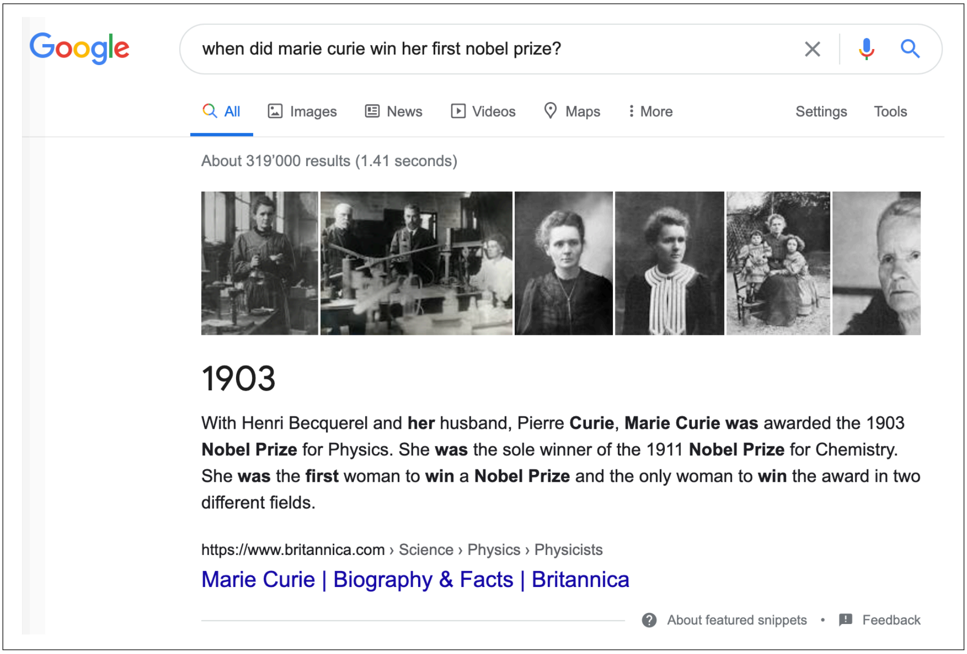
CHAPTER 7

Question Answering

Whether you're a researcher, analyst, or data scientist, chances are that at some point you've needed to wade through oceans of documents to find the information you're looking for. To make matters worse, you're constantly reminded by Google and Bing that there exist better ways to search! For instance, if we search for "When did Marie Curie win her first Nobel Prize?" on Google, we immediately get the correct answer of "1903," as illustrated in Figure 7-1.



About 319'000 results (1.41 seconds)

1903

With Henri Becquerel and **her** husband, Pierre **Curie**, **Marie Curie was** awarded the 1903 **Nobel Prize** for Physics. She **was** the sole winner of the 1911 **Nobel Prize** for Chemistry. She **was** the **first** woman to **win** a **Nobel Prize** and the only woman to **win** the award in two different fields.

https://www.britannica.com › Science › Physics › Physicists

Marie Curie | Biography & Facts | Britannica

Figure 7-1. A Google search query and corresponding answer snippet

In this example, Google first retrieved around 319,000 documents that were relevant to the query, and then performed an additional processing step to extract the answer snippet with the corresponding passage and web page. It's not hard to see why these answer snippets are useful. For example, if we search for a trickier question like "Which guitar tuning is the best?" Google doesn't provide an answer, and instead we have to click on one of the web pages returned by the search engine to find it ourselves.[1]

The general approach behind this technology is called *question answering* (QA). There are many flavors of QA, but the most common is *extractive QA*, which involves questions whose answer can be identified as a span of text in a document, where the document might be a web page, legal contract, or news article. The two-stage process of first retrieving relevant documents and then extracting answers from them is also the basis for many modern QA systems, including semantic search engines, intelligent assistants, and automated information extractors. In this chapter, we'll apply this process to tackle a common problem facing ecommerce websites: helping consumers answer specific queries to evaluate a product. We'll see that customer reviews can be used as a rich and challenging source of information for QA, and along the way we'll learn how transformers act as powerful *reading comprehension* models that can extract meaning from text. Let's begin by fleshing out the use case.

> This chapter focuses on extractive QA, but other forms of QA may be more suitable for your use case. For example, *community QA* involves gathering question-answer pairs that are generated by users on forums like Stack Overflow, and then using semantic similarity search to find the closest matching answer to a new question. There is also *long-form QA*, which aims to generate complex paragraph-length answers to open-ended questions like "Why is the sky blue?" Remarkably, it is also possible to do QA over tables, and transformer models like TAPAS can even perform aggregations to produce the final answer!

Building a Review-Based QA System

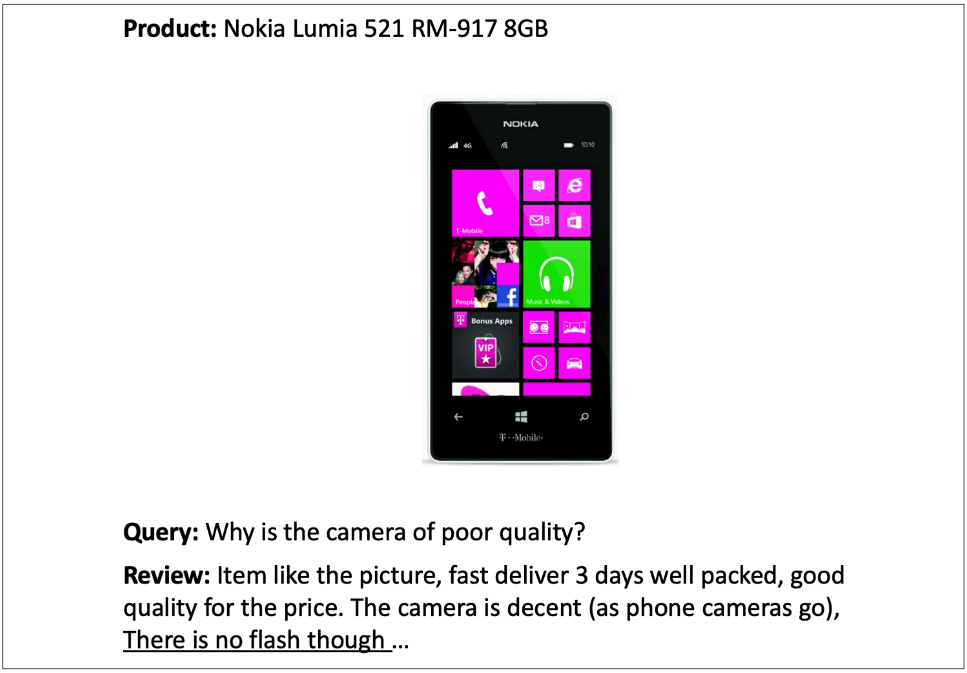
If you've ever purchased a product online, you probably relied on customer reviews to help inform your decision. These reviews can often help answer specific questions like "Does this guitar come with a strap?" or "Can I use this camera at night?" that may be hard to answer from the product description alone. However, popular products can have hundreds to thousands of reviews, so it can be a major drag to find one that is relevant. One alternative is to post your question on the community QA

[1] Although, in this particular case, everyone agrees that Drop C is the best guitar tuning.

platforms provided by websites like Amazon, but it usually takes days to get an answer (if you get one at all). Wouldn't it be nice if we could get an immediate answer, like in the Google example from Figure 7-1? Let's see if we can do this using transformers!

The Dataset

To build our QA system we'll use the SubjQA dataset,[2] which consists of more than 10,000 customer reviews in English about products and services in six domains: TripAdvisor, Restaurants, Movies, Books, Electronics, and Grocery. As illustrated in Figure 7-2, each review is associated with a question that can be answered using one or more sentences from the review.[3]



Product: Nokia Lumia 521 RM-917 8GB

Query: Why is the camera of poor quality?

Review: Item like the picture, fast deliver 3 days well packed, good quality for the price. The camera is decent (as phone cameras go), There is no flash though ...

Figure 7-2. A question about a product and the corresponding review (the answer span is underlined)

The interesting aspect of this dataset is that most of the questions and answers are *subjective*; that is, they depend on the personal experience of the users. The example in Figure 7-2 shows why this feature makes the task potentially more difficult than

[2] J. Bjerva et al., "SubjQA: A Dataset for Subjectivity and Review Comprehension", (2020).

[3] As we'll soon see, there are also *unanswerable* questions that are designed to produce more robust models.

finding answers to factual questions like “What is the currency of the United Kingdom?” First, the query is about “poor quality,” which is subjective and depends on the user’s definition of quality. Second, important parts of the query do not appear in the review at all, which means it cannot be answered with shortcuts like keyword search or paraphrasing the input question. These features make SubjQA a realistic dataset to benchmark our review-based QA models on, since user-generated content like that shown in Figure 7-2 resembles what we might encounter in the wild.

> QA systems are usually categorized by the *domain* of data that they have access to when responding to a query. *Closed-domain* QA deals with questions about a narrow topic (e.g., a single product category), while *open-domain* QA deals with questions about almost anything (e.g., Amazon’s whole product catalog). In general, closed-domain QA involves searching through fewer documents than the open-domain case.

To get started, let’s download the dataset from the Hugging Face Hub. As we did in Chapter 4, we can use the `get_dataset_config_names()` function to find out which subsets are available:

```
from datasets import get_dataset_config_names

domains = get_dataset_config_names("subjqa")
domains
```

```
['books', 'electronics', 'grocery', 'movies', 'restaurants', 'tripadvisor']
```

For our use case, we’ll focus on building a QA system for the Electronics domain. To download the `electronics` subset, we just need to pass this value to the `name` argument of the `load_dataset()` function:

```
from datasets import load_dataset

subjqa = load_dataset("subjqa", name="electronics")
```

Like other question answering datasets on the Hub, SubjQA stores the answers to each question as a nested dictionary. For example, if we inspect one of the rows in the `answers` column:

```
print(subjqa["train"]["answers"][1])
```

```
{'text': ['Bass is weak as expected', 'Bass is weak as expected, even with EQ
adjusted up'], 'answer_start': [1302, 1302], 'answer_subj_level': [1, 1],
'ans_subj_score': [0.5083333253860474, 0.5083333253860474], 'is_ans_subjective':
[True, True]}
```

we can see that the answers are stored in a `text` field, while the starting character indices are provided in `answer_start`. To explore the dataset more easily, we’ll flatten these nested columns with the `flatten()` method and convert each split to a Pandas `DataFrame` as follows:

```
import pandas as pd

dfs = {split: dset.to_pandas() for split, dset in subjqa.flatten().items()}

for split, df in dfs.items():
    print(f"Number of questions in {split}: {df['id'].nunique()}")
```

```
Number of questions in train: 1295
Number of questions in test: 358
Number of questions in validation: 255
```

Notice that the dataset is relatively small, with only 1,908 examples in total. This simulates a real-world scenario, since getting domain experts to label extractive QA datasets is labor-intensive and expensive. For example, the CUAD dataset for extractive QA on legal contracts is estimated to have a value of $2 million to account for the legal expertise needed to annotate its 13,000 examples![4]

There are quite a few columns in the SubjQA dataset, but the most interesting ones for building our QA system are shown in Table 7-1.

Table 7-1. Column names and their descriptions from the SubjQA dataset

Column name	Description
`title`	The Amazon Standard Identification Number (ASIN) associated with each product
`question`	The question
`answers.answer_text`	The span of text in the review labeled by the annotator
`answers.answer_start`	The start character index of the answer span
`context`	The customer review

Let's focus on these columns and take a look at a few of the training examples. We can use the `sample()` method to select a random sample:

```
qa_cols = ["title", "question", "answers.text",
           "answers.answer_start", "context"]
sample_df = dfs["train"][qa_cols].sample(2, random_state=7)
sample_df
```

[4] D. Hendrycks et al., "CUAD: An Expert-Annotated NLP Dataset for Legal Contract Review", (2021).

title	question	answers.text	answers.answer_start	context
B005DKZTMG	Does the keyboard lightweight?	[this keyboard is compact]	[215]	I really like this keyboard. I give it 4 stars because it doesn’t have a CAPS LOCK key so I never know if my caps are on. But for the price, it really suffices as a wireless keyboard. I have very large hands and this keyboard is compact, but I have no complaints.
B00AAIPT76	How is the battery?	[]	[]	I bought this after the first spare gopro battery I bought wouldn’t hold a charge. I have very realistic expectations of this sort of product, I am skeptical of amazing stories of charge time and battery life but I do expect the batteries to hold a charge for a couple of weeks at least and for the charger to work like a charger. In this I was not disappointed. I am a river rafter and found that the gopro burns through power in a hurry so this purchase solved that issue. the batteries held a charge, on shorter trips the extra two batteries were enough and on longer trips I could use my friends JOOS Orange to recharge them.I just bought a newtrent xtreme powerpak and expect to be able to charge these with that so I will not run out of power again.

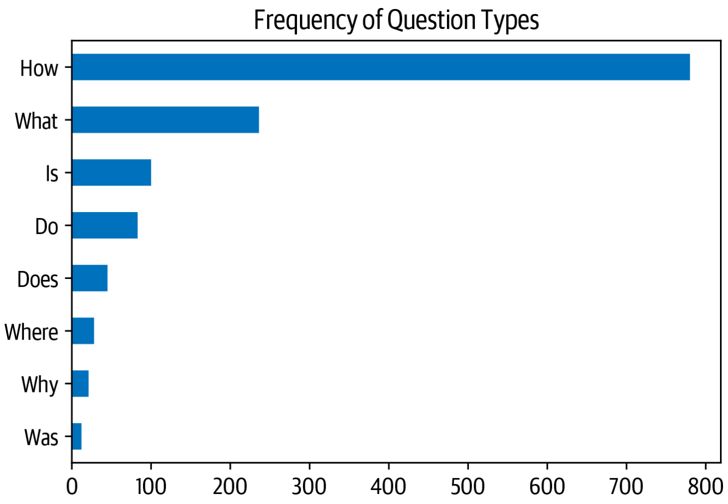
From these examples we can make a few observations. First, the questions are not grammatically correct, which is quite common in the FAQ sections of ecommerce websites. Second, an empty answers.text entry denotes “unanswerable” questions whose answer cannot be found in the review. Finally, we can use the start index and length of the answer span to slice out the span of text in the review that corresponds to the answer:

```
start_idx = sample_df["answers.answer_start"].iloc[0][0]
end_idx = start_idx + len(sample_df["answers.text"].iloc[0][0])
sample_df["context"].iloc[0][start_idx:end_idx]
```

```
'this keyboard is compact'
```

Next, let’s get a feel for what types of questions are in the training set by counting the questions that begin with a few common starting words:

```
counts = {}
question_types = ["What", "How", "Is", "Does", "Do", "Was", "Where", "Why"]

for q in question_types:
    counts[q] = dfs["train"]["question"].str.startswith(q).value_counts()[True]

pd.Series(counts).sort_values().plot.barh()
plt.title("Frequency of Question Types")
plt.show()
```



We can see that questions beginning with “How”, “What”, and “Is” are the most common ones, so let’s have a look at some examples:

```
for question_type in ["How", "What", "Is"]:
    for question in (
        dfs["train"][dfs["train"].question.str.startswith(question_type)]
        .sample(n=3, random_state=42)['question']):
        print(question)
```

```
How is the camera?
How do you like the control?
How fast is the charger?
What is direction?
What is the quality of the construction of the bag?
What is your impression of the product?
Is this how zoom works?
Is sound clear?
Is it a wireless keyboard?
```

The Stanford Question Answering Dataset

The *(question, review, [answer sentences])* format of SubjQA is commonly used in extractive QA datasets, and was pioneered in the Stanford Question Answering Dataset (SQuAD).[5] This is a famous dataset that is often used to test the ability of machines to read a passage of text and answer questions about it. The dataset was created by sampling several hundred English articles from Wikipedia, partitioning each article into paragraphs, and then asking crowdworkers to generate a set of questions

[5] P. Rajpurkar et al., “SQuAD: 100,000+ Questions for Machine Comprehension of Text”, (2016).

and answers for each paragraph. In the first version of SQuAD, each answer to a question was guaranteed to exist in the corresponding passage. But it wasn't long before sequence models started performing better than humans at extracting the correct span of text with the answer. To make the task more difficult, SQuAD 2.0 was created by augmenting SQuAD 1.1 with a set of adversarial questions that are relevant to a given passage but cannot be answered from the text alone.[6] The state of the art as of this book's writing is shown in Figure 7-3, with most models since 2019 surpassing human performance.

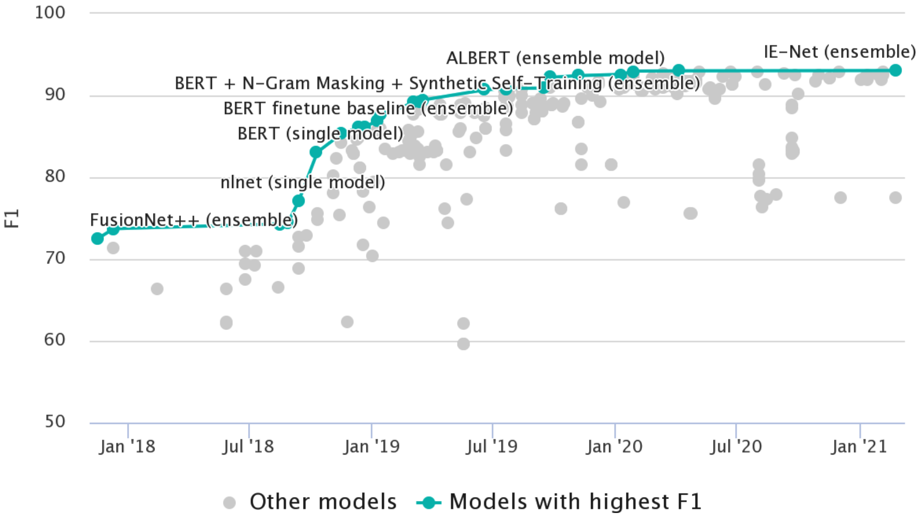


Figure 7-3. Progress on the SQuAD 2.0 benchmark (image from Papers with Code)

However, this superhuman performance does not appear to reflect genuine reading comprehension, since answers to the "unanswerable" questions can usually be identified through patterns in the passages like antonyms. To address these problems Google released the Natural Questions (NQ) dataset,[7] which involves fact-seeking questions obtained from Google Search users. The answers in NQ are much longer than in SQuAD and present a more challenging benchmark.

Now that we've explored our dataset a bit, let's dive into understanding how transformers can extract answers from text.

[6] P. Rajpurkar, R. Jia, and P. Liang, "Know What You Don't Know: Unanswerable Questions for SQuAD", (2018).

[7] T. Kwiatkowski et al., "Natural Questions: A Benchmark for Question Answering Research," *Transactions of the Association for Computational Linguistics* 7 (March 2019): 452–466, *http://dx.doi.org/10.1162/tacl_a_00276*.

Extracting Answers from Text

The first thing we'll need for our QA system is to find a way to identify a potential answer as a span of text in a customer review. For example, if a we have a question like "Is it waterproof?" and the review passage is "This watch is waterproof at 30m depth", then the model should output "waterproof at 30m". To do this we'll need to understand how to:

- Frame the supervised learning problem.
- Tokenize and encode text for QA tasks.
- Deal with long passages that exceed a model's maximum context size.

Let's start by taking a look at how to frame the problem.

Span classification

The most common way to extract answers from text is by framing the problem as a *span classification* task, where the start and end tokens of an answer span act as the labels that a model needs to predict. This process is illustrated in Figure 7-4.

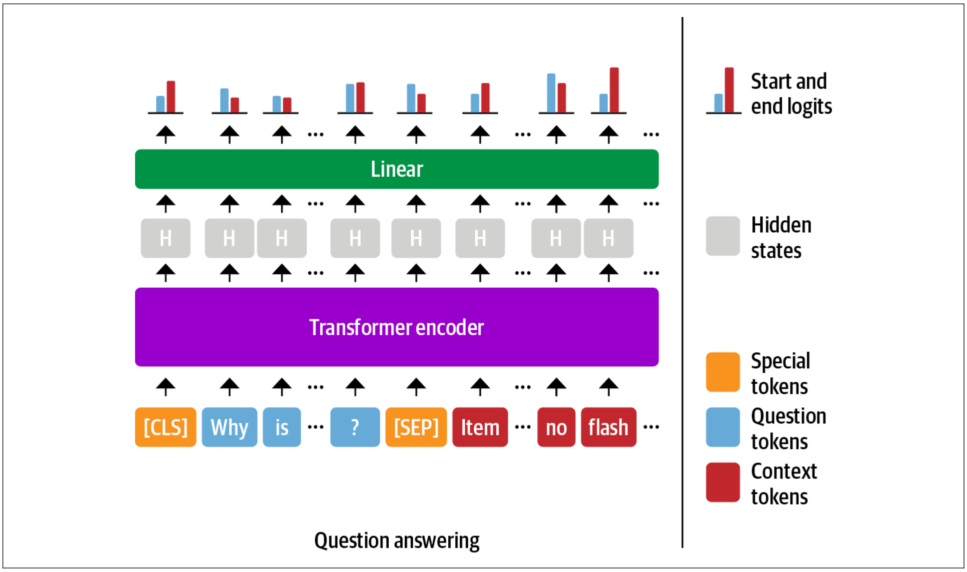
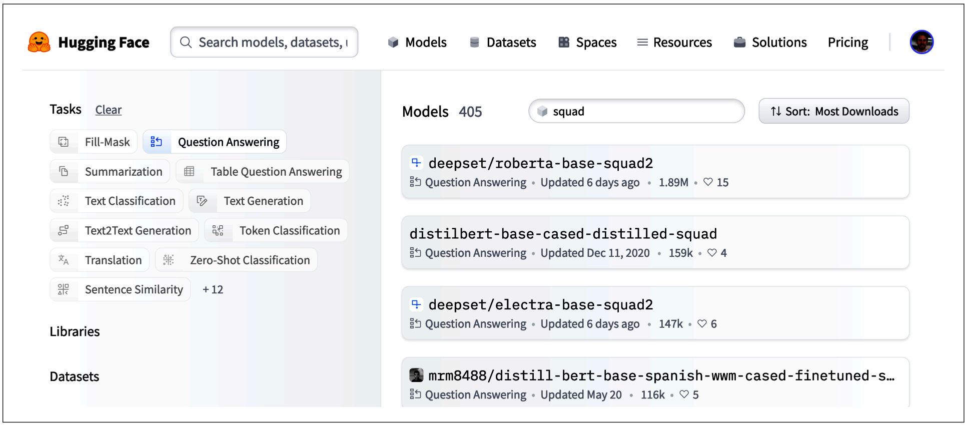


Figure 7-4. The span classification head for QA tasks

Since our training set is relatively small, with only 1,295 examples, a good strategy is to start with a language model that has already been fine-tuned on a large-scale QA dataset like SQuAD. In general, these models have strong reading comprehension capabilities and serve as a good baseline upon which to build a more accurate system. This is a somewhat different approach to that taken in previous chapters, where we typically started with a pretrained model and fine-tuned the task-specific head ourselves. For example, in Chapter 2, we had to fine-tune the classification head because the number of classes was tied to the dataset at hand. For extractive QA, we can actually start with a fine-tuned model since the structure of the labels remains the same across datasets.

You can find a list of extractive QA models by navigating to the Hugging Face Hub and searching for "squad" on the Models tab (Figure 7-5).



Models 405

squad

deepset/roberta-base-squad2
Question Answering • Updated 6 days ago • 1.89M • ♡ 15

distilbert-base-cased-distilled-squad
Question Answering • Updated Dec 11, 2020 • 159k • ♡ 4

deepset/electra-base-squad2
Question Answering • Updated 6 days ago • 147k • ♡ 6

mrm8488/distill-bert-base-spanish-wwm-cased-finetuned-s…
Question Answering • Updated May 20 • 116k • ♡ 5

Figure 7-5. A selection of extractive QA models on the Hugging Face Hub

As you can see, at the time of writing, there are more than 350 QA models to choose from—so which one should you pick? In general, the answer depends on various factors like whether your corpus is mono- or multilingual and the constraints of running the model in a production environment. Table 7-2 lists a few models that provide a good foundation to build on.

Table 7-2. Baseline transformer models that are fine-tuned on SQuAD 2.0

Transformer	Description	Number of parameters	F_1-score on SQuAD 2.0
MiniLM	A distilled version of BERT-base that preserves 99% of the performance while being twice as fast	66M	79.5
RoBERTa-base	RoBERTa models have better performance than their BERT counterparts and can be fine-tuned on most QA datasets using a single GPU	125M	83.0
ALBERT-XXL	State-of-the-art performance on SQuAD 2.0, but computationally intensive and difficult to deploy	235M	88.1
XLM-RoBERTa-large	Multilingual model for 100 languages with strong zero-shot performance	570M	83.8

For the purposes of this chapter, we'll use a fine-tuned MiniLM model since it is fast to train and will allow us to quickly iterate on the techniques that we'll be exploring.[8] As usual, the first thing we need is a tokenizer to encode our texts, so let's take a look at how this works for QA tasks.

Tokenizing text for QA

To encode our texts, we'll load the MiniLM model checkpoint from the Hugging Face Hub as usual:

```
from transformers import AutoTokenizer

model_ckpt = "deepset/minilm-uncased-squad2"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

To see the model in action, let's first try to extract an answer from a short passage of text. In extractive QA tasks, the inputs are provided as (question, context) pairs, so we pass them both to the tokenizer as follows:

```
question = "How much music can this hold?"
context = """An MP3 is about 1 MB/minute, so about 6000 hours depending on \
file size."""
inputs = tokenizer(question, context, return_tensors="pt")
```

Here we've returned PyTorch `Tensor` objects, since we'll need them to run the forward pass through the model. If we view the tokenized inputs as a table:

input_ids	101	2129	2172	2189	2064	2023	...	5834	2006	5371	2946	1012	102
token_type_ids	0	0	0	0	0	0	...	1	1	1	1	1	1
attention_mask	1	1	1	1	1	1	...	1	1	1	1	1	1

we can see the familiar `input_ids` and `attention_mask` tensors, while the `token_type_ids` tensor indicates which part of the inputs corresponds to the question and context (a 0 indicates a question token, a 1 indicates a context token).[9]

To understand how the tokenizer formats the inputs for QA tasks, let's decode the `input_ids` tensor:

```
print(tokenizer.decode(inputs["input_ids"][0]))
```

[8] W. Wang et al., "MINILM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers", (2020).

[9] Note that the `token_type_ids` are not present in all transformer models. In the case of BERT-like models such as MiniLM, the `token_type_ids` are also used during pretraining to incorporate the next sentence prediction task.

```
[CLS] how much music can this hold? [SEP] an mp3 is about 1 mb / minute, so
about 6000 hours depending on file size. [SEP]
```

We see that for each QA example, the inputs take the format:

```
[CLS] question tokens [SEP] context tokens [SEP]
```

where the location of the first `[SEP]` token is determined by the `token_type_ids`. Now that our text is tokenized, we just need to instantiate the model with a QA head and run the inputs through the forward pass:

```
import torch
from transformers import AutoModelForQuestionAnswering

model = AutoModelForQuestionAnswering.from_pretrained(model_ckpt)

with torch.no_grad():
    outputs = model(**inputs)
print(outputs)
```

```
QuestionAnsweringModelOutput(loss=None, start_logits=tensor([[-0.9862, -4.7750,
         -5.4025, -5.2378, -5.2863, -5.5117, -4.9819, -6.1880,
         -0.9862,  0.2596, -0.2144, -1.7136,  3.7806,  4.8561, -1.0546, -3.9097,
         -1.7374, -4.5944, -1.4278,  3.9949,  5.0390, -0.2018, -3.0193, -4.8549,
         -2.3107, -3.5110, -3.5713, -0.9862]]), end_logits=tensor([[-0.9623,
         -5.4733, -5.0326, -5.1639, -5.4278, -5.5151, -5.1749, -4.6233,
         -0.9623, -3.7855, -0.8715, -3.7745, -3.0161, -1.1780,  0.1758, -2.7365,
          4.8934,  0.3046, -3.1761, -3.2762,  0.8937,  5.6606, -0.3623, -4.9554,
         -3.2531, -0.0914,  1.6211, -0.9623]]), hidden_states=None,
attentions=None)
```

Here we can see that we get a `QuestionAnsweringModelOutput` object as the output of the QA head. As illustrated in Figure 7-4, the QA head corresponds to a linear layer that takes the hidden states from the encoder and computes the logits for the start and end spans.[10] This means that we treat QA as a form of token classification, similar to what we encountered for named entity recognition in Chapter 4. To convert the outputs into an answer span, we first need to get the logits for the start and end tokens:

```
start_logits = outputs.start_logits
end_logits = outputs.end_logits
```

If we compare the shapes of these logits to the input IDs:

```
print(f"Input IDs shape: {inputs.input_ids.size()}")
print(f"Start logits shape: {start_logits.size()}")
print(f"End logits shape: {end_logits.size()}")
```

[10] See Chapter 2 for details on how these hidden states can be extracted.

```
Input IDs shape: torch.Size([1, 28])
Start logits shape: torch.Size([1, 28])
End logits shape: torch.Size([1, 28])
```

we see that there are two logits (a start and end) associated with each input token. As illustrated in Figure 7-6, larger, positive logits correspond to more likely candidates for the start and end tokens. In this example we can see that the model assigns the highest start token logits to the numbers "1" and "6000", which makes sense since our question is asking about a quantity. Similarly, we see that the end tokens with the highest logits are "minute" and "hours".

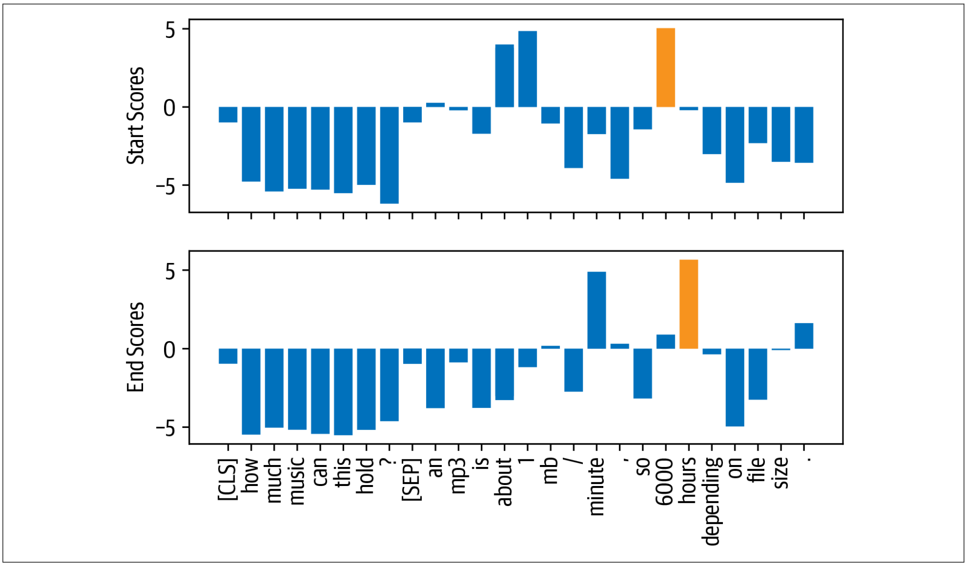


Figure 7-6. Predicted logits for the start and end tokens; the token with the highest score is colored in orange

To get the final answer, we can compute the argmax over the start and end token logits and then slice the span from the inputs. The following code performs these steps and decodes the result so we can print the resulting text:

```
import torch

start_idx = torch.argmax(start_logits)
end_idx = torch.argmax(end_logits) + 1
answer_span = inputs["input_ids"][0][start_idx:end_idx]
answer = tokenizer.decode(answer_span)
print(f"Question: {question}")
print(f"Answer: {answer}")
```

```
Question: How much music can this hold?
Answer: 6000 hours
```

Great, it worked! In 🤗 Transformers, all of these preprocessing and postprocessing steps are conveniently wrapped in a dedicated pipeline. We can instantiate the pipeline by passing our tokenizer and fine-tuned model as follows:

```
from transformers import pipeline

pipe = pipeline("question-answering", model=model, tokenizer=tokenizer)
pipe(question=question, context=context, topk=3)
```

```
[{'score': 0.26516005396842957,
  'start': 38,
  'end': 48,
  'answer': '6000 hours'},
 {'score': 0.2208300083875656,
  'start': 16,
  'end': 48,
  'answer': '1 MB/minute, so about 6000 hours'},
 {'score': 0.10253632068634033,
  'start': 16,
  'end': 27,
  'answer': '1 MB/minute'}]
```

In addition to the answer, the pipeline also returns the model's probability estimate in the `score` field (obtained by taking a softmax over the logits). This is handy when we want to compare multiple answers within a single context. We've also shown that we can have the model predict multiple answers by specifying the `topk` parameter. Sometimes, it is possible to have questions for which no answer is possible, like the empty `answers.answer_start` examples in SubjQA. In these cases the model will assign a high start and end score to the `[CLS]` token, and the pipeline maps this output to an empty string:

```
pipe(question="Why is there no data?", context=context,
     handle_impossible_answer=True)
```

```
{'score': 0.9068416357040405, 'start': 0, 'end': 0, 'answer': ''}
```

In our simple example, we obtained the start and end indices by taking the argmax of the corresponding logits. However, this heuristic can produce out-of-scope answers by selecting tokens that belong to the question instead of the context. In practice, the pipeline computes the best combination of start and end indices subject to various constraints such as being in-scope, requiring the start indices to precede the end indices, and so on.

Dealing with long passages

One subtlety faced by reading comprehension models is that the context often contains more tokens than the maximum sequence length of the model (which is usually a few hundred tokens at most). As illustrated in Figure 7-7, a decent portion of the SubjQA training set contains question-context pairs that won't fit within MiniLM's context size of 512 tokens.

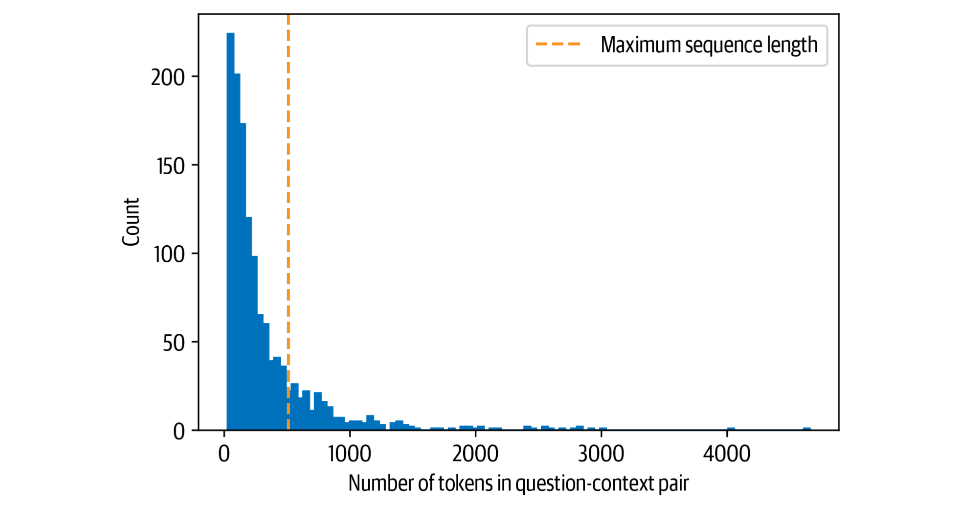


Figure 7-7. Distribution of tokens for each question-context pair in the SubjQA training set

For other tasks, like text classification, we simply truncated long texts under the assumption that enough information was contained in the embedding of the `[CLS]` token to generate accurate predictions. For QA, however, this strategy is problematic because the answer to a question could lie near the end of the context and thus would be removed by truncation. As illustrated in Figure 7-8, the standard way to deal with this is to apply a *sliding window* across the inputs, where each window contains a passage of tokens that fit in the model's context.

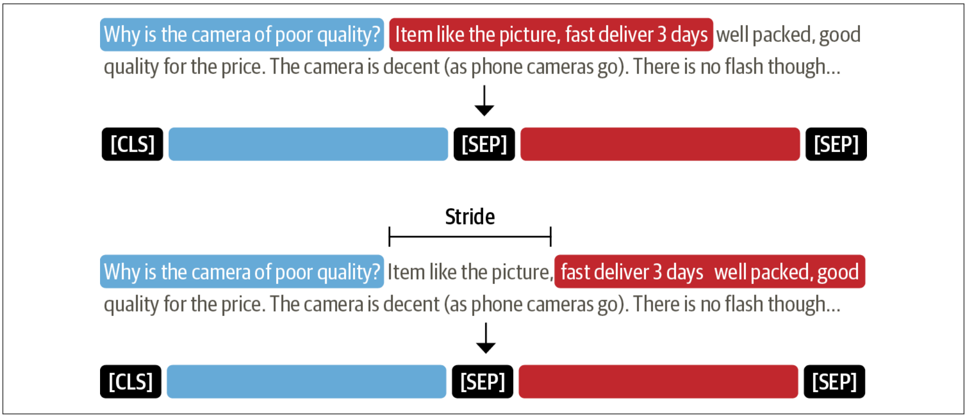

Why is the camera of poor quality?
Item like the picture, fast deliver 3 days well packed, good quality for the price. The camera is decent (as phone cameras go). There is no flash though...
[CLS]
[SEP]
[SEP]
Stride
Why is the camera of poor quality?
Item like the picture, fast deliver 3 days well packed, good quality for the price. The camera is decent (as phone cameras go). There is no flash though...
[CLS]
[SEP]
[SEP]

Figure 7-8. How the sliding window creates multiple question-context pairs for long documents—the first bar corresponds to the question, while the second bar is the context captured in each window

In 🤗 Transformers, we can set `return_overflowing_tokens=True` in the tokenizer to enable the sliding window. The size of the sliding window is controlled by the `max_seq_length` argument, and the size of the stride is controlled by `doc_stride`. Let's grab the first example from our training set and define a small window to illustrate how this works:

```
example = dfs["train"].iloc[0][["question", "context"]]
tokenized_example = tokenizer(example["question"], example["context"],
                              return_overflowing_tokens=True, max_length=100,
                              stride=25)
```

In this case we now get a list of `input_ids`, one for each window. Let's check the number of tokens we have in each window:

```
for idx, window in enumerate(tokenized_example["input_ids"]):
    print(f"Window #{idx} has {len(window)} tokens")

Window #0 has 100 tokens
Window #1 has 88 tokens
```

Finally, we can see where two windows overlap by decoding the inputs:

```
for window in tokenized_example["input_ids"]:
    print(f"{tokenizer.decode(window)} \n")

[CLS] how is the bass? [SEP] i have had koss headphones in the past, pro 4aa and
qz - 99. the koss portapro is portable and has great bass response. the work
great with my android phone and can be " rolled up " to be carried in my
motorcycle jacket or computer bag without getting crunched. they are very light
and don't feel heavy or bear down on your ears even after listening to music
with them on all day. the sound is [SEP]

[CLS] how is the bass? [SEP] and don't feel heavy or bear down on your ears even
```

```
after listening to music with them on all day. the sound is night and day better
than any ear - bud could be and are almost as good as the pro 4aa. they are "
open air " headphones so you cannot match the bass to the sealed types, but it
comes close. for $ 32, you cannot go wrong. [SEP]
```

Now that we have some intuition about how QA models can extract answers from text, let's look at the other components we need to build an end-to-end QA pipeline.

Using Haystack to Build a QA Pipeline

In our simple answer extraction example, we provided both the question and the context to the model. However, in reality our system's users will only provide a question about a product, so we need some way of selecting relevant passages from among all the reviews in our corpus. One way to do this would be to concatenate all the reviews of a given product together and feed them to the model as a single, long context. Although simple, the drawback of this approach is that the context can become extremely long and thereby introduce an unacceptable latency for our users' queries. For example, let's suppose that on average, each product has 30 reviews and each review takes 100 milliseconds to process. If we need to process all the reviews to get an answer, this would result in an average latency of 3 seconds per user query—much too long for ecommerce websites!

To handle this, modern QA systems are typically based on the *retriever-reader* architecture, which has two main components:

Retriever
    Responsible for retrieving relevant documents for a given query. Retrievers are usually categorized as *sparse* or *dense*. Sparse retrievers use word frequencies to represent each document and query as a sparse vector.[11] The relevance of a query and a document is then determined by computing an inner product of the vectors. On the other hand, dense retrievers use encoders like transformers to represent the query and document as contextualized embeddings (which are dense vectors). These embeddings encode semantic meaning, and allow dense retrievers to improve search accuracy by understanding the content of the query.

Reader
    Responsible for extracting an answer from the documents provided by the retriever. The reader is usually a reading comprehension model, although at the end of the chapter we'll see examples of models that can generate free-form answers.

[11] A vector is sparse if most of its elements are zero.

As illustrated in Figure 7-9, there can also be other components that apply postprocessing to the documents fetched by the retriever or to the answers extracted by the reader. For example, the retrieved documents may need reranking to eliminate noisy or irrelevant ones that can confuse the reader. Similarly, postprocessing of the reader's answers is often needed when the correct answer comes from various passages in a long document.

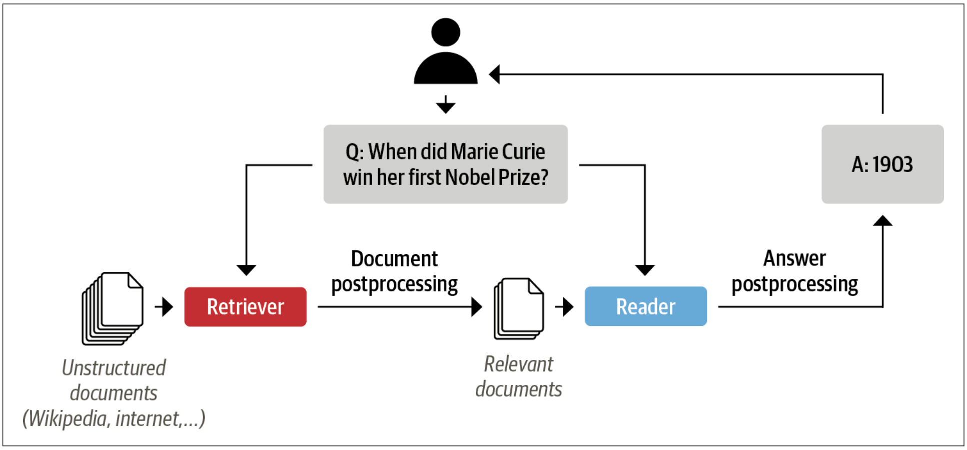

Q: When did Marie Curie win her first Nobel Prize?
A: 1903
Document postprocessing
Answer postprocessing
Retriever
Reader
Unstructured documents (Wikipedia, internet,...)
Relevant documents

Figure 7-9. The retriever-reader architecture for modern QA systems

To build our QA system, we'll use the *Haystack* library developed by deepset, a German company focused on NLP. Haystack is based on the retriever-reader architecture, abstracts much of the complexity involved in building these systems, and integrates tightly with 🤗 Transformers.

In addition to the retriever and reader, there are two more components involved when building a QA pipeline with Haystack:

Document store
: A document-oriented database that stores documents and metadata which are provided to the retriever at query time

Pipeline
: Combines all the components of a QA system to enable custom query flows, merging documents from multiple retrievers, and more

In this section we'll look at how we can use these components to quickly build a prototype QA pipeline. Later, we'll examine how we can improve its performance.

This chapter was written using version 0.9.0 of the Haystack library. In version 0.10.0, the pipeline and evaluation APIs were redesigned to make it easier to inspect whether the retriever or reader are impacting performance. To see what this chapter's code looks like with the new API, check out the GitHub repository.

Initializing a document store

In Haystack, there are various document stores to choose from and each one can be paired with a dedicated set of retrievers. This is illustrated in Table 7-3, where the compatibility of sparse (TF-IDF, BM25) and dense (Embedding, DPR) retrievers is shown for each of the available document stores. We'll explain what all these acronyms mean later in this chapter.

Table 7-3. Compatibility of Haystack retrievers and document stores

	In memory	Elasticsearch	FAISS	Milvus
TF-IDF	Yes	Yes	No	No
BM25	No	Yes	No	No
Embedding	Yes	Yes	Yes	Yes
DPR	Yes	Yes	Yes	Yes

Since we'll be exploring both sparse and dense retrievers in this chapter, we'll use the `ElasticsearchDocumentStore`, which is compatible with both retriever types. Elasticsearch is a search engine that is capable of handling a diverse range of data types, including textual, numerical, geospatial, structured, and unstructured. Its ability to store huge volumes of data and quickly filter it with full-text search features makes it especially well suited for developing QA systems. It also has the advantage of being the industry standard for infrastructure analytics, so there's a good chance your company already has a cluster that you can work with.

To initialize the document store, we first need to download and install Elasticsearch. By following Elasticsearch's guide,[12] we can grab the latest release for Linux with `wget` and unpack it with the `tar` shell command:

```
url = """https://artifacts.elastic.co/downloads/elasticsearch/\
elasticsearch-7.9.2-linux-x86_64.tar.gz"""
!wget -nc -q {url}
!tar -xzf elasticsearch-7.9.2-linux-x86_64.tar.gz
```

Next we need to start the Elasticsearch server. Since we're running all the code in this book within Jupyter notebooks, we'll need to use Python's `Popen()` function to spawn

[12] The guide also provides installation instructions for macOS and Windows.

a new process. While we're at it, let's also run the subprocess in the background using the chown shell command:

```
import os
from subprocess import Popen, PIPE, STDOUT

# Run Elasticsearch as a background process
!chown -R daemon:daemon elasticsearch-7.9.2
es_server = Popen(args=['elasticsearch-7.9.2/bin/elasticsearch'],
                  stdout=PIPE, stderr=STDOUT, preexec_fn=lambda: os.setuid(1))
# Wait until Elasticsearch has started
!sleep 30
```

In the `Popen()` function, the `args` specify the program we wish to execute, while `stdout=PIPE` creates a new pipe for the standard output and `stderr=STDOUT` collects the errors in the same pipe. The `preexec_fn` argument specifies the ID of the subprocess we wish to use. By default, Elasticsearch runs locally on port 9200, so we can test the connection by sending an HTTP request to `localhost`:

```
!curl -X GET "localhost:9200/?pretty"
```

```
{
  "name" : "96938eee37cd",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "ABGDdvbbRWmMb9Umz79HbA",
  "version" : {
    "number" : "7.9.2",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "d34da0ea4a966c4e49417f2da2f244e3e97b4e6e",
    "build_date" : "2020-09-23T00:45:33.626720Z",
    "build_snapshot" : false,
    "lucene_version" : "8.6.2",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Now that our Elasticsearch server is up and running, the next thing to do is instantiate the document store:

```
from haystack.document_store.elasticsearch import ElasticsearchDocumentStore

# Return the document embedding for later use with dense retriever
document_store = ElasticsearchDocumentStore(return_embedding=True)
```

By default, `ElasticsearchDocumentStore` creates two indices on Elasticsearch: one called `document` for (you guessed it) storing documents, and another called `label` for storing the annotated answer spans. For now, we'll just populate the `document` index with the SubjQA reviews, and Haystack's document stores expect a list of dictionaries with `text` and `meta` keys as follows:

```
{
    "text": "<the-context>",
    "meta": {
        "field_01": "<additional-metadata>",
        "field_02": "<additional-metadata>",
        ...
    }
}
```

The fields in `meta` can be used for applying filters during retrieval. For our purposes we'll include the `item_id` and `q_review_id` columns of SubjQA so we can filter by product and question ID, along with the corresponding training split. We can then loop through the examples in each `DataFrame` and add them to the index with the `write_documents()` method as follows:

```python
for split, df in dfs.items():
    # Exclude duplicate reviews
    docs = [{"text": row["context"],
             "meta":{"item_id": row["title"], "question_id": row["id"],
                     "split": split}}
        for _,row in df.drop_duplicates(subset="context").iterrows()]
    document_store.write_documents(docs, index="document")

print(f"Loaded {document_store.get_document_count()} documents")
```

```
Loaded 1615 documents
```

Great, we've loaded all our reviews into an index! To search the index we'll need a retriever, so let's look at how we can initialize one for Elasticsearch.

Initializing a retriever

The Elasticsearch document store can be paired with any of the Haystack retrievers, so let's start by using a sparse retriever based on BM25 (short for "Best Match 25"). BM25 is an improved version of the classic Term Frequency-Inverse Document Frequency (TF-IDF) algorithm and represents the question and context as sparse vectors that can be searched efficiently on Elasticsearch. The BM25 score measures how much matched text is about a search query and improves on TF-IDF by saturating TF values quickly and normalizing the document length so that short documents are favored over long ones.[13]

[13] For an in-depth explanation of document scoring with TF-IDF and BM25 see Chapter 23 of *Speech and Language Processing*, 3rd edition, by D. Jurafsky and J.H. Martin (Prentice Hall).

In Haystack, the BM25 retriever is used by default in `ElasticsearchRetriever`, so let's initialize this class by specifying the document store we wish to search over:

```
from haystack.retriever.sparse import ElasticsearchRetriever

es_retriever = ElasticsearchRetriever(document_store=document_store)
```

Next, let's look at a simple query for a single electronics product in the training set. For review-based QA systems like ours, it's important to restrict the queries to a single item because otherwise the retriever would source reviews about products that are not related to a user's query. For example, asking "Is the camera quality any good?" without a product filter could return reviews about phones, when the user might be asking about a specific laptop camera instead. By themselves, the ASIN values in our dataset are a bit cryptic, but we can decipher them with online tools like *amazon ASIN* or by simply appending the value of `item_id` to the *www.amazon.com/dp/* URL. The following item ID corresponds to one of Amazon's Fire tablets, so let's use the retriever's `retrieve()` method to ask if it's any good for reading with:

```
item_id = "B0074BW614"
query = "Is it good for reading?"
retrieved_docs = es_retriever.retrieve(
    query=query, top_k=3, filters={"item_id":[item_id], "split":["train"]})
```

Here we've specified how many documents to return with the `top_k` argument and applied a filter on both the `item_id` and `split` keys that were included in the `meta` field of our documents. Each element of `retrieved_docs` is a Haystack `Document` object that is used to represent documents and includes the retriever's query score along with other metadata. Let's have a look at one of the retrieved documents:

```
print(retrieved_docs[0])
```

```
{'text': 'This is a gift to myself.  I have been a kindle user for 4 years and
this is my third one.  I never thought I would want a fire for I mainly use it
for book reading.  I decided to try the fire for when I travel I take my laptop,
my phone and my iPod classic.  I love my iPod but watching movies on the plane
with it can be challenging because it is so small. Laptops battery life is not
as good as the Kindle.  So the Fire combines for me what I needed all three to
do. So far so good.', 'score': 6.243799, 'probability': 0.6857824513476455,
'question': None, 'meta': {'item_id': 'B0074BW614', 'question_id':
'868e311275e26dbafe5af70774a300f3', 'split': 'train'}, 'embedding': None, 'id':
'252e83e25d52df7311d597dc89eef9f6'}
```

In addition to the document's text, we can see the `score` that Elasticsearch computed for its relevance to the query (larger scores imply a better match). Under the hood, Elasticsearch relies on Lucene for indexing and search, so by default it uses Lucene's *practical scoring function*. You can find the nitty-gritty details behind the scoring function in the Elasticsearch documentation, but in brief terms it first filters the candidate documents by applying a Boolean test (does the document match the query?), and then applies a similarity metric that's based on representing both the document and the query as vectors.

Now that we have a way to retrieve relevant documents, the next thing we need is a way to extract answers from them. This is where the reader comes in, so let's take a look at how we can load our MiniLM model in Haystack.

Initializing a reader

In Haystack, there are two types of readers one can use to extract answers from a given context:

`FARMReader`
: Based on deepset's *FARM* framework for fine-tuning and deploying transformers. Compatible with models trained using 🤗 Transformers and can load models directly from the Hugging Face Hub.

`TransformersReader`
: Based on the QA pipeline from 🤗 Transformers. Suitable for running inference only.

Although both readers handle a model's weights in the same way, there are some differences in the way the predictions are converted to produce answers:

- In 🤗 Transformers, the QA pipeline normalizes the start and end logits with a softmax in each passage. This means that it is only meaningful to compare answer scores between answers extracted from the same passage, where the probabilities sum to 1. For example, an answer score of 0.9 from one passage is not necessarily better than a score of 0.8 in another. In FARM, the logits are not normalized, so inter-passage answers can be compared more easily.
- The `TransformersReader` sometimes predicts the same answer twice, but with different scores. This can happen in long contexts if the answer lies across two overlapping windows. In FARM, these duplicates are removed.

Since we will be fine-tuning the reader later in the chapter, we'll use the `FARMReader`. As with 🤗 Transformers, to load the model we just need to specify the MiniLM checkpoint on the Hugging Face Hub along with some QA-specific arguments:

```
from haystack.reader.farm import FARMReader

model_ckpt = "deepset/minilm-uncased-squad2"
max_seq_length, doc_stride = 384, 128
reader = FARMReader(model_name_or_path=model_ckpt, progress_bar=False,
                    max_seq_len=max_seq_length, doc_stride=doc_stride,
                    return_no_answer=True)
```

It is also possible to fine-tune a reading comprehension model directly in 🤗 Transformers and then load it in `Transformers Reader` to run inference. For details on how to do the fine-tuning step, see the question answering tutorial in the library's documentation.

In `FARMReader`, the behavior of the sliding window is controlled by the same `max_seq_length` and `doc_stride` arguments that we saw for the tokenizer. Here we've used the values from the MiniLM paper. To confirm, let's now test the reader on our simple example from earlier:

```
print(reader.predict_on_texts(question=question, texts=[context], top_k=1))
```

```
{'query': 'How much music can this hold?', 'no_ans_gap': 12.648084878921509,
'answers': [{'answer': '6000 hours', 'score': 10.69961929321289, 'probability':
0.3988136053085327, 'context': 'An MP3 is about 1 MB/minute, so about 6000 hours
depending on file size.', 'offset_start': 38, 'offset_end': 48,
'offset_start_in_doc': 38, 'offset_end_in_doc': 48, 'document_id':
'e344757014e804eff50faa3ecf1c9c75'}]}
```

Great, the reader appears to be working as expected—so next, let's tie together all our components using one of Haystack's pipelines.

Putting it all together

Haystack provides a `Pipeline` abstraction that allows us to combine retrievers, readers, and other components together as a graph that can be easily customized for each use case. There are also predefined pipelines analogous to those in 🤗 Transformers, but specialized for QA systems. In our case, we're interested in extracting answers, so we'll use the `ExtractiveQAPipeline`, which takes a single retriever-reader pair as its arguments:

```
from haystack.pipeline import ExtractiveQAPipeline

pipe = ExtractiveQAPipeline(reader, es_retriever)
```

Each `Pipeline` has a `run()` method that specifies how the query flow should be executed. For the `ExtractiveQAPipeline` we just need to pass the `query`, the number of documents to retrieve with `top_k_retriever`, and the number of answers to extract from these documents with `top_k_reader`. In our case, we also need to specify a filter over the item ID, which can be done using the `filters` argument as we did with the retriever earlier. Let's run a simple example using our question about the Amazon Fire tablet again, but this time returning the extracted answers:

```
n_answers = 3
preds = pipe.run(query=query, top_k_retriever=3, top_k_reader=n_answers,
                 filters={"item_id": [item_id], "split":["train"]})

print(f"Question: {preds['query']} \n")
```

```
for idx in range(n_answers):
    print(f"Answer {idx+1}: {preds['answers'][idx]['answer']}")
    print(f"Review snippet: ...{preds['answers'][idx]['context']}...")
    print("\n\n")
```

```
Question: Is it good for reading?

Answer 1: I mainly use it for book reading
Review snippet: ... is my third one.  I never thought I would want a fire for I
mainly use it for book reading.  I decided to try the fire for when I travel I
take my la...


Answer 2: the larger screen compared to the Kindle makes for easier reading
Review snippet: ...ght enough that I can hold it to read, but the larger screen
compared to the Kindle makes for easier reading. I love the color, something I
never thou...


Answer 3: it is great for reading books when no light is available
Review snippet: ...ecoming addicted to hers! Our son LOVES it and it is great
for reading books when no light is available. Amazing sound but I suggest good
headphones t...
```

Great, we now have an end-to-end QA system for Amazon product reviews! This is a good start, but notice that the second and third answers are closer to what the question is actually asking. To do better, we'll need some metrics to quantify the performance of the retriever and reader. We'll take a look at that next.

Improving Our QA Pipeline

Although much of the recent research on QA has focused on improving reading comprehension models, in practice it doesn't matter how good your reader is if the retriever can't find the relevant documents in the first place! In particular, the retriever sets an upper bound on the performance of the whole QA system, so it's important to make sure it's doing a good job. With this in mind, let's start by introducing some common metrics to evaluate the retriever so that we can compare the performance of sparse and dense representations.

Evaluating the Retriever

A common metric for evaluating retrievers is *recall*, which measures the fraction of all relevant documents that are retrieved. In this context, "relevant" simply means whether the answer is present in a passage of text or not, so given a set of questions, we can compute recall by counting the number of times an answer appears in the top *k* documents returned by the retriever.

In Haystack, there are two ways to evaluate retrievers:

- Use the retriever's in-built `eval()` method. This can be used for both open- and closed-domain QA, but not for datasets like SubjQA where each document is paired with a single product and we need to filter by product ID for every query.
- Build a custom `Pipeline` that combines a retriever with the `EvalRetriever` class. This enables the implementation of custom metrics and query flows.

A complementary metric to recall is *mean average precision* (mAP), which rewards retrievers that can place the correct answers higher up in the document ranking.

Since we need to evaluate the recall per product and then aggregate across all products, we'll opt for the second approach. Each node in the `Pipeline` graph represents a class that takes some inputs and produces some outputs via a `run()` method:

```
class PipelineNode:
    def __init__(self):
        self.outgoing_edges = 1

    def run(self, **kwargs):
        ...
        return (outputs, "outgoing_edge_name")
```

Here `kwargs` corresponds to the outputs from the previous node in the graph, which is manipulated within the `run()` method to return a tuple of the outputs for the next node, along with a name for the outgoing edge. The only other requirement is to include an `outgoing_edges` attribute that indicates the number of outputs from the node (in most cases `outgoing_edges=1`, unless you have branches in the pipeline that route the inputs according to some criterion).

In our case, we need a node to evaluate the retriever, so we'll use the `EvalRetriever` class whose `run()` method keeps track of which documents have answers that match the ground truth. With this class we can then build up a `Pipeline` graph by adding the evaluation node after a node that represents the retriever itself:

```
from haystack.pipeline import Pipeline
from haystack.eval import EvalDocuments

class EvalRetrieverPipeline:
    def __init__(self, retriever):
        self.retriever = retriever
        self.eval_retriever = EvalDocuments()
        pipe = Pipeline()
        pipe.add_node(component=self.retriever, name="ESRetriever",
                      inputs=["Query"])
        pipe.add_node(component=self.eval_retriever, name="EvalRetriever",
                      inputs=["ESRetriever"])
        self.pipeline = pipe


pipe = EvalRetrieverPipeline(es_retriever)
```

Notice that each node is given a `name` and a list of `inputs`. In most cases, each node has a single outgoing edge, so we just need to include the name of the previous node in `inputs`.

Now that we have our evaluation pipeline, we need to pass some queries and their corresponding answers. To do this, we'll add the answers to a dedicated `label` index on our document store. Haystack provides a `Label` object that represents the answer spans and their metadata in a standardized fashion. To populate the `label` index, we'll first create a list of `Label` objects by looping over each question in the test set and extracting the matching answers and additional metadata:

```
from haystack import Label

labels = []
for i, row in dfs["test"].iterrows():
    # Metadata used for filtering in the Retriever
    meta = {"item_id": row["title"], "question_id": row["id"]}
    # Populate labels for questions with answers
    if len(row["answers.text"]):
        for answer in row["answers.text"]:
            label = Label(
                question=row["question"], answer=answer, id=i, origin=row["id"],
                meta=meta, is_correct_answer=True, is_correct_document=True,
                no_answer=False)
            labels.append(label)
    # Populate labels for questions without answers
    else:
        label = Label(
            question=row["question"], answer="", id=i, origin=row["id"],
            meta=meta, is_correct_answer=True, is_correct_document=True,
            no_answer=True)
        labels.append(label)
```

If we peek at one of these labels:

```
print(labels[0])
```

```
{'id': 'e28f5e62-85e8-41b2-8a34-fbff63b7a466', 'created_at': None, 'updated_at':
None, 'question': 'What is the tonal balance of these headphones?', 'answer': 'I
have been a headphone fanatic for thirty years', 'is_correct_answer': True,
'is_correct_document': True, 'origin': 'd0781d13200014aa25860e44da9d5ea7',
'document_id': None, 'offset_start_in_doc': None, 'no_answer': False,
'model_id': None, 'meta': {'item_id': 'B00001WRSJ', 'question_id':
'd0781d13200014aa25860e44da9d5ea7'}}
```

we can see the question-answer pair, along with an origin field that contains the unique question ID so we can filter the document store per question. We've also added the product ID to the meta field so we can filter the labels by product. Now that we have our labels, we can write them to the label index on Elasticsearch as follows:

```
document_store.write_labels(labels, index="label")
print(f"""Loaded {document_store.get_label_count(index="label")} \
question-answer pairs""")
```

```
Loaded 358 question-answer pairs
```

Next, we need to build up a mapping between our question IDs and corresponding answers that we can pass to the pipeline. To get all the labels, we can use the get_all_labels_aggregated() method from the document store that will aggregate all question-answer pairs associated with a unique ID. This method returns a list of MultiLabel objects, but in our case we only get one element since we're filtering by question ID. We can build up a list of aggregated labels as follows:

```
labels_agg = document_store.get_all_labels_aggregated(
    index="label",
    open_domain=True,
    aggregate_by_meta=["item_id"]
)
print(len(labels_agg))
```

```
330
```

By peeking at one of these labels we can see that all the answers associated with a given question are aggregated together in a multiple_answers field:

```
print(labels_agg[109])
```

```
{'question': 'How does the fan work?', 'multiple_answers': ['the fan is really
really good', "the fan itself isn't super loud. There is an adjustable dial to
change fan speed"], 'is_correct_answer': True, 'is_correct_document': True,
'origin': '5a9b7616541f700f103d21f8ad41bc4b', 'multiple_document_ids': [None,
None], 'multiple_offset_start_in_docs': [None, None], 'no_answer': False,
'model_id': None, 'meta': {'item_id': 'B002MU1ZRS'}}
```

We now have all the ingredients for evaluating the retriever, so let's define a function that feeds each question-answer pair associated with each product to the evaluation pipeline and tracks the correct retrievals in our pipe object:

```
def run_pipeline(pipeline, top_k_retriever=10, top_k_reader=4):
    for l in labels_agg:
        _ = pipeline.pipeline.run(
            query=l.question,
            top_k_retriever=top_k_retriever,
            top_k_reader=top_k_reader,
            top_k_eval_documents=top_k_retriever,
            labels=l,
            filters={"item_id": [l.meta["item_id"]], "split": ["test"]})
```

```
run_pipeline(pipe, top_k_retriever=3)
print(f"Recall@3: {pipe.eval_retriever.recall:.2f}")
```

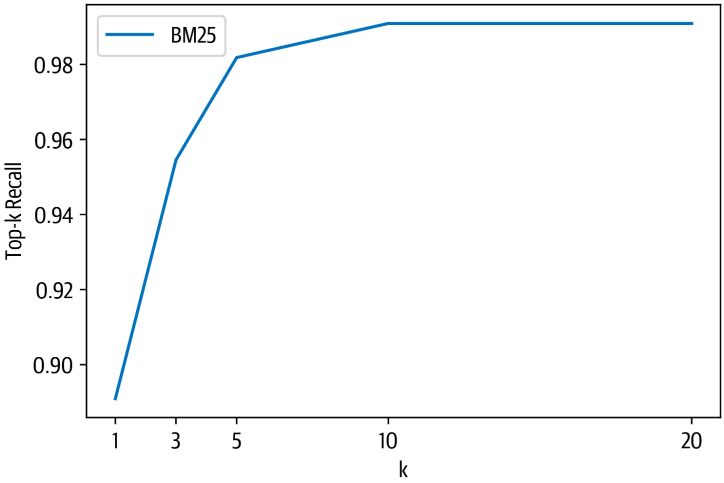
```
Recall@3: 0.95
```

Great, it works! Notice that we picked a specific value for `top_k_retriever` to specify the number of documents to retrieve. In general, increasing this parameter will improve the recall, but at the expense of providing more documents to the reader and slowing down the end-to-end pipeline. To guide our decision on which value to pick, we'll create a function that loops over several *k* values and compute the recall across the whole test set for each *k*:

```
def evaluate_retriever(retriever, topk_values = [1,3,5,10,20]):
    topk_results = {}

    for topk in topk_values:
        # Create Pipeline
        p = EvalRetrieverPipeline(retriever)
        # Loop over each question-answers pair in test set
        run_pipeline(p, top_k_retriever=topk)
        # Get metrics
        topk_results[topk] = {"recall": p.eval_retriever.recall}

    return pd.DataFrame.from_dict(topk_results, orient="index")


es_topk_df = evaluate_retriever(es_retriever)
```

If we plot the results, we can see how the recall improves as we increase *k*:

```
def plot_retriever_eval(dfs, retriever_names):
    fig, ax = plt.subplots()
    for df, retriever_name in zip(dfs, retriever_names):
        df.plot(y="recall", ax=ax, label=retriever_name)
    plt.xticks(df.index)
    plt.ylabel("Top-k Recall")
    plt.xlabel("k")
    plt.show()

plot_retriever_eval([es_topk_df], ["BM25"])
```



From the plot, we can see that there's an inflection point around $k = 5$ and we get almost perfect recall from $k = 10$ onwards. Let's now take a look at retrieving documents with dense vector techniques.

Dense Passage Retrieval

We've seen that we get almost perfect recall when our sparse retriever returns $k = 10$ documents, but can we do better at smaller values of k? The advantage of doing so is that we can pass fewer documents to the reader and thereby reduce the overall latency of our QA pipeline. A well-known limitation of sparse retrievers like BM25 is that they can fail to capture the relevant documents if the user query contains terms that don't match exactly those of the review. One promising alternative is to use dense embeddings to represent the question and document, and the current state of the art is an architecture known as *Dense Passage Retrieval* (DPR).[14] The main idea behind DPR is to use two BERT models as encoders for the question and the passage. As illustrated in Figure 7-10, these encoders map the input text into a d-dimensional vector representation of the `[CLS]` token.

[14] V. Karpukhin et al., "Dense Passage Retrieval for Open-Domain Question Answering", (2020).

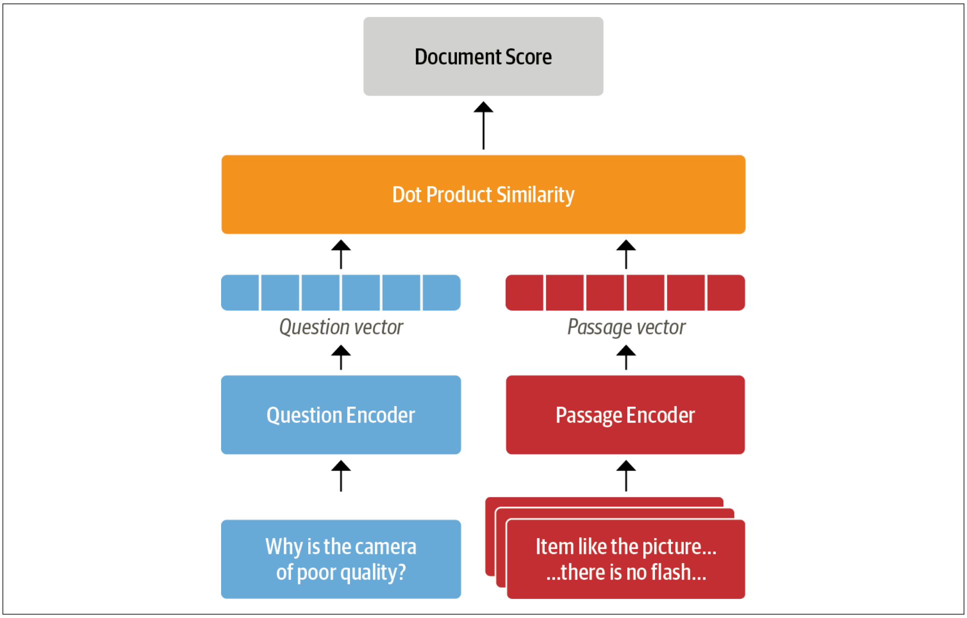


Figure 7-10. DPR's bi-encoder architecture for computing the relevance of a document and query

In Haystack, we can initialize a retriever for DPR in a similar way to what we did for BM25. In addition to specifying the document store, we also need to pick the BERT encoders for the question and passage. These encoders are trained by giving them questions with relevant (positive) passages and irrelevant (negative) passages, where the goal is to learn that relevant question-passage pairs have a higher similarity. For our use case, we'll use encoders that have been fine-tuned on the NQ corpus in this way:

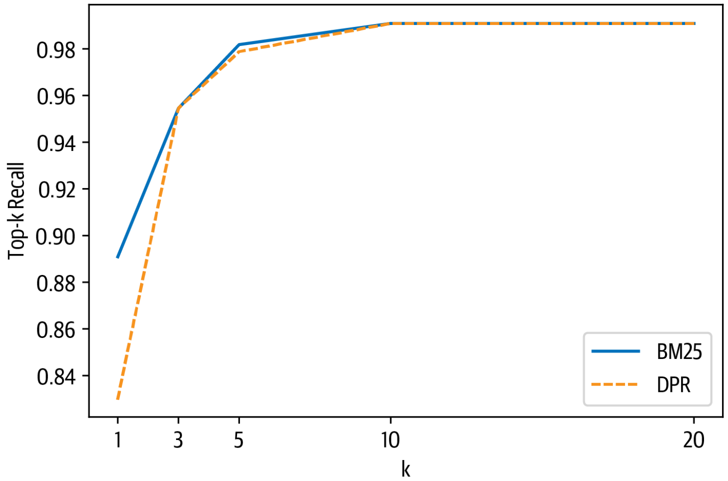
```
from haystack.retriever.dense import DensePassageRetriever

dpr_retriever = DensePassageRetriever(document_store=document_store,
    query_embedding_model="facebook/dpr-question_encoder-single-nq-base",
    passage_embedding_model="facebook/dpr-ctx_encoder-single-nq-base",
    embed_title=False)
```

Here we've also set `embed_title=False` since concatenating the document's title (i.e., `item_id`) doesn't provide any additional information because we filter per product. Once we've initialized the dense retriever, the next step is to iterate over all the indexed documents in our Elasticsearch index and apply the encoders to update the embedding representation. This can be done as follows:

```
document_store.update_embeddings(retriever=dpr_retriever)
```

We're now set to go! We can evaluate the dense retriever in the same way we did for BM25 and compare the top-*k* recall:

```
dpr_topk_df = evaluate_retriever(dpr_retriever)
plot_retriever_eval([es_topk_df, dpr_topk_df], ["BM25", "DPR"])
```



Here we can see that DPR does not provide a boost in recall over BM25 and saturates around *k* = 3.

Performing similarity search of the embeddings can be sped up by using Facebook's FAISS library as the document store. Similarly, the performance of the DPR retriever can be improved by fine-tuning on the target domain. If you'd like to learn how to fine-tune DPR, check out the Haystack tutorial.

Now that we've explored the evaluation of the retriever, let's turn to evaluating the reader.

Evaluating the Reader

In extractive QA, there are two main metrics that are used for evaluating readers:

Exact Match (EM)
A binary metric that gives EM = 1 if the characters in the predicted and ground truth answers match exactly, and EM = 0 otherwise. If no answer is expected, the model gets EM = 0 if it predicts any text at all.

F_1-score
Measures the harmonic mean of the precision and recall.

Let's see how these metrics work by importing some helper functions from FARM and applying them to a simple example:

```
from farm.evaluation.squad_evaluation import compute_f1, compute_exact

pred = "about 6000 hours"
label = "6000 hours"
print(f"EM: {compute_exact(label, pred)}")
print(f"F1: {compute_f1(label, pred)}")
```

```
EM: 0
F1: 0.8
```

Under the hood, these functions first normalize the prediction and label by removing punctuation, fixing whitespace, and converting to lowercase. The normalized strings are then tokenized as a bag-of-words, before finally computing the metric at the token level. From this simple example we can see that EM is a much stricter metric than the F_1-score: adding a single token to the prediction gives an EM of zero. On the other hand, the F_1-score can fail to catch truly incorrect answers. For example, if our predicted answer span is "about 6000 dollars", then we get:

```
pred = "about 6000 dollars"
print(f"EM: {compute_exact(label, pred)}")
print(f"F1: {compute_f1(label, pred)}")
```

```
EM: 0
F1: 0.4
```

Relying on just the F_1-score is thus misleading, and tracking both metrics is a good strategy to balance the trade-off between underestimating (EM) and overestimating (F_1-score) model performance.

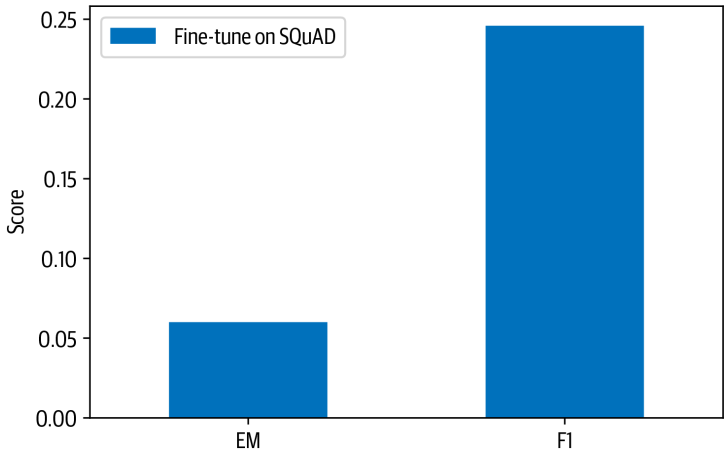
Now in general, there are multiple valid answers per question, so these metrics are calculated for each question-answer pair in the evaluation set, and the best score is selected over all possible answers. The overall EM and F_1 scores for the model are then obtained by averaging over the individual scores of each question-answer pair.

To evaluate the reader we'll create a new pipeline with two nodes: a reader node and a node to evaluate the reader. We'll use the `EvalReader` class that takes the predictions from the reader and computes the corresponding EM and F_1 scores. To compare with the SQuAD evaluation, we'll take the best answers for each query with the `top_1_em` and `top_1_f1` metrics that are stored in `EvalAnswers`:

```
from haystack.eval import EvalAnswers

def evaluate_reader(reader):
    score_keys = ['top_1_em', 'top_1_f1']
    eval_reader = EvalAnswers(skip_incorrect_retrieval=False)
    pipe = Pipeline()
    pipe.add_node(component=reader, name="QAReader", inputs=["Query"])
    pipe.add_node(component=eval_reader, name="EvalReader", inputs=["QAReader"])

    for l in labels_agg:
        doc = document_store.query(l.question,
                                   filters={"question_id":[l.origin]})
        _ = pipe.run(query=l.question, documents=doc, labels=l)

    return {k:v for k,v in eval_reader.__dict__.items() if k in score_keys}

reader_eval = {}
reader_eval["Fine-tune on SQuAD"] = evaluate_reader(reader)
```

Notice that we specified `skip_incorrect_retrieval=False`. This is to ensure that the retriever always passes the context to the reader (as in the SQuAD evaluation). Now that we've run every question through the reader, let's print the scores:

```
def plot_reader_eval(reader_eval):
    fig, ax = plt.subplots()
    df = pd.DataFrame.from_dict(reader_eval)
    df.plot(kind="bar", ylabel="Score", rot=0, ax=ax)
    ax.set_xticklabels(["EM", "F1"])
    plt.legend(loc='upper left')
    plt.show()

plot_reader_eval(reader_eval)
```



OK, it seems that the fine-tuned model performs significantly worse on SubjQA than on SQuAD 2.0, where MiniLM achieves EM and F_1 scores of 76.1 and 79.5, respectively. One reason for the performance drop is that customer reviews are quite different from the Wikipedia articles the SQuAD 2.0 dataset is generated from, and the language they use is often informal. Another factor is likely the inherent subjectivity of our dataset, where both questions and answers differ from the factual information contained in Wikipedia. Let's look at how to fine-tune a model on a dataset to get better results with domain adaptation.

Domain Adaptation

Although models that are fine-tuned on SQuAD will often generalize well to other domains, we've seen that for SubjQA the EM and F_1 scores of our model were much worse than for SQuAD. This failure to generalize has also been observed in other extractive QA datasets and is understood as evidence that transformer models are particularly adept at overfitting to SQuAD.[15] The most straightforward way to improve the reader is by fine-tuning our MiniLM model further on the SubjQA training set. The `FARMReader` has a `train()` method that is designed for this purpose and expects the data to be in SQuAD JSON format, where all the question-answer pairs are grouped together for each item as illustrated in Figure 7-11.

[15] D. Yogatama et al., "Learning and Evaluating General Linguistic Intelligence", (2019).

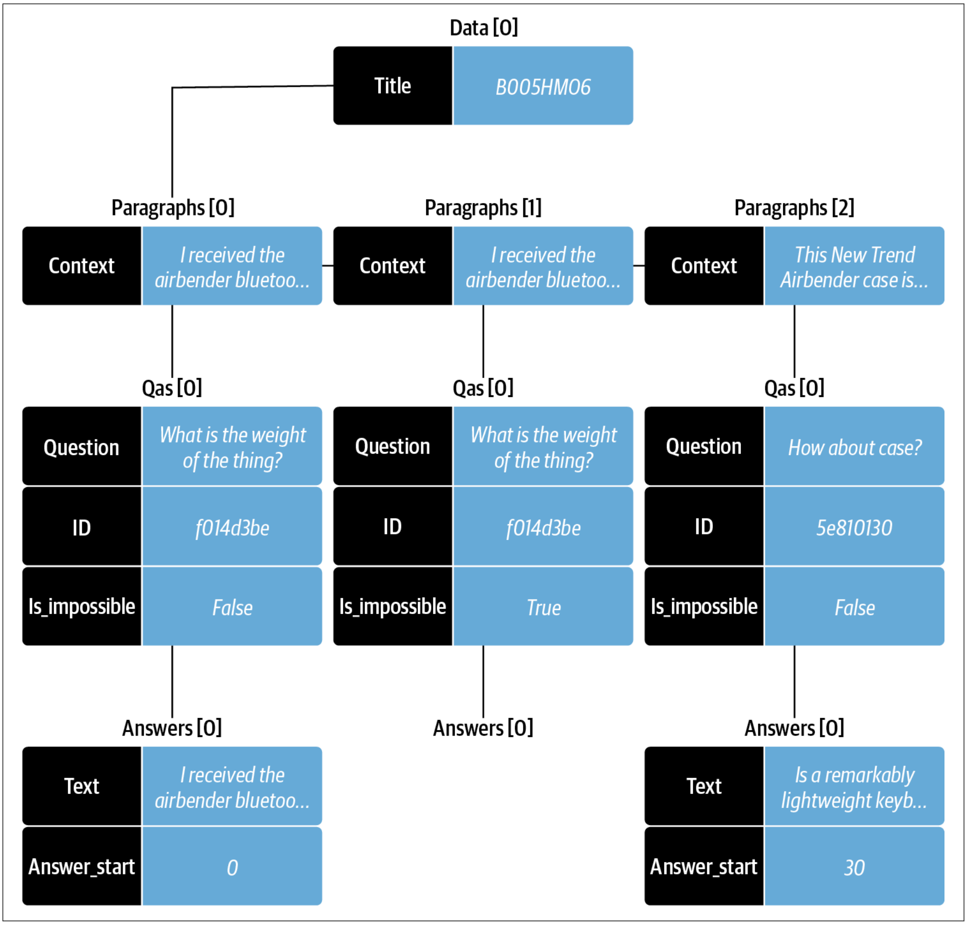


Figure 7-11. Visualization of the SQuAD JSON format

This is quite a complex data format, so we'll need a few functions and some Pandas magic to help us do the conversion. The first thing we need to do is implement a function that can create the `paragraphs` array associated with each product ID. Each element in this array contains a single context (i.e., review) and a `qas` array of question-answer pairs. Here's a function that builds up the `paragraphs` array:

```python
def create_paragraphs(df):
    paragraphs = []
    id2context = dict(zip(df["review_id"], df["context"]))
    for review_id, review in id2context.items():
        qas = []
        # Filter for all question-answer pairs about a specific context
        review_df = df.query(f"review_id == '{review_id}'")
        id2question = dict(zip(review_df["id"], review_df["question"]))
        # Build up the qas array
        for qid, question in id2question.items():
            # Filter for a single question ID
            question_df = df.query(f"id == '{qid}'").to_dict(orient="list")
            ans_start_idxs = question_df["answers.answer_start"][0].tolist()
            ans_text = question_df["answers.text"][0].tolist()
            # Fill answerable questions
            if len(ans_start_idxs):
                answers = [
                    {"text": text, "answer_start": answer_start}
                    for text, answer_start in zip(ans_text, ans_start_idxs)]
                is_impossible = False
            else:
                answers = []
                is_impossible = True
            # Add question-answer pairs to qas
            qas.append({"question": question, "id": qid,
                        "is_impossible": is_impossible, "answers": answers})
        # Add context and question-answer pairs to paragraphs
        paragraphs.append({"qas": qas, "context": review})
    return paragraphs
```

Now, when we apply to the rows of a `DataFrame` associated with a single product ID, we get the SQuAD format:

```
product = dfs["train"].query("title == 'B00001P4ZH'")
create_paragraphs(product)
```

```
[{'qas': [{'question': 'How is the bass?',
    'id': '2543d296da9766d8d17d040ecc781699',
    'is_impossible': True,
    'answers': []}],
  'context': 'I have had Koss headphones ...',
    'id': 'd476830bf9282e2b9033e2bb44bbb995',
    'is_impossible': False,
    'answers': [{'text': 'Bass is weak as expected', 'answer_start': 1302},
     {'text': 'Bass is weak as expected, even with EQ adjusted up',
      'answer_start': 1302}]}],
  'context': 'To anyone who hasn\'t tried all ...'},
 {'qas': [{'question': 'How is the bass?',
    'id': '455575557886d6dfeea5aa19577e5de4',
    'is_impossible': False,
    'answers': [{'text': 'The only fault in the sound is the bass',
      'answer_start': 650}]}],
  'context': "I have had many sub-$100 headphones ..."}]
```

The final step is to then apply this function to each product ID in the `DataFrame` of each split. The following `convert_to_squad()` function does this trick and stores the result in an *electronics-{split}.json* file:

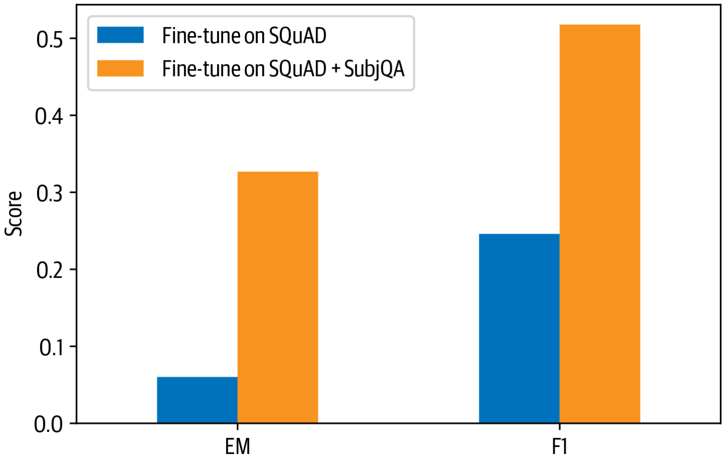
```
import json

def convert_to_squad(dfs):
    for split, df in dfs.items():
        subjqa_data = {}
        # Create `paragraphs` for each product ID
        groups = (df.groupby("title").apply(create_paragraphs)
            .to_frame(name="paragraphs").reset_index())
        subjqa_data["data"] = groups.to_dict(orient="records")
        # Save the result to disk
        with open(f"electronics-{split}.json", "w+", encoding="utf-8") as f:
            json.dump(subjqa_data, f)

convert_to_squad(dfs)
```

Now that we have the splits in the right format, let's fine-tune our reader by specifying the locations of the train and dev splits, along with where to save the fine-tuned model:

```
train_filename = "electronics-train.json"
dev_filename = "electronics-validation.json"

reader.train(data_dir=".", use_gpu=True, n_epochs=1, batch_size=16,
             train_filename=train_filename, dev_filename=dev_filename)
```

With the reader fine-tuned, let's now compare its performance on the test set against our baseline model:

```
reader_eval["Fine-tune on SQuAD + SubjQA"] = evaluate_reader(reader)
plot_reader_eval(reader_eval)
```



Wow, domain adaptation has increased our EM score by a factor of six and more than doubled the F_1-score! At this point, you might be wondering why we didn't just fine-tune a pretrained language model directly on the SubjQA training set. One reason is that we only have 1,295 training examples in SubjQA while SQuAD has over 100,000, so we might run into challenges with overfitting. Nevertheless, let's take a look at what naive fine-tuning produces. For a fair comparison, we'll use the same language model that was used for fine-tuning our baseline on SQuAD. As before, we'll load up the model with the FARMReader:

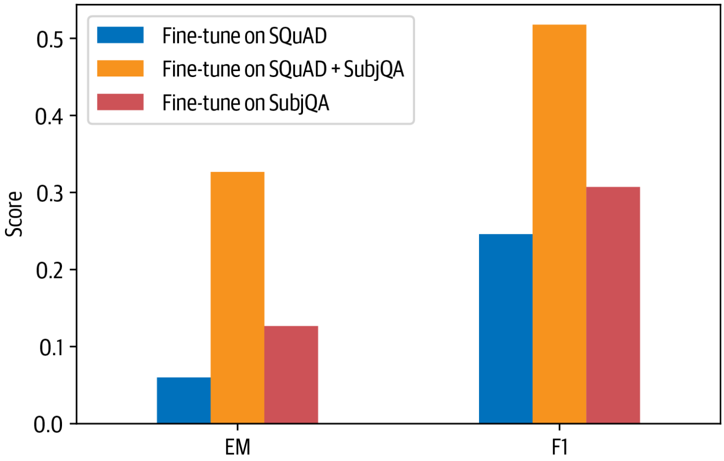
```
minilm_ckpt = "microsoft/MiniLM-L12-H384-uncased"
minilm_reader = FARMReader(model_name_or_path=minilm_ckpt, progress_bar=False,
                           max_seq_len=max_seq_length, doc_stride=doc_stride,
                           return_no_answer=True)
```

Next, we fine-tune for one epoch:

```
minilm_reader.train(data_dir=".", use_gpu=True, n_epochs=1, batch_size=16,
             train_filename=train_filename, dev_filename=dev_filename)
```

and include the evaluation on the test set:

```
reader_eval["Fine-tune on SubjQA"] = evaluate_reader(minilm_reader)
plot_reader_eval(reader_eval)
```



We can see that fine-tuning the language model directly on SubjQA results in considerably worse performance than fine-tuning on SQuAD and SubjQA.

When dealing with small datasets, it is best practice to use cross-validation when evaluating transformers as they can be prone to overfitting. You can find an example of how to perform cross-validation with SQuAD-formatted datasets in the FARM repository.

Evaluating the Whole QA Pipeline

Now that we've seen how to evaluate the reader and retriever components individually, let's tie them together to measure the overall performance of our pipeline. To do so, we'll need to augment our retriever pipeline with nodes for the reader and its evaluation. We've seen that we get almost perfect recall at $k = 10$, so we can fix this value and assess the impact this has on the reader's performance (since it will now receive multiple contexts per query compared to the SQuAD-style evaluation):

```
# Initialize retriever pipeline
pipe = EvalRetrieverPipeline(es_retriever)
# Add nodes for reader
eval_reader = EvalAnswers()
pipe.pipeline.add_node(component=reader, name="QAReader",
              inputs=["EvalRetriever"])
pipe.pipeline.add_node(component=eval_reader, name="EvalReader",
              inputs=["QAReader"])
# Evaluate!
run_pipeline(pipe)
# Extract metrics from reader
reader_eval["QA Pipeline (top-1)"] = {
    k:v for k,v in eval_reader.__dict__.items()
    if k in ["top_1_em", "top_1_f1"]}
```

We can then compare the top 1 EM and F_1 scores for the model to predict an answer in the documents returned by the retriever in Figure 7-12.

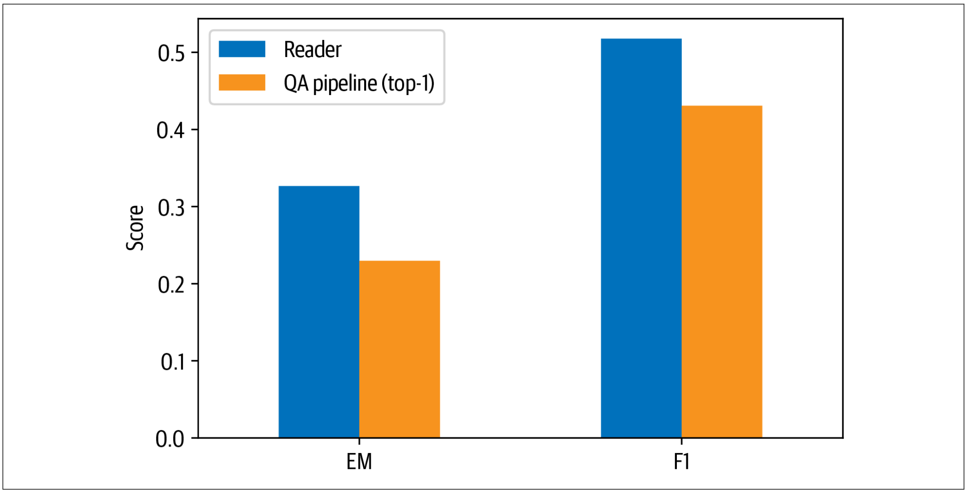


Figure 7-12. Comparison of EM and F_1 scores for the reader against the whole QA pipeline

From this plot we can see the effect that the retriever has on the overall performance. In particular, there is an overall degradation compared to matching the question-context pairs, as is done in the SQuAD-style evaluation. This can be circumvented by increasing the number of possible answers that the reader is allowed to predict.

Until now we have only extracted answer spans from the context, but in general it could be that bits and pieces of the answer are scattered throughout the document and we would like our model to synthesize these fragments into a single coherent answer. Let's have a look at how we can use generative QA to succeed at this task.

Going Beyond Extractive QA

One interesting alternative to extracting answers as spans of text in a document is to generate them with a pretrained language model. This approach is often referred to as *abstractive* or *generative* QA and has the potential to produce better-phrased answers that synthesize evidence across multiple passages. Although less mature than extractive QA, this is a fast-moving field of research, so chances are that these approaches will be widely adopted in industry by the time you are reading this! In this section we'll briefly touch on the current state of the art: *retrieval-augmented generation* (RAG).[16]

RAG extends the classic retriever-reader architecture that we've seen in this chapter by swapping the reader for a *generator* and using DPR as the retriever. The generator is a pretrained sequence-to-sequence transformer like T5 or BART that receives latent vectors of documents from DPR and then iteratively generates an answer based on the query and these documents. Since DPR and the generator are differentiable, the whole process can be fine-tuned end-to-end as illustrated in Figure 7-13.

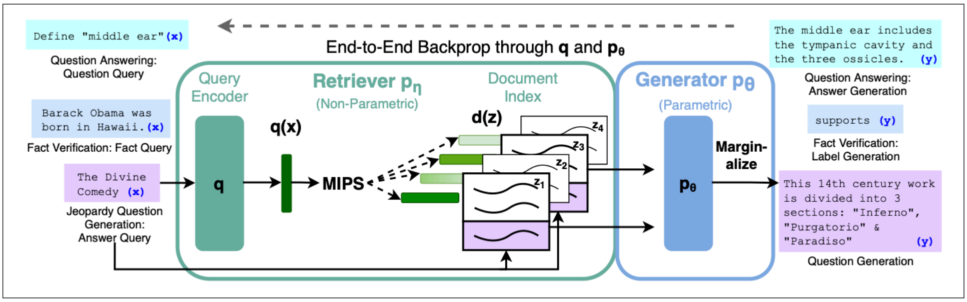

Define "middle ear" (x)
Question Answering: Question Query
Barack Obama was born in Hawaii. (x)
Fact Verification: Fact Query
The Divine Comedy (x)
Jeopardy Question Generation: Answer Query
End-to-End Backprop through q and p_θ
Query Encoder
Retriever p_η
(Non-Parametric)
Document Index
q
q(x)
MIPS
d(z)
z_4
z_3
z_2
z_1
Generator p_θ
(Parametric)
p_θ
Margin-alize
The middle ear includes the tympanic cavity and the three ossicles. (y)
Question Answering: Answer Generation
supports (y)
Fact Verification: Label Generation
This 14th century work is divided into 3 sections: "Inferno", "Purgatorio" & "Paradiso" (y)
Question Generation

Figure 7-13. The RAG architecture for fine-tuning a retriever and generator end-to-end (courtesy of Ethan Perez)

To show RAG in action we'll use the `DPRetriever` from earlier, so we just need to instantiate a generator. There are two types of RAG models to choose from:

RAG-Sequence

Uses the same retrieved document to generate the complete answer. In particular, the top *k* documents from the retriever are fed to the generator, which produces an output sequence for each document, and the result is marginalized to obtain the best answer.

[16] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", (2020).

RAG-Token

Can use a different document to generate each token in the answer. This allows the generator to synthesize evidence from multiple documents.

Since RAG-Token models tend to perform better than RAG-Sequence ones, we'll use the token model that was fine-tuned on NQ as our generator. Instantiating a generator in Haystack is similar to instantiating the reader, but instead of specifying the `max_seq_length` and `doc_stride` parameters for a sliding window over the contexts, we specify hyperparameters that control the text generation:

```python
from haystack.generator.transformers import RAGenerator

generator = RAGenerator(model_name_or_path="facebook/rag-token-nq",
                        embed_title=False, num_beams=5)
```

Here `num_beams` specifies the number of beams to use in beam search (text generation is covered at length in Chapter 5). As we did with the DPR retriever, we don't embed the document titles since our corpus is always filtered per product ID.

The next thing to do is tie together the retriever and generator using Haystack's `GenerativeQAPipeline`:

```python
from haystack.pipeline import GenerativeQAPipeline

pipe = GenerativeQAPipeline(generator=generator, retriever=dpr_retriever)
```

In RAG, both the query encoder and the generator are trained end-to-end, while the context encoder is frozen. In Haystack, the `GenerativeQAPipeline` uses the query encoder from `RAGenerator` and the context encoder from `DensePassageRetriever`.

Let's now give RAG a spin by feeding in some queries about the Amazon Fire tablet from before. To simplify the querying, we'll write a simple function that takes the query and prints out the top answers:

```python
def generate_answers(query, top_k_generator=3):
    preds = pipe.run(query=query, top_k_generator=top_k_generator,
                     top_k_retriever=5, filters={"item_id":["B0074BW614"]})
    print(f"Question: {preds['query']} \n")
    for idx in range(top_k_generator):
        print(f"Answer {idx+1}: {preds['answers'][idx]['answer']}")
```

OK, now we're ready to give it a test:

```python
generate_answers(query)
```

```
Question: Is it good for reading?

Answer 1:  the screen is absolutely beautiful
Answer 2:  the Screen is absolutely beautiful
Answer 3:  Kindle fire
```

This result isn't too bad for an answer, but it does suggest that the subjective nature of the question is confusing the generator. Let's try with something a bit more factual:

```
generate_answers("What is the main drawback?")
```

```
Question: What is the main drawback?

Answer 1:  the price
Answer 2:  no flash support
Answer 3:  the cost
```

This is more sensible! To get better results we could fine-tune RAG end-to-end on SubjQA; we'll leave this as an exercise, but if you're interested in exploring it there are scripts in the 🤗 Transformers repository to help you get started.

Conclusion

Well, that was a whirlwind tour of QA, and you probably have many more questions that you'd like answered (pun intended!). In this chapter, we discussed two approaches to QA (extractive and generative) and examined two different retrieval algorithms (BM25 and DPR). Along the way, we saw that domain adaptation can be a simple technique to boost the performance of our QA system by a significant margin, and we looked at a few of the most common metrics that are used for evaluating such systems. Although we focused on closed-domain QA (i.e., a single domain of electronic products), the techniques in this chapter can easily be generalized to the open-domain case; we recommend reading Cloudera's excellent Fast Forward QA series to see what's involved.

Deploying QA systems in the wild can be a tricky business to get right, and our experience is that a significant part of the value comes from first providing end users with useful search capabilities, followed by an extractive component. In this respect, the reader can be used in novel ways beyond answering on-demand user queries. For example, researchers at Grid Dynamics were able to use their reader to automatically extract a set of pros and cons for each product in a client's catalog. They also showed that a reader can be used to extract named entities in a zero-shot fashion by creating queries like "What kind of camera?" Given its infancy and subtle failure modes, we recommend exploring generative QA only once the other two approaches have been exhausted. This "hierarchy of needs" for tackling QA problems is illustrated in Figure 7-14.

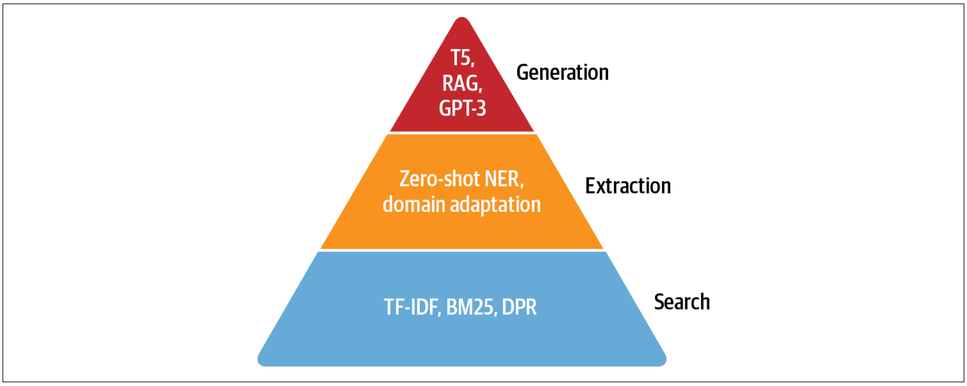


Figure 7-14. The QA hierarchy of needs

Looking ahead, one exciting research area is *multimodal QA*, which involves QA over multiple modalities like text, tables, and images. As described in the MultiModalQA benchmark,[17] such systems could enable users to answer complex questions that integrate information across different modalities, like "When was the famous painting with two touching fingers completed?" Another area with practical business applications is QA over a *knowledge graph*, where the nodes of the graph correspond to real-world entities and their relations are defined by the edges. By encoding factoids as (*subject*, *predicate*, *object*) triples, one can use the graph to answer questions about a missing element. For an example that combines transformers with knowledge graphs, see the Haystack tutorials. One more promising direction is *automatic question generation* as a way to do some form of unsupervised/weakly supervised training using unlabeled data or data augmentation. Two recent examples include the papers on the Probably Answered Questions (PAQ) benchmark and synthetic data augmentation for cross-lingual settings.[18]

In this chapter we've seen that in order to successfully use QA models for real-world use cases we need to apply a few tricks, such as implementing a fast retrieval pipeline to make predictions in near real time. Still, applying a QA model to a handful of preselected documents can take a couple of seconds on production hardware. Although this may not sound like much, imagine how different your experience would be if you had to wait a few seconds to get the results of a Google search—a few seconds of wait time can decide the fate of your transformer-powered application. In the next chapter we'll have a look at a few methods to accelerate model predictions further.

[17] A. Talmor et al., "MultiModalQA: Complex Question Answering over Text, Tables and Images", (2021).

[18] P. Lewis et al., "PAQ: 65 Million Probably-Asked Questions and What You Can Do with Them", (2021); A. Riabi et al., "Synthetic Data Augmentation for Zero-Shot Cross-Lingual Question Answering", (2020).