CHAPTER 6

Summarization

At one point or another, you've probably needed to summarize a document, be it a research article, a financial earnings report, or a thread of emails. If you think about it, this requires a range of abilities, such as understanding long passages, reasoning about the contents, and producing fluent text that incorporates the main topics from the original document. Moreover, accurately summarizing a news article is very different from summarizing a legal contract, so being able to do so requires a sophisticated degree of domain generalization. For these reasons, text summarization is a difficult task for neural language models, including transformers. Despite these challenges, text summarization offers the prospect for domain experts to significantly speed up their workflows and is used by enterprises to condense internal knowledge, summarize contracts, automatically generate content for social media releases, and more.

To help you understand the challenges involved, this chapter will explore how we can leverage pretrained transformers to summarize documents. Summarization is a classic sequence-to-sequence (seq2seq) task with an input text and a target text. As we saw in Chapter 1, this is where encoder-decoder transformers excel.

In this chapter we will build our own encoder-decoder model to condense dialogues between several people into a crisp summary. But before we get to that, let's begin by taking a look at one of the canonical datasets for summarization: the CNN/DailyMail corpus.

The CNN/DailyMail Dataset

The CNN/DailyMail dataset consists of around 300,000 pairs of news articles and their corresponding summaries, composed from the bullet points that CNN and the DailyMail attach to their articles. An important aspect of the dataset is that the

summaries are *abstractive* and not *extractive*, which means that they consist of new sentences instead of simple excerpts. The dataset is available on the Hub; we'll use version 3.0.0, which is a nonanonymized version set up for summarization. We can select versions in a similar manner as splits, we saw in Chapter 4, with a `version` keyword. So let's dive in and have a look at it:

```
from datasets import load_dataset

dataset = load_dataset("cnn_dailymail", version="3.0.0")
print(f"Features: {dataset['train'].column_names}")
```

```
Features: ['article', 'highlights', 'id']
```

The dataset has three columns: `article`, which contains the news articles, `highlights` with the summaries, and `id` to uniquely identify each article. Let's look at an excerpt from an article:

```
sample = dataset["train"][1]
print(f"""
Article (excerpt of 500 characters, total length: {len(sample["article"])}):
""")
print(sample["article"][:500])
print(f'\nSummary (length: {len(sample["highlights"])}):')
print(sample["highlights"])
```

```
Article (excerpt of 500 characters, total length: 3192):

(CNN) -- Usain Bolt rounded off the world championships Sunday by claiming his
third gold in Moscow as he anchored Jamaica to victory in the men's 4x100m
relay. The fastest man in the world charged clear of United States rival Justin
Gatlin as the Jamaican quartet of Nesta Carter, Kemar Bailey-Cole, Nickel
Ashmeade and Bolt won in 37.36 seconds. The U.S finished second in 37.56 seconds
with Canada taking the bronze after Britain were disqualified for a faulty
handover. The 26-year-old Bolt has n

Summary (length: 180):
Usain Bolt wins third gold of world championship .
Anchors Jamaica to 4x100m relay victory .
Eighth gold at the championships for Bolt .
Jamaica double up in women's 4x100m relay .
```

We see that the articles can be very long compared to the target summary; in this particular case the difference is 17-fold. Long articles pose a challenge to most transformer models since the context size is usually limited to 1,000 tokens or so, which is equivalent to a few paragraphs of text. The standard, yet crude way to deal with this for summarization is to simply truncate the texts beyond the model's context size. Obviously there could be important information for the summary toward the end of the text, but for now we need to live with this limitation of the model architectures.

Text Summarization Pipelines

Let's see how a few of the most popular transformer models for summarization perform by first looking qualitatively at the outputs for the preceding example. Although the model architectures we will be exploring have varying maximum input sizes, let's restrict the input text to 2,000 characters to have the same input for all models and thus make the outputs more comparable:

```
sample_text = dataset["train"][1]["article"][:2000]
# We'll collect the generated summaries of each model in a dictionary
summaries = {}
```

A convention in summarization is to separate the summary sentences by a newline. We could add a newline token after each full stop, but this simple heuristic would fail for strings like "U.S." or "U.N." The Natural Language Toolkit (NLTK) package includes a more sophisticated algorithm that can differentiate the end of a sentence from punctuation that occurs in abbreviations:

```
import nltk
from nltk.tokenize import sent_tokenize

nltk.download("punkt")
```

```
string = "The U.S. are a country. The U.N. is an organization."
sent_tokenize(string)
```

```
['The U.S. are a country.', 'The U.N. is an organization.']
```

In the following sections we will load several large models. If you run out of memory, you can either replace the large models with smaller checkpoints (e.g., "gpt", "t5-small") or skip this section and jump to "Evaluating PEGASUS on the CNN/DailyMail Dataset" on page 154.

Summarization Baseline

A common baseline for summarizing news articles is to simply take the first three sentences of the article. With NLTK's sentence tokenizer, we can easily implement such a baseline:

```
def three_sentence_summary(text):
    return "\n".join(sent_tokenize(text)[:3])
```

```
summaries["baseline"] = three_sentence_summary(sample_text)
```

GPT-2

We've already seen in Chapter 5 how GPT-2 can generate text given some prompt. One of the model's surprising features is that we can also use it to generate summaries by simply appending "TL;DR" at the end of the input text. The expression "TL;DR" (too long; didn't read) is often used on platforms like Reddit to indicate a short version of a long post. We will start our summarization experiment by re-creating the procedure of the original paper with the `pipeline()` function from 🤗 Transformers.[1] We create a text generation pipeline and load the large GPT-2 model:

```
from transformers import pipeline, set_seed

set_seed(42)
pipe = pipeline("text-generation", model="gpt2-xl")
gpt2_query = sample_text + "\nTL;DR:\n"
pipe_out = pipe(gpt2_query, max_length=512, clean_up_tokenization_spaces=True)
summaries["gpt2"] = "\n".join(
    sent_tokenize(pipe_out[0]["generated_text"][len(gpt2_query) :]))
```

Here we just store the summaries of the generated text by slicing off the input query and keep the result in a Python dictionary for later comparison.

T5

Next let's try the T5 transformer. As we saw in Chapter 3, the developers of this model performed a comprehensive study of transfer learning in NLP and found they could create a universal transformer architecture by formulating all tasks as text-to-text tasks. The T5 checkpoints are trained on a mixture of unsupervised data (to reconstruct masked words) and supervised data for several tasks, including summarization. These checkpoints can thus be directly used to perform summarization without fine-tuning by using the same prompts used during pretraining. In this framework, the input format for the model to summarize a document is `"summarize: <ARTICLE>"`, and for translation it looks like `"translate English to German: <TEXT>"`. As shown in Figure 6-1, this makes T5 extremely versatile and allows you to solve many tasks with a single model.

We can directly load T5 for summarization with the `pipeline()` function, which also takes care of formatting the inputs in the text-to-text format so we don't need to prepend them with `"summarize"`:

```
pipe = pipeline("summarization", model="t5-large")
pipe_out = pipe(sample_text)
summaries["t5"] = "\n".join(sent_tokenize(pipe_out[0]["summary_text"]))
```

1 A. Radford et al., "Language Models Are Unsupervised Multitask Learners", OpenAI (2019).

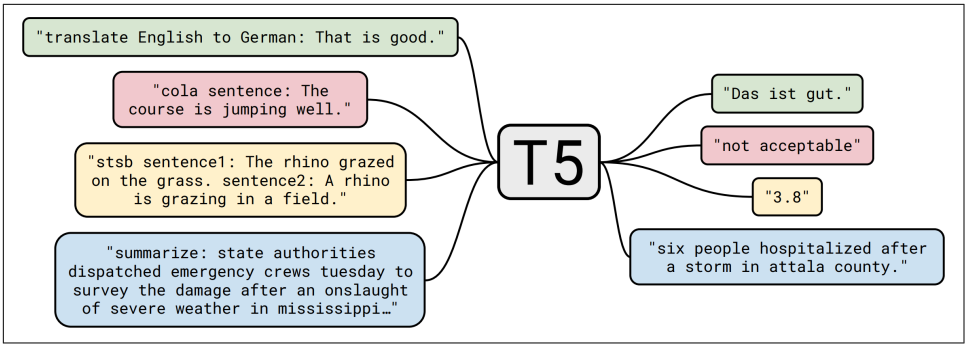


Figure 6-1. Diagram of T5's text-to-text framework (courtesy of Colin Raffel); besides translation and summarization, the CoLA (linguistic acceptability) and STSB (semantic similarity) tasks are shown

BART

BART also uses an encoder-decoder architecture and is trained to reconstruct corrupted inputs. It combines the pretraining schemes of BERT and GPT-2.[2] We'll use the `facebook/bart-large-ccn` checkpoint, which has been specifically fine-tuned on the CNN/DailyMail dataset:

```
pipe = pipeline("summarization", model="facebook/bart-large-cnn")
pipe_out = pipe(sample_text)
summaries["bart"] = "\n".join(sent_tokenize(pipe_out[0]["summary_text"]))
```

PEGASUS

Like BART, PEGASUS is an encoder-decoder transformer.[3] As shown in Figure 6-2, its pretraining objective is to predict masked sentences in multisentence texts. The authors argue that the closer the pretraining objective is to the downstream task, the more effective it is. With the aim of finding a pretraining objective that is closer to summarization than general language modeling, they automatically identified, in a very large corpus, sentences containing most of the content of their surrounding paragraphs (using summarization evaluation metrics as a heuristic for content overlap) and pretrained the PEGASUS model to reconstruct these sentences, thereby obtaining a state-of-the-art model for text summarization.

2 M. Lewis et al., "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension", (2019).

3 J. Zhang et al., "PEGASUS: Pre-Training with Extracted Gap-Sentences for Abstractive Summarization", (2019).

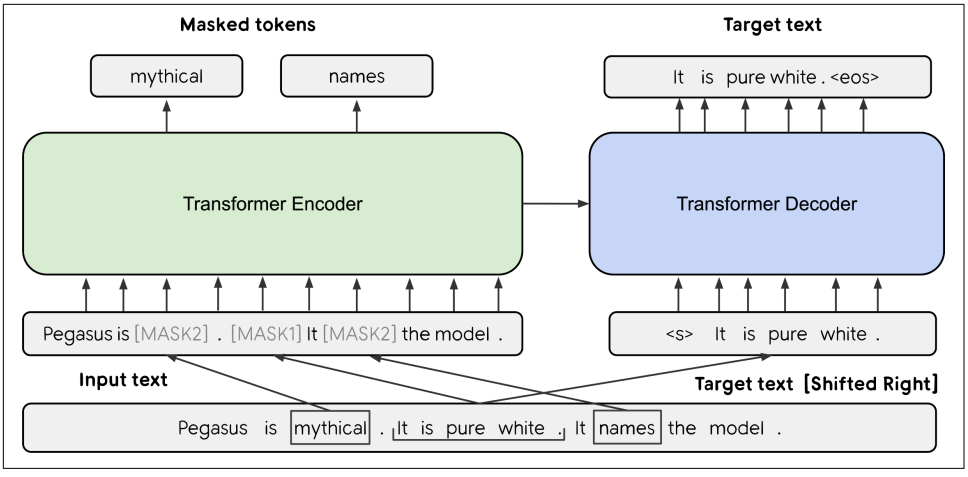


Figure 6-2. Diagram of PEGASUS architecture (courtesy of Jingqing Zhang et al.)

This model has a special token for newlines, which is why we don't need the `sent_tokenize()` function:

```
pipe = pipeline("summarization", model="google/pegasus-cnn_dailymail")
pipe_out = pipe(sample_text)
summaries["pegasus"] = pipe_out[0]["summary_text"].replace(" .<n>", ".\n")
```

Comparing Different Summaries

Now that we have generated summaries with four different models, let's compare the results. Keep in mind that one model has not been trained on the dataset at all (GPT-2), one model has been fine-tuned on this task among others (T5), and two models have exclusively been fine-tuned on this task (BART and PEGASUS). Let's have a look at the summaries these models have generated:

```
print("GROUND TRUTH")
print(dataset["train"][1]["highlights"])
print("")

for model_name in summaries:
    print(model_name.upper())
    print(summaries[model_name])
    print("")
```

```
GROUND TRUTH
Usain Bolt wins third gold of world championship .
Anchors Jamaica to 4x100m relay victory .
Eighth gold at the championships for Bolt .
Jamaica double up in women's 4x100m relay .

BASELINE
```

```
(CNN) -- Usain Bolt rounded off the world championships Sunday by claiming his
third gold in Moscow as he anchored Jamaica to victory in the men's 4x100m
relay.
The fastest man in the world charged clear of United States rival Justin Gatlin
as the Jamaican quartet of Nesta Carter, Kemar Bailey-Cole, Nickel Ashmeade and
Bolt won in 37.36 seconds.
The U.S finished second in 37.56 seconds with Canada taking the bronze after
Britain were disqualified for a faulty handover.

GPT2
Nesta, the fastest man in the world.
Gatlin, the most successful Olympian ever.
Kemar, a Jamaican legend.
Shelly-Ann, the fastest woman ever.
Bolt, the world's greatest athlete.
The team sport of pole vaulting

T5
usain bolt wins his third gold medal of the world championships in the men's
4x100m relay .
the 26-year-old anchored Jamaica to victory in the event in the Russian capital
.
he has now collected eight gold medals at the championships, equaling the record
.

BART
Usain Bolt wins his third gold of the world championships in Moscow.
Bolt anchors Jamaica to victory in the men's 4x100m relay.
The 26-year-old has now won eight gold medals at world championships.
Jamaica's women also win gold in the relay, beating France in the process.

PEGASUS
Usain Bolt wins third gold of world championships.
Anchors Jamaica to victory in men's 4x100m relay.
Eighth gold at the championships for Bolt.
Jamaica also win women's 4x100m relay .
```

The first thing we notice by looking at the model outputs is that the summary generated by GPT-2 is quite different from the others. Instead of giving a summary of the text, it summarizes the characters. Often the GPT-2 model "hallucinates" or invents facts, since it was not explicitly trained to generate truthful summaries. For example, at the time of writing, Nesta is not the fastest man in the world, but sits in ninth place. Comparing the other three model summaries against the ground truth, we see that there is remarkable overlap, with PEGASUS's output bearing the most striking resemblance.

Now that we have inspected a few models, let's try to decide which one we would use in a production setting. All four models seem to provide qualitatively reasonable results, and we could generate a few more examples to help us decide. However, this is not a systematic way of determining the best model! Ideally, we would define a

metric, measure it for all models on some benchmark dataset, and choose the one with the best performance. But how do you define a metric for text generation? The standard metrics that we've seen, like accuracy, recall, and precision, are not easy to apply to this task. For each "gold standard" summary written by a human, dozens of other summaries with synonyms, paraphrases, or a slightly different way of formulating the facts could be just as acceptable.

In the next section we will look at some common metrics that have been developed for measuring the quality of generated text.

Measuring the Quality of Generated Text

Good evaluation metrics are important, since we use them to measure the performance of models not only when we train them but also later, in production. If we have bad metrics we might be blind to model degradation, and if they are misaligned with the business goals we might not create any value.

Measuring performance on a text generation task is not as easy as with standard classification tasks such as sentiment analysis or named entity recognition. Take the example of translation; given a sentence like "I love dogs!" in English and translating it to Spanish there can be multiple valid possibilities, like "¡Me encantan los perros!" or "¡Me gustan los perros!" Simply checking for an exact match to a reference translation is not optimal; even humans would fare badly on such a metric because we all write text slightly differently from each other (and even from ourselves, depending on the time of the day or year!). Fortunately, there are alternatives.

Two of the most common metrics used to evaluate generated text are BLEU and ROUGE. Let's take a look at how they're defined.

BLEU

The idea of BLEU is simple:[4] instead of looking at how many of the tokens in the generated texts are perfectly aligned with the reference text tokens, we look at words or *n*-grams. BLEU is a precision-based metric, which means that when we compare the two texts we count the number of words in the generation that occur in the reference and divide it by the length of the reference.

However, there is an issue with this vanilla precision. Assume the generated text just repeats the same word over and over again, and this word also appears in the reference. If it is repeated as many times as the length of the reference text, then we get

4 K. Papineni et al., "BLEU: A Method for Automatic Evaluation of Machine Translation," *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics* (July 2002): 311–318, http://dx.doi.org/10.3115/1073083.1073135.

perfect precision! For this reason, the authors of the BLEU paper introduced a slight modification: a word is only counted as many times as it occurs in the reference. To illustrate this point, suppose we have the reference text "the cat is on the mat" and the generated text "the the the the the the".

From this simple example, we can calculate the precision values as follows:

$$p_{vanilla} = \frac{6}{6}$$

$$p_{mod} = \frac{2}{6}$$

and we can see that the simple correction has produced a much more reasonable value. Now let's extend this by not only looking at single words, but *n*-grams as well. Let's assume we have one generated sentence, *snt*, that we want to compare against a reference sentence, *snt′*. We extract all possible *n*-grams of degree *n* and do the accounting to get the precision p_n:

$$p_n = \frac{\Sigma_{n\text{-}gram \in snt} Count_{clip}(n\text{-}gram)}{\Sigma_{n\text{-}gram \in snt'} Count(n\text{-}gram)}$$

In order to avoid rewarding repetitive generations, the count in the numerator is clipped. What this means is that the occurrence count of an *n*-gram is capped at how many times it appears in the reference sentence. Also note that the definition of a sentence is not very strict in this equation, and if you had a generated text spanning multiple sentences you would treat it as one sentence.

In general we have more than one sample in the test set we want to evaluate, so we need to slightly extend the equation by summing over all samples in the corpus *C*:

$$p_n = \frac{\Sigma_{snt \in C} \Sigma_{n\text{-}gram \in snt} Count_{clip}(n\text{-}gram)}{\Sigma_{snt' \in C} \Sigma_{n\text{-}gram \in snt'} Count(n\text{-}gram)}$$

We're almost there. Since we are not looking at recall, all generated sequences that are short but precise have a benefit compared to sentences that are longer. Therefore, the precision score favors short generations. To compensate for that the authors of BLEU introduced an additional term, the *brevity penalty*:

$$BR = \min\left(1, e^{1 - \ell_{ref}/\ell_{gen}}\right)$$

By taking the minimum, we ensure that this penalty never exceeds 1 and the exponential term becomes exponentially small when the length of the generated text l_{gen} is smaller than the reference text l_{ref}. At this point you might ask, why don't we just use something like an F_1-score to account for recall as well? The answer is that often in translation datasets there are multiple reference sentences instead of just one, so if we also measured recall we would incentivize translations that used all the words from all the references. Therefore, it's preferable to look for high precision in the translation and make sure the translation and reference have a similar length.

Finally, we can put everything together and get the equation for the BLEU score:

$$\text{BLEU-}N = BR \times \left(\prod_{n=1}^{N} p_n \right)^{1/N}$$

The last term is the geometric mean of the modified precision up to n-gram N. In practice, the BLEU-4 score is often reported. However, you can probably already see that this metric has many limitations; for instance, it doesn't take synonyms into account, and many steps in the derivation seem like ad hoc and rather fragile heuristics. You can find a wonderful exposition of BLEU's flaws in Rachel Tatman's blog post "Evaluating Text Output in NLP: BLEU at Your Own Risk".

In general, the field of text generation is still looking for better evaluation metrics, and finding ways to overcome the limits of metrics like BLEU is an active area of research. Another weakness of the BLEU metric is that it expects the text to already be tokenized. This can lead to varying results if the exact same method for text tokenization is not used. The SacreBLEU metric addresses this issue by internalizing the tokenization step; for this reason, it is the preferred metric for benchmarking.

We've now worked through some theory, but what we really want to do is calculate the score for some generated text. Does that mean we need to implement all this logic in Python? Fear not, 🤗 Datasets also provides metrics! Loading a metric works just like loading a dataset:

```
from datasets import load_metric

bleu_metric = load_metric("sacrebleu")
```

The `bleu_metric` object is an instance of the `Metric` class, and works like an aggregator: you can add single instances with `add()` or whole batches via `add_batch()`. Once you have added all the samples you need to evaluate, you then call `compute()` and the metric is calculated. This returns a dictionary with several values, such as the precision for each n-gram, the length penalty, as well as the final BLEU score. Let's look at the example from before:

```
import pandas as pd
import numpy as np

bleu_metric.add(
    prediction="the the the the the the", reference=["the cat is on the mat"])
results = bleu_metric.compute(smooth_method="floor", smooth_value=0)
results["precisions"] = [np.round(p, 2) for p in results["precisions"]]
pd.DataFrame.from_dict(results, orient="index", columns=["Value"])
```

	Value
score	0.0
counts	[2, 0, 0, 0]
totals	[6, 5, 4, 3]
precisions	[33.33, 0.0, 0.0, 0.0]
bp	1.0
sys_len	6
ref_len	6

The BLEU score also works if there are multiple reference translations. This is why `reference` is passed as a list. To make the metric smoother for zero counts in the *n*-grams, BLEU integrates methods to modify the precision calculation. One method is to add a constant to the numerator. That way, a missing *n*-gram does not cause the score to automatically go to zero. For the purpose of explaining the values, we turn it off by setting `smooth_value=0`.

We can see the precision of the 1-gram is indeed 2/6, whereas the precisions for the 2/3/4-grams are all 0. (For more information about the individual metrics, like counts and bp, see the SacreBLEU repository.) This means the geometric mean is zero, and thus also the BLEU score. Let's look at another example where the prediction is almost correct:

```
bleu_metric.add(
    prediction="the cat is on mat", reference=["the cat is on the mat"])
results = bleu_metric.compute(smooth_method="floor", smooth_value=0)
results["precisions"] = [np.round(p, 2) for p in results["precisions"]]
pd.DataFrame.from_dict(results, orient="index", columns=["Value"])
```

	Value
score	57.893007
counts	[5, 3, 2, 1]
totals	[5, 4, 3, 2]
precisions	[100.0, 75.0, 66.67, 50.0]

	Value
bp	0.818731
sys_len	5
ref_len	6

We observe that the precision scores are much better. The 1-grams in the prediction all match, and only in the precision scores do we see that something is off. For the 4-gram there are only two candidates, `["the", "cat", "is", "on"]` and `["cat", "is", "on", "mat"]`, where the last one does not match, hence the precision of 0.5.

The BLEU score is widely used for evaluating text, especially in machine translation, since precise translations are usually favored over translations that include all possible and appropriate words.

There are other applications, such as summarization, where the situation is different. There, we want all the important information in the generated text, so we favor high recall. This is where the ROUGE score is usually used.

ROUGE

The ROUGE score was specifically developed for applications like summarization where high recall is more important than just precision.[5] The approach is very similar to the BLEU score in that we look at different *n*-grams and compare their occurrences in the generated text and the reference texts. The difference is that with ROUGE we check how many *n*-grams in the reference text also occur in the generated text. For BLEU we looked at how many *n*-grams in the generated text appear in the reference, so we can reuse the precision formula with the minor modification that we count the (unclipped) occurrence of reference *n*-grams in the generated text in the numerator:

$$\text{ROUGE-}N = \frac{\sum_{\text{snt'} \in C} \sum_{n\text{-}gram \in snt'} Count_{match}(n\text{-}gram)}{\sum_{\text{snt'} \in C} \sum_{n\text{-}gram \in snt'} Count(n\text{-}gram)}$$

This was the original proposal for ROUGE. Subsequently, researchers have found that fully removing precision can have strong negative effects. Going back to the BLEU formula without the clipped counting, we can measure precision as well, and we can then combine both precision and recall ROUGE scores in the harmonic mean to get an F_1-score. This score is the metric that is nowadays commonly reported for ROUGE.

5 C-Y. Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," *Text Summarization Branches Out* (July 2004), *https://aclanthology.org/W04-1013.pdf*.

There is a separate score in ROUGE to measure the longest common substring (LCS), called ROUGE-L. The LCS can be calculated for any pair of strings. For example, the LCS for "abab" and "abc" would be "ab", and its the length would be 2. If we want to compare this value between two samples we need to somehow normalize it because otherwise a longer text would be at an advantage. To achieve this, the inventor of ROUGE came up with an *F*-score-like scheme where the LCS is normalized with the length of the reference and generated text, then the two normalized scores are mixed together:

$$R_{LCS} = \frac{LCS(X, Y)}{m}$$

$$P_{LCS} = \frac{LCS(X, Y)}{n}$$

$$F_{LCS} = \frac{\left(1 + \beta^2\right) R_{LCS} P_{LCS}}{R_{LCS} + \beta P_{LCS}}, \text{where } \beta = P_{LCS} / R_{LCS}$$

That way the LCS score is properly normalized and can be compared across samples. In the 🤗 Datasets implementation, two variations of ROUGE are calculated: one calculates the score per sentence and averages it for the summaries (ROUGE-L), and the other calculates it directly over the whole summary (ROUGE-Lsum).

We can load the metric as follows:

```
rouge_metric = load_metric("rouge")
```

We already generated a set of summaries with GPT-2 and the other models, and now we have a metric to compare the summaries systematically. Let's apply the ROUGE score to all the summaries generated by the models:

```
reference = dataset["train"][1]["highlights"]
records = []
rouge_names = ["rouge1", "rouge2", "rougeL", "rougeLsum"]

for model_name in summaries:
    rouge_metric.add(prediction=summaries[model_name], reference=reference)
    score = rouge_metric.compute()
    rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
    records.append(rouge_dict)
pd.DataFrame.from_records(records, index=summaries.keys())
```

	rouge1	rouge2	rougeL	rougeLsum
baseline	0.303571	0.090909	0.214286	0.232143
gpt2	0.187500	0.000000	0.125000	0.187500

	rouge1	rouge2	rougeL	rougeLsum
t5	0.486486	0.222222	0.378378	0.486486
bart	0.582278	0.207792	0.455696	0.506329
pegasus	0.866667	0.655172	0.800000	0.833333

The ROUGE metric in the 🤗 Datasets library also calculates confidence intervals (by default, the 5th and 95th percentiles). The average value is stored in the attribute `mid` and the interval can be retrieved with `low` and `high`.

These results are obviously not very reliable as we only looked at a single sample, but we can compare the quality of the summary for that one example. The table confirms our observation that of the models we considered, GPT-2 performs worst. This is not surprising since it is the only model of the group that was not explicitly trained to summarize. It is striking, however, that the simple first-three-sentence baseline doesn't fare too poorly compared to the transformer models that have on the order of a billion parameters! PEGASUS and BART are the best models overall (higher ROUGE scores are better), but T5 is slightly better on ROUGE-1 and the LCS scores. These results place T5 and PEGASUS as the best models, but again these results should be treated with caution as we only evaluated the models on a single example. Looking at the results in the PEGASUS paper, we would expect the PEGASUS to outperform T5 on the CNN/DailyMail dataset.

Let's see if we can reproduce those results with PEGASUS.

Evaluating PEGASUS on the CNN/DailyMail Dataset

We now have all the pieces in place to evaluate the model properly: we have a dataset with a test set from CNN/DailyMail, we have a metric with ROUGE, and we have a summarization model. We just need to put the pieces together. Let's first evaluate the performance of the three-sentence baseline:

```
def evaluate_summaries_baseline(dataset, metric,
                                column_text="article",
                                column_summary="highlights"):
    summaries = [three_sentence_summary(text) for text in dataset[column_text]]
    metric.add_batch(predictions=summaries,
                     references=dataset[column_summary])
    score = metric.compute()
    return score
```

Now we'll apply the function to a subset of the data. Since the test fraction of the CNN/DailyMail dataset consists of roughly 10,000 samples, generating summaries for all these articles takes a lot of time. Recall from Chapter 5 that every generated token

requires a forward pass through the model; generating just 100 tokens for each sample will thus require 1 million forward passes, and if we use beam search this number is multiplied by the number of beams. For the purpose of keeping the calculations relatively fast, we'll subsample the test set and run the evaluation on 1,000 samples instead. This should give us a much more stable score estimation while completing in less than one hour on a single GPU for the PEGASUS model:

```
test_sampled = dataset["test"].shuffle(seed=42).select(range(1000))

score = evaluate_summaries_baseline(test_sampled, rouge_metric)
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame.from_dict(rouge_dict, orient="index", columns=["baseline"]).T
```

	rouge1	rouge2	rougeL	rougeLsum
baseline	0.396061	0.173995	0.245815	0.361158

The scores are mostly worse than on the previous example, but still better than those achieved by GPT-2! Now let's implement the same evaluation function for evaluating the PEGASUS model:

```
from tqdm import tqdm
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"

def chunks(list_of_elements, batch_size):
    """Yield successive batch-sized chunks from list_of_elements."""
    for i in range(0, len(list_of_elements), batch_size):
        yield list_of_elements[i : i + batch_size]

def evaluate_summaries_pegasus(dataset, metric, model, tokenizer,
                               batch_size=16, device=device,
                               column_text="article",
                               column_summary="highlights"):
    article_batches = list(chunks(dataset[column_text], batch_size))
    target_batches = list(chunks(dataset[column_summary], batch_size))

    for article_batch, target_batch in tqdm(
        zip(article_batches, target_batches), total=len(article_batches)):

        inputs = tokenizer(article_batch, max_length=1024,  truncation=True,
                        padding="max_length", return_tensors="pt")

        summaries = model.generate(input_ids=inputs["input_ids"].to(device),
                         attention_mask=inputs["attention_mask"].to(device),
                         length_penalty=0.8, num_beams=8, max_length=128)

        decoded_summaries = [tokenizer.decode(s, skip_special_tokens=True,
                                clean_up_tokenization_spaces=True)
               for s in summaries]
```

```
            decoded_summaries = [d.replace("<n>", " ") for d in decoded_summaries]
            metric.add_batch(predictions=decoded_summaries, references=target_batch)

        score = metric.compute()
        return score
```

Let's unpack this evaluation code a bit. First we split the dataset into smaller batches that we can process simultaneously. Then for each batch we tokenize the input articles and feed them to the `generate()` function to produce the summaries using beam search. We use the same generation parameters as proposed in the paper. The new parameter for length penalty ensures that the model does not generate sequences that are too long. Finally, we decode the generated texts, replace the `<n>` token, and add the decoded texts with the references to the metric. At the end, we compute and return the ROUGE scores. Let's now load the model again with the `AutoModelForSeq2SeqLM` class, used for seq2seq generation tasks, and evaluate it:

```
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer

model_ckpt = "google/pegasus-cnn_dailymail"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = AutoModelForSeq2SeqLM.from_pretrained(model_ckpt).to(device)
score = evaluate_summaries_pegasus(test_sampled, rouge_metric,
                                   model, tokenizer, batch_size=8)
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame(rouge_dict, index=["pegasus"])
```

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.434381	0.210883	0.307195	0.373231

These numbers are very close to the published results. One thing to note here is that the loss and per-token accuracy are decoupled to some degree from the ROUGE scores. The loss is independent of the decoding strategy, whereas the ROUGE score is strongly coupled.

Since ROUGE and BLEU correlate better with human judgment than loss or accuracy, we should focus on them and carefully explore and choose the decoding strategy when building text generation models. These metrics are far from perfect, however, and one should always consider human judgments as well.

Now that we're equipped with an evaluation function, it's time to train our own model for summarization.

Training a Summarization Model

We've worked through a lot of details on text summarization and evaluation, so let's put this to use to train a custom text summarization model! For our application, we'll use the SAMSum dataset developed by Samsung, which consists of a collection of dialogues along with brief summaries. In an enterprise setting, these dialogues might represent the interactions between a customer and the support center, so generating accurate summaries can help improve customer service and detect common patterns among customer requests. Let's load it and look at an example:

```
dataset_samsum = load_dataset("samsum")
split_lengths = [len(dataset_samsum[split])for split in dataset_samsum]

print(f"Split lengths: {split_lengths}")
print(f"Features: {dataset_samsum['train'].column_names}")
print("\nDialogue:")
print(dataset_samsum["test"][0]["dialogue"])
print("\nSummary:")
print(dataset_samsum["test"][0]["summary"])
```

```
Split lengths: [14732, 819, 818]
Features: ['id', 'dialogue', 'summary']

Dialogue:
Hannah: Hey, do you have Betty's number?
Amanda: Lemme check
Hannah: <file_gif>
Amanda: Sorry, can't find it.
Amanda: Ask Larry
Amanda: He called her last time we were at the park together
Hannah: I don't know him well
Hannah: <file_gif>
Amanda: Don't be shy, he's very nice
Hannah: If you say so..
Hannah: I'd rather you texted him
Amanda: Just text him 🙂
Hannah: Urgh.. Alright
Hannah: Bye
Amanda: Bye bye

Summary:
Hannah needs Betty's number but Amanda doesn't have it. She needs to contact
Larry.
```

The dialogues look like what you would expect from a chat via SMS or WhatsApp, including emojis and placeholders for GIFs. The `dialogue` field contains the full text and the `summary` the summarized dialogue. Could a model that was fine-tuned on the CNN/DailyMail dataset deal with that? Let's find out!

Evaluating PEGASUS on SAMSum

First we'll run the same summarization pipeline with PEGASUS to see what the output looks like. We can reuse the code we used for the CNN/DailyMail summary generation:

```
pipe_out = pipe(dataset_samsum["test"][0]["dialogue"])
print("Summary:")
print(pipe_out[0]["summary_text"].replace(" .<n>", ".\n"))
```

```
Summary:
Amanda: Ask Larry Amanda: He called her last time we were at the park together.
Hannah: I'd rather you texted him.
Amanda: Just text him .
```

We can see that the model mostly tries to summarize by extracting the key sentences from the dialogue. This probably worked relatively well on the CNN/DailyMail dataset, but the summaries in SAMSum are more abstract. Let's confirm this by running the full ROUGE evaluation on the test set:

```
score = evaluate_summaries_pegasus(dataset_samsum["test"], rouge_metric, model,
                                   tokenizer, column_text="dialogue",
                                   column_summary="summary", batch_size=8)

rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame(rouge_dict, index=["pegasus"])
```

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.296168	0.087803	0.229604	0.229514

Well, the results aren't great, but this is not unexpected since we've moved quite a bit away from the CNN/DailyMail data distribution. Nevertheless, setting up the evaluation pipeline before training has two advantages: we can directly measure the success of training with the metric and we have a good baseline. Fine-tuning the model on our dataset should result in an immediate improvement in the ROUGE metric, and if that is not the case we'll know something is wrong with our training loop.

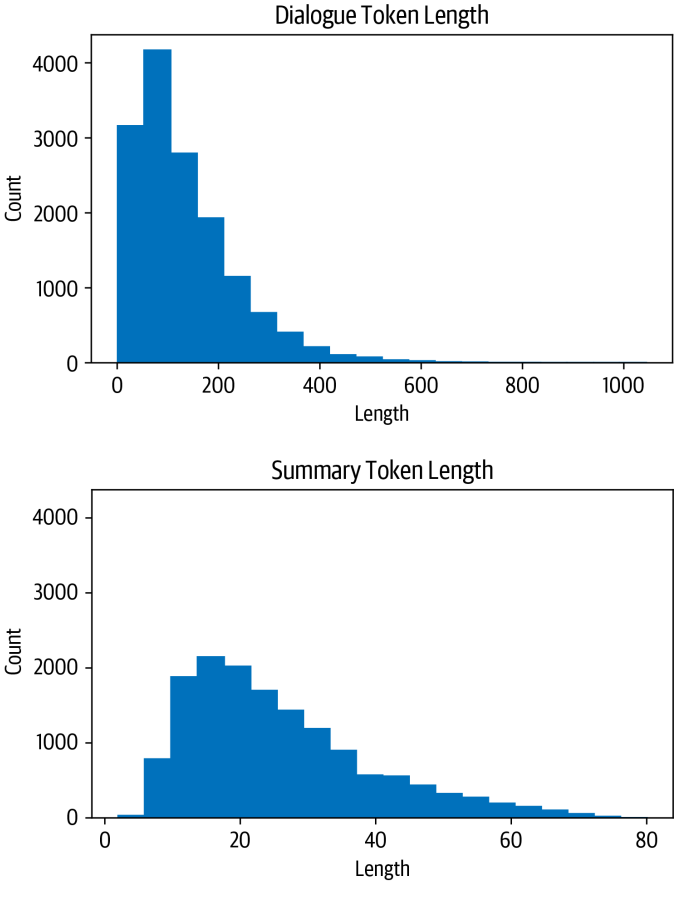
Fine-Tuning PEGASUS

Before we process the data for training, let's have a quick look at the length distribution of the input and outputs:

```
d_len = [len(tokenizer.encode(s)) for s in dataset_samsum["train"]["dialogue"]]
s_len = [len(tokenizer.encode(s)) for s in dataset_samsum["train"]["summary"]]

fig, axes = plt.subplots(1, 2, figsize=(10, 3.5), sharey=True)
axes[0].hist(d_len, bins=20, color="C0", edgecolor="C0")
axes[0].set_title("Dialogue Token Length")
axes[0].set_xlabel("Length")
```

```
axes[0].set_ylabel("Count")
axes[1].hist(s_len, bins=20, color="C0", edgecolor="C0")
axes[1].set_title("Summary Token Length")
axes[1].set_xlabel("Length")
plt.tight_layout()
plt.show()
```



We see that most dialogues are much shorter than the CNN/DailyMail articles, with 100–200 tokens per dialogue. Similarly, the summaries are much shorter, with around 20–40 tokens (the average length of a tweet).

Let's keep those observations in mind as we build the data collator for the `Trainer`. First we need to tokenize the dataset. For now, we'll set the maximum lengths to 1024 and 128 for the dialogues and summaries, respectively:

```
def convert_examples_to_features(example_batch):
    input_encodings = tokenizer(example_batch["dialogue"], max_length=1024,
                                truncation=True)
```

```
        with tokenizer.as_target_tokenizer():
            target_encodings = tokenizer(example_batch["summary"], max_length=128,
                                         truncation=True)

        return {"input_ids": input_encodings["input_ids"],
                "attention_mask": input_encodings["attention_mask"],
                "labels": target_encodings["input_ids"]}

    dataset_samsum_pt = dataset_samsum.map(convert_examples_to_features,
                                           batched=True)
    columns = ["input_ids", "labels", "attention_mask"]
    dataset_samsum_pt.set_format(type="torch", columns=columns)
```

A new thing in the use of the tokenization step is the `tokenizer.as_target_tokenizer()` context. Some models require special tokens in the decoder inputs, so it's important to differentiate between the tokenization of encoder and decoder inputs. In the `with` statement (called a *context manager*), the tokenizer knows that it is tokenizing for the decoder and can process sequences accordingly.

Now, we need to create the data collator. This function is called in the `Trainer` just before the batch is fed through the model. In most cases we can use the default collator, which collects all the tensors from the batch and simply stacks them. For the summarization task we need to not only stack the inputs but also prepare the targets on the decoder side. PEGASUS is an encoder-decoder transformer and thus has the classic seq2seq architecture. In a seq2seq setup, a common approach is to apply "teacher forcing" in the decoder. With this strategy, the decoder receives input tokens (like in decoder-only models such as GPT-2) that consists of the labels shifted by one in addition to the encoder output; so, when making the prediction for the next token the decoder gets the ground truth shifted by one as an input, as illustrated in the following table:

	decoder_input	label
step		
1	[PAD]	Transformers
2	[PAD, Transformers]	are
3	[PAD, Transformers, are]	awesome
4	[PAD, Transformers, are, awesome]	for
5	[PAD, Transformers, are, awesome, for]	text
6	[PAD, Transformers, are, awesome, for, text]	summarization

We shift it by one so that the decoder only sees the previous ground truth labels and not the current or future ones. Shifting alone suffices since the decoder has masked self-attention that masks all inputs at present and in the future.

So, when we prepare our batch, we set up the decoder inputs by shifting the labels to the right by one. After that, we make sure the padding tokens in the labels are ignored by the loss function by setting them to –100. We actually don't have to do this manually, though, since the `DataCollatorForSeq2Seq` comes to the rescue and takes care of all these steps for us:

```
from transformers import DataCollatorForSeq2Seq

seq2seq_data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)
```

Then, as usual, we set up a the `TrainingArguments` for training:

```
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(
    output_dir='pegasus-samsum', num_train_epochs=1, warmup_steps=500,
    per_device_train_batch_size=1, per_device_eval_batch_size=1,
    weight_decay=0.01, logging_steps=10, push_to_hub=True,
    evaluation_strategy='steps', eval_steps=500, save_steps=1e6,
    gradient_accumulation_steps=16)
```

One thing that is different from the previous settings is that new argument, `gradient_accumulation_steps`. Since the model is quite big, we had to set the batch size to 1. However, a batch size that is too small can hurt convergence. To resolve that issue, we can use a nifty technique called *gradient accumulation*. As the name suggests, instead of calculating the gradients of the full batch all at once, we make smaller batches and aggregate the gradients. When we have aggregated enough gradients, we run the optimization step. Naturally this is a bit slower than doing it in one pass, but it saves us a lot of GPU memory.

Let's now make sure that we are logged in to Hugging Face so we can push the model to the Hub after training:

```
from huggingface_hub import notebook_login

notebook_login()
```

We have now everything we need to initialize the trainer with the model, tokenizer, training arguments, and data collator, as well as the training and evaluation sets:

```
trainer = Trainer(model=model, args=training_args,
                  tokenizer=tokenizer, data_collator=seq2seq_data_collator,
                  train_dataset=dataset_samsum_pt["train"],
                  eval_dataset=dataset_samsum_pt["validation"])
```

We are ready for training. After training, we can directly run the evaluation function on the test set to see how well the model performs:

```
trainer.train()
score = evaluate_summaries_pegasus(
    dataset_samsum["test"], rouge_metric, trainer.model, tokenizer,
    batch_size=2, column_text="dialogue", column_summary="summary")
```

```
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame(rouge_dict, index=[f"pegasus"])
```

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.427614	0.200571	0.340648	0.340738

We see that the ROUGE scores improved considerably over the model without fine-tuning, so even though the previous model was also trained for summarization, it was not well adapted for the new domain. Let's push our model to the Hub:

```
trainer.push_to_hub("Training complete!")
```

In the next section we'll use the model to generate a few summaries for us.

You can also evaluate the generations as part of the training loop: use the extension of `TrainingArguments` called `Seq2SeqTrainingArguments` and specify `predict_with_generate=True`. Pass it to the dedicated `Trainer` called `Seq2SeqTrainer`, which then uses the `generate()` function instead of the model's forward pass to create predictions for evaluation. Give it a try!

Generating Dialogue Summaries

Looking at the losses and ROUGE scores, it seems the model is showing a significant improvement over the original model trained on CNN/DailyMail only. Let's see what a summary generated on a sample from the test set looks like:

```
gen_kwargs = {"length_penalty": 0.8, "num_beams":8, "max_length": 128}
sample_text = dataset_samsum["test"][0]["dialogue"]
reference = dataset_samsum["test"][0]["summary"]
pipe = pipeline("summarization", model="transformersbook/pegasus-samsum")

print("Dialogue:")
print(sample_text)
print("\nReference Summary:")
print(reference)
print("\nModel Summary:")
print(pipe(sample_text, **gen_kwargs)[0]["summary_text"])
```

```
Dialogue:
Hannah: Hey, do you have Betty's number?
Amanda: Lemme check
Hannah: <file_gif>
Amanda: Sorry, can't find it.
Amanda: Ask Larry
Amanda: He called her last time we were at the park together
Hannah: I don't know him well
Hannah: <file_gif>
```

```
Amanda: Don't be shy, he's very nice
Hannah: If you say so..
Hannah: I'd rather you texted him
Amanda: Just text him 🙂
Hannah: Urgh.. Alright
Hannah: Bye
Amanda: Bye bye

Reference Summary:
Hannah needs Betty's number but Amanda doesn't have it. She needs to contact
Larry.

Model Summary:
Amanda can't find Betty's number. Larry called Betty last time they were at the
park together. Hannah wants Amanda to text Larry instead of calling Betty.
```

That looks much more like the reference summary. It seems the model has learned to synthesize the dialogue into a summary without just extracting passages. Now, the ultimate test: how well does the model work on a custom input?

```
custom_dialogue = """\
Thom: Hi guys, have you heard of transformers?
Lewis: Yes, I used them recently!
Leandro: Indeed, there is a great library by Hugging Face.
Thom: I know, I helped build it ;)
Lewis: Cool, maybe we should write a book about it. What do you think?
Leandro: Great idea, how hard can it be?!
Thom: I am in!
Lewis: Awesome, let's do it together!
"""
print(pipe(custom_dialogue, **gen_kwargs)[0]["summary_text"])
```

```
Thom, Lewis and Leandro are going to write a book about transformers. Thom
helped build a library by Hugging Face. They are going to do it together.
```

The generated summary of the custom dialogue makes sense. It summarizes well that all the people in the discussion want to write the book together and does not simply extract single sentences. For example, it synthesizes the third and fourth lines into a logical combination.

Conclusion

Text summarization poses some unique challenges compared to other tasks that can be framed as classification tasks, like sentiment analysis, named entity recognition, or question answering. Conventional metrics such as accuracy do not reflect the quality of the generated text. As we saw, the BLEU and ROUGE metrics can better evaluate generated texts; however, human judgment remains the best measure.

A common question when working with summarization models is how we can summarize documents where the texts are longer than the model's context length.

Unfortunately, there is no single strategy to solve this problem, and to date this is still an open and active research question. For example, recent work by OpenAI showed how to scale summarization by applying it recursively to long documents and using human feedback in the loop.[6]

In the next chapter we'll look at question answering, which is the task of providing an answer to a question based on a text passage. In contrast to summarization, with this task there exist good strategies to deal with long or many documents, and we'll show you how to scale question answering to thousands of documents.

6 J. Wu et al., "Recursively Summarizing Books with Human Feedback", (2021).