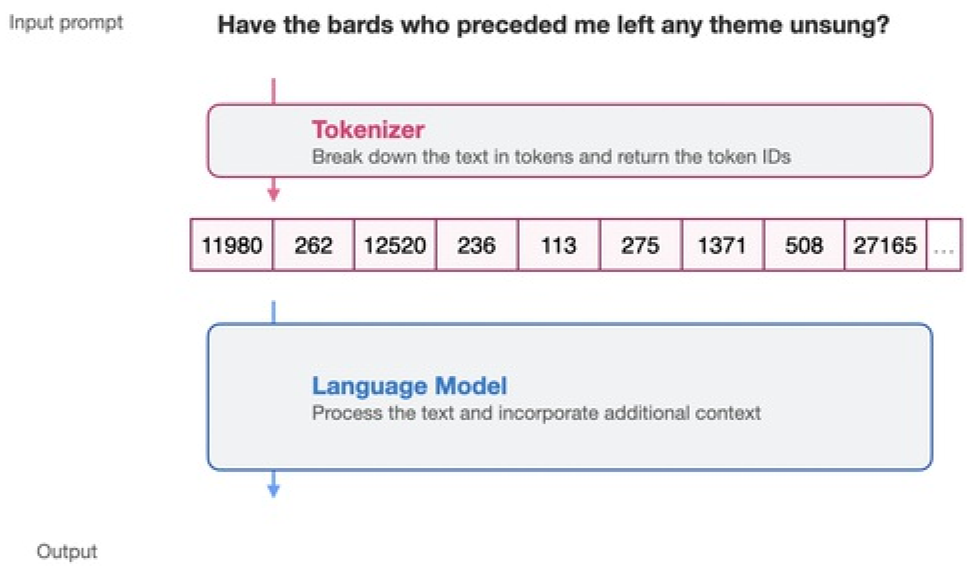

Figure 4-3. A tokenizer processes the input prompt and prepares the actual input into the language model: a list of token ids.

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 4-4 shows.

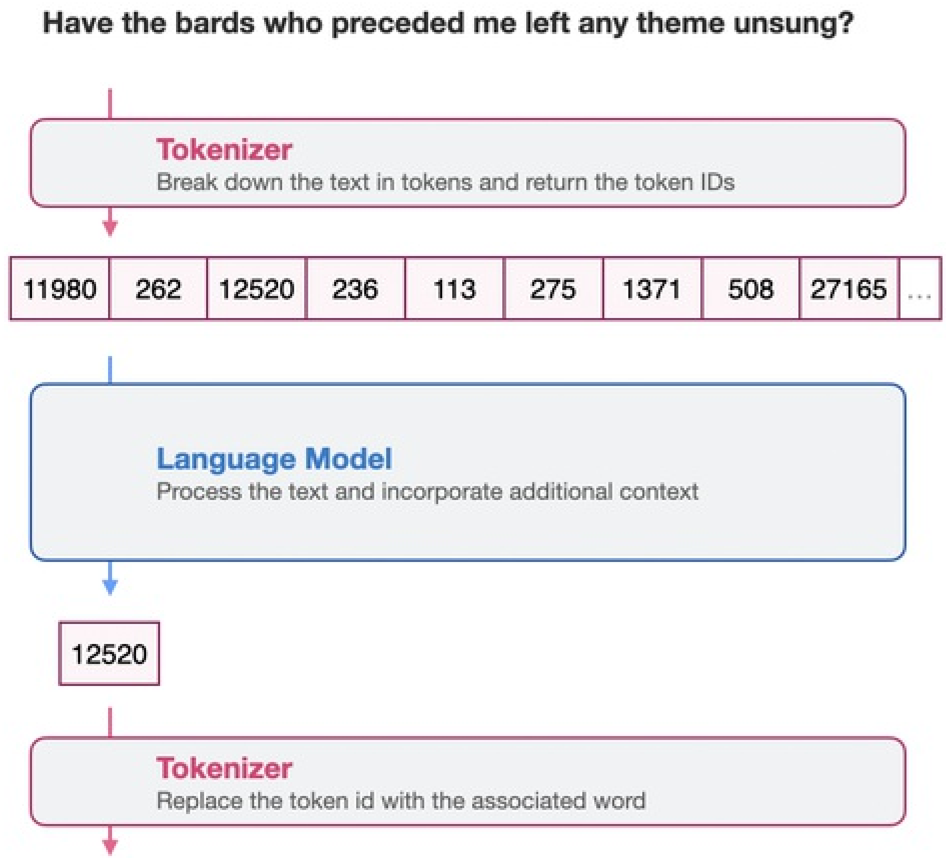

Figure 4-4. Tokenizers are also used to process the output of the model by converting the output token ID into the word or token associated with that ID.

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 4-5. 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.

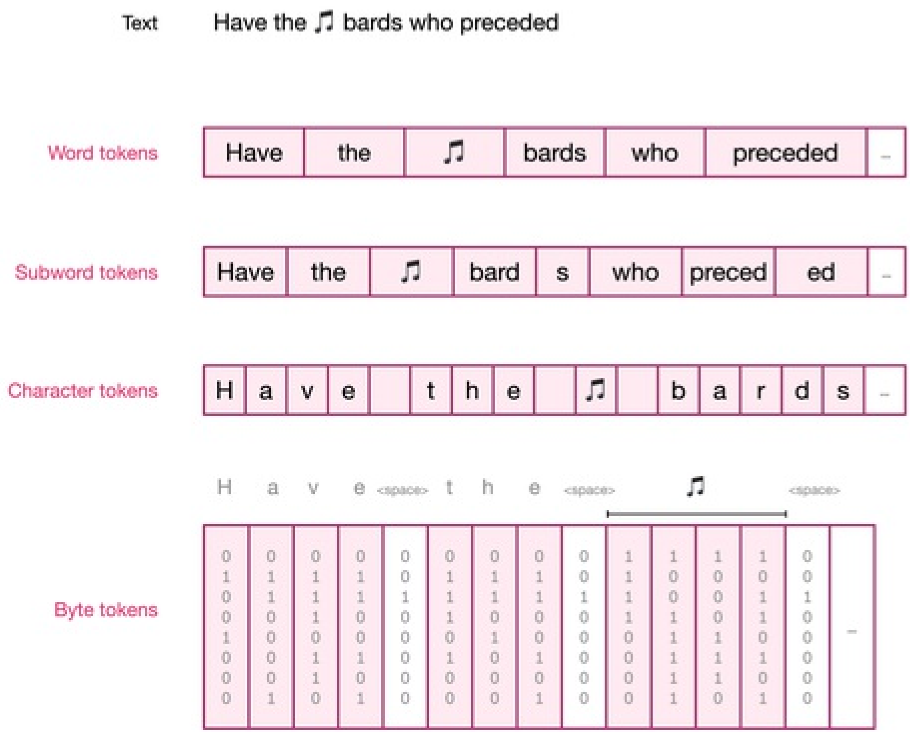

Figure 4-5. There are multiple methods of tokenization that break down the text to different sizes of components (words, subwords, characters, and bytes).

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 (sub word 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
␣堩蟠
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 CAP ITAL IZ ATION

� � � � � �

show _ t ok ens False None el if == >= 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 CA PI TAL IZ ATION <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 t abs : "              "
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 ##Z`
gpt2	`English and CAP ITAL IZ ATION � � � �`

google/flan-t5-xxl	`English and CA PI TAL IZ ATION <unk> <unk>`
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 4-6. 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 4-6. 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).

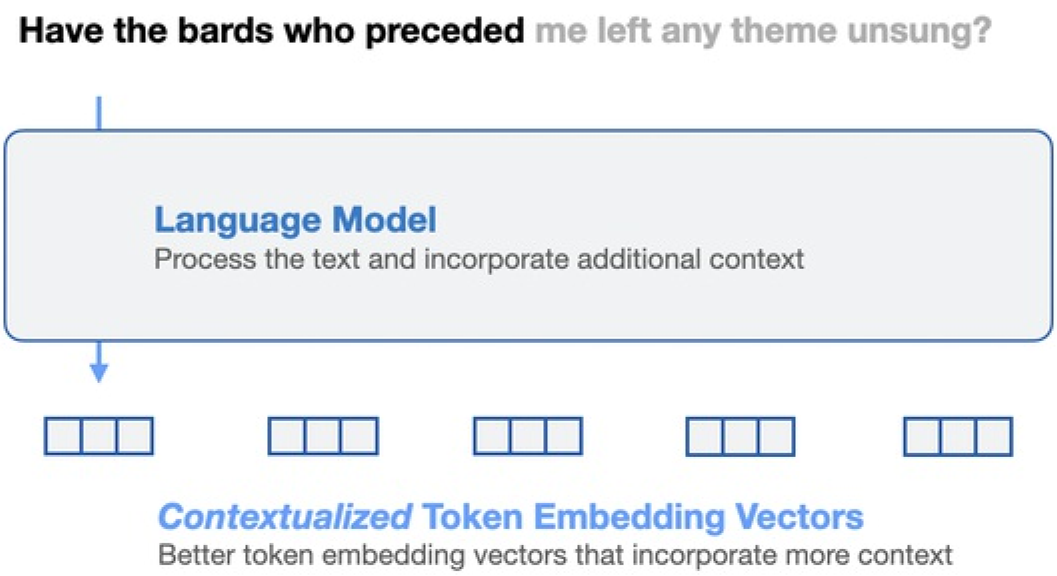

Figure 4-7. Language models produce contextualized token embeddings that improve on raw, static token embeddings

Instead of representing each token or word with a static vector, language models create contextualized word embeddings (shown in Figure 4-7) 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 4-8. Technically, the switch from token IDs into raw embeddings is the first step that happens inside a language model.

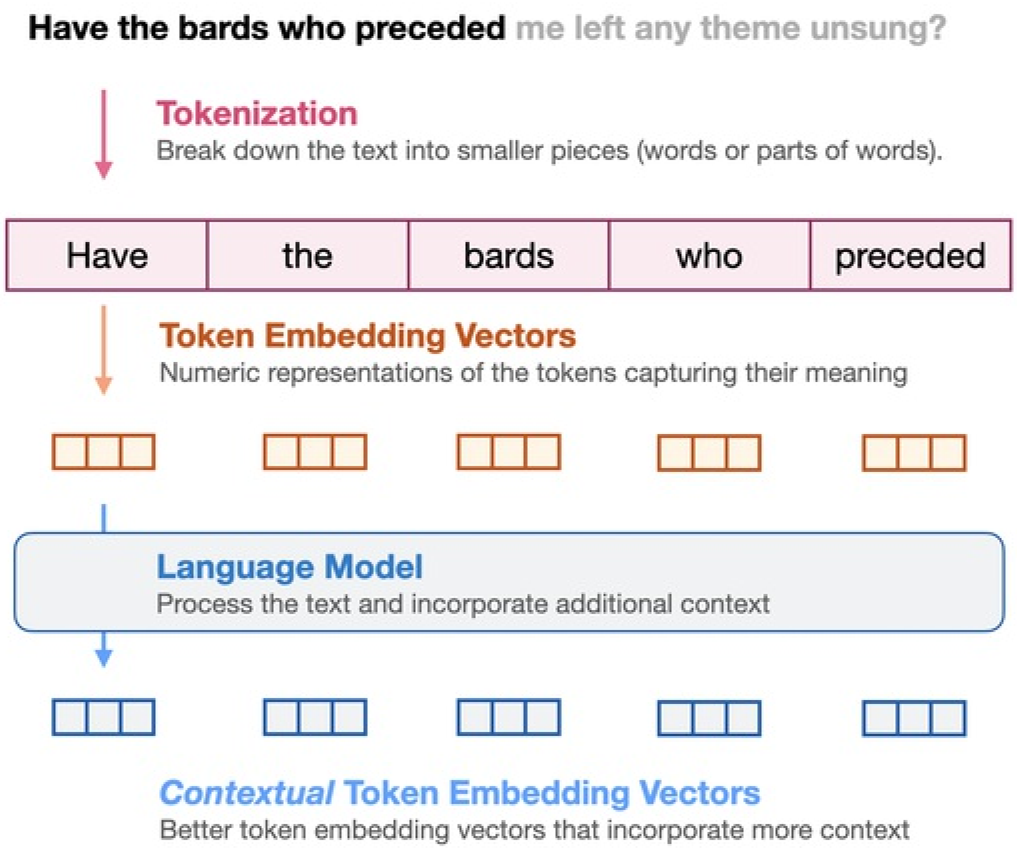

Figure 4-8. A language model operates on raw, static embeddings as its input and produces contextual text embeddings.

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 wikiped
# 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 4-9.

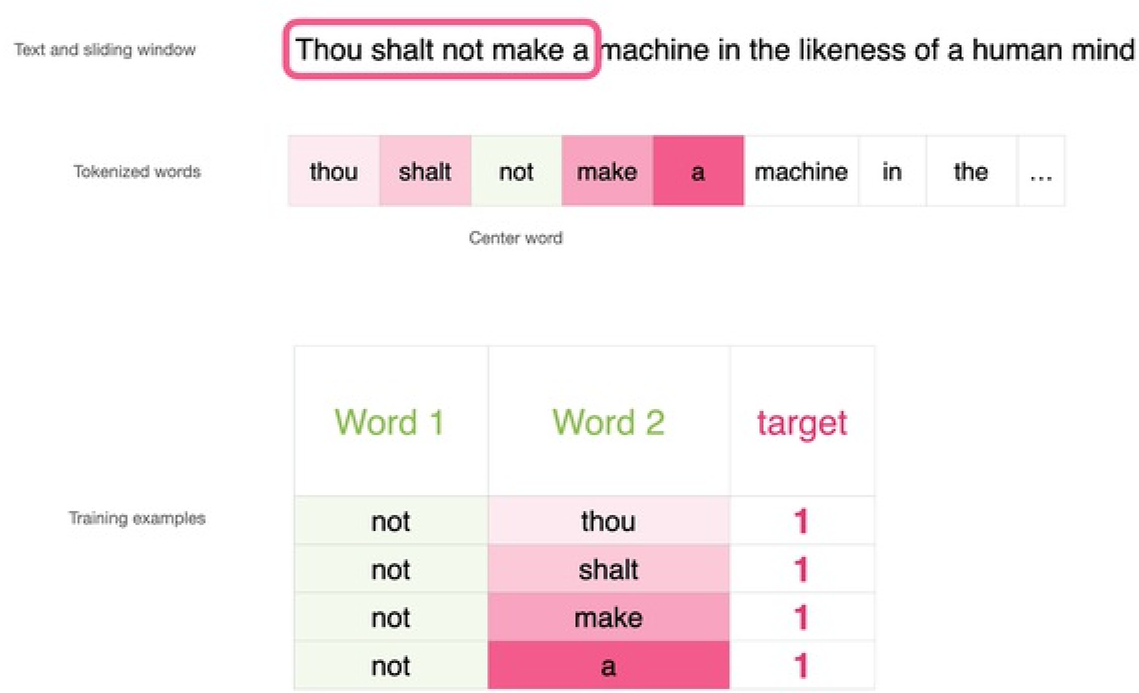

Figure 4-9. A sliding window is used to generate training examples for the word2vec algorithm to later predict if two words are neighbors or not.

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 4-10.

Training examples

Word 1	Word 2	Target
not	thou	1
not	shalt	1
not	make	1
not	a	1

Figure 4-10. 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 4-11.

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

Positive Examples

Negative Examples

Figure 4-11. 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 4-12): Skipgram - the method of selecting neighboring words and negative sampling

- adding negative examples by random sampling from the dataset.

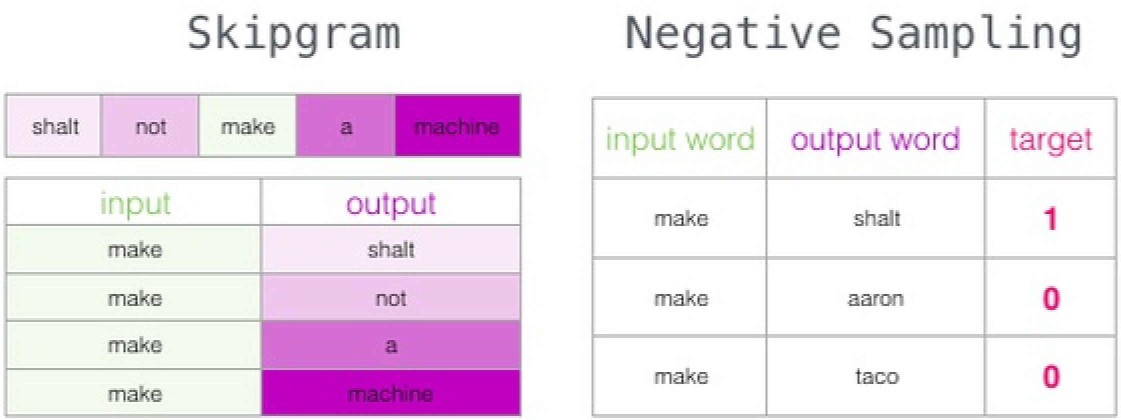

Figure 4-12. Skipgram and Negative Sampling are two of the main ideas behind the word2vec algorithm and are useful in many other problems that can be formulated as token sequence problems.

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 4-13. In practice, this is a matrix of dimensions vocab_size x embedding_dimensions.

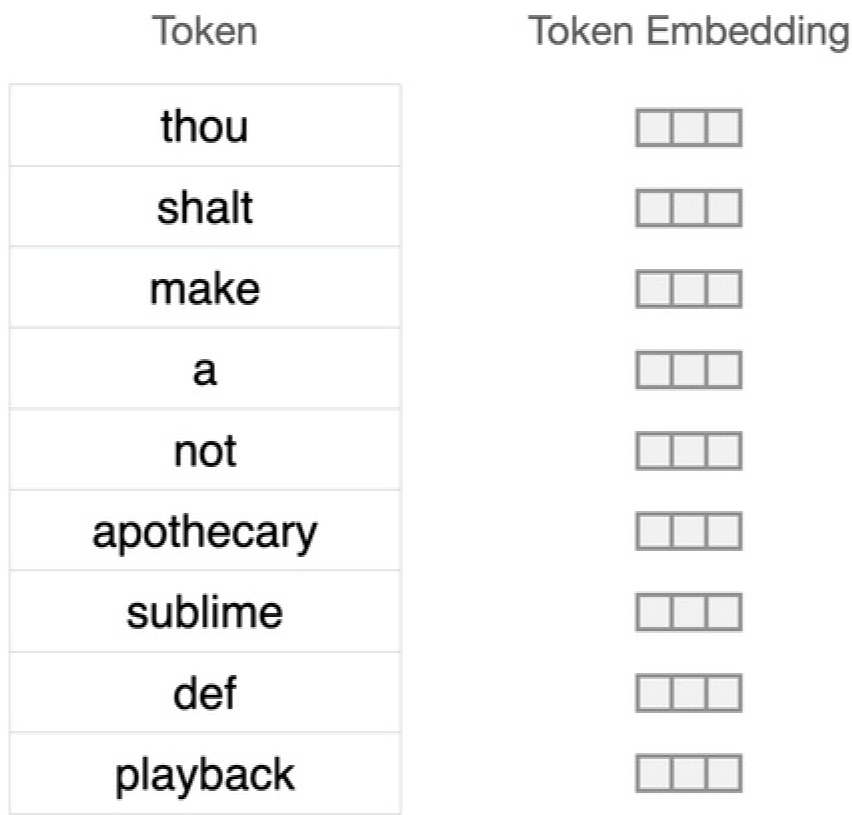

Figure 4-13. A vocabulary of words and their starting, random, uninitialized embedding vectors.

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 4-14:

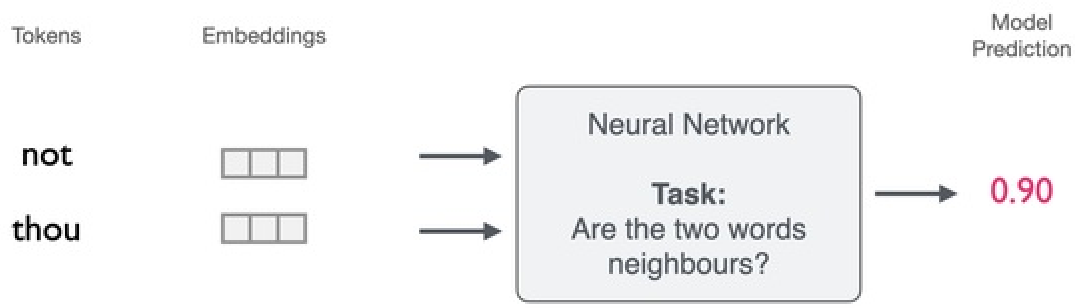

Figure 4-14. A neural network is trained to predict if two words are neighbors. It updates the

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 4-15 demonstrates this dataset.

Figure 4-15. 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.

About the Authors

Jay Alammar is Director and Engineering Fellow at Cohere (pioneering provider of large language models as an API). A role in which 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). Jay is also a co-creator of popular machine learning and natural language processing courses on Udacity.

Maarten Grootendorst is a 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 blog, 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 are used by data professionals and organizations across the world.