



Hugging Face

NLP with Hugging Face

Reference and Docs located on: huggingface.co

The course is released under the permissive [Apache 2 license](https://www.apache.org/licenses/LICENSE-2.0)

```
@misc{huggingfacecourse,  
  author = {Hugging Face},  
  title = {The Hugging Face Course, 2022},  
  howpublished = "\url{https://huggingface.co/course}",  
  year = {2022},  
  note = "[Online; accessed <today>]"  
}
```

NLP Course DOC with Completed Code

Python 3.11 or above with PyTorch Examples

Modified for Chapters 1 – 5 for ONLC course code: LDLMH2

Setup | Introduction

Welcome to the Hugging Face course! This introduction will guide you through setting up a working environment. If you're just starting the course, we recommend you first take a look at [Chapter 1](#), then come back and set up your environment so you can try the code yourself.

All the libraries that we'll be using in this course are available as Python packages, so here we'll show you how to set up a Python environment and install the specific libraries you'll need.

We'll cover two ways of setting up your working environment, using a Colab notebook or a Python virtual environment. Feel free to choose the one that resonates with you the most. For beginners, we strongly recommend that you get started by using a Colab notebook.

Note that we will not be covering the Windows system. If you're running on Windows, we recommend following along using a Colab notebook. If you're using a Linux distribution or macOS, you can use either approach described here.

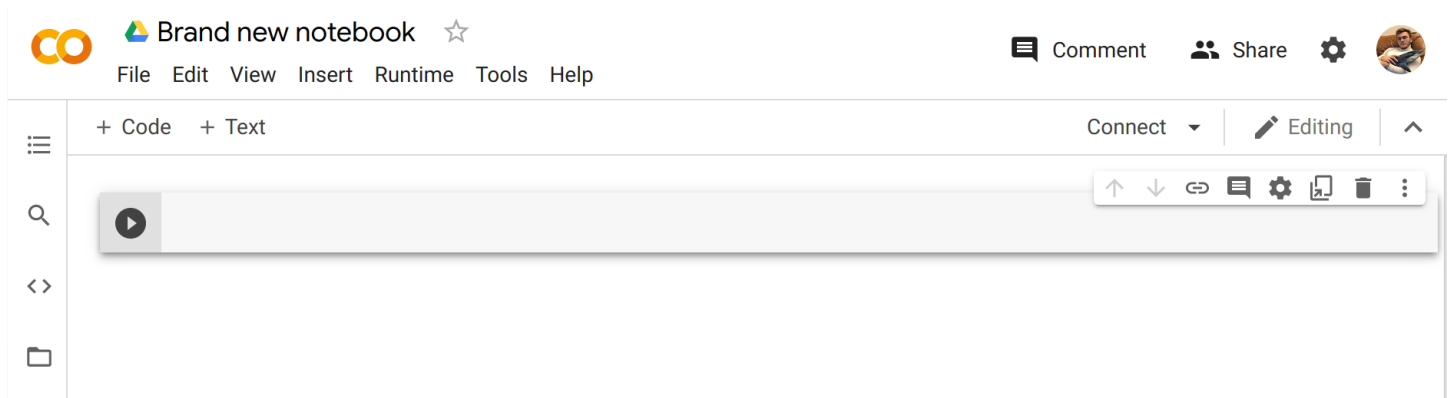
Most of the course relies on you having a Hugging Face account. We recommend creating one now: [create an account](#).

Using a Google Colab notebook

Using a Colab notebook is the simplest possible setup; boot up a notebook in your browser and get straight to coding!

If you're not familiar with Colab, we recommend you start by following the [introduction](#). Colab allows you to use some accelerating hardware, like GPUs or TPUs, and it is free for smaller workloads.

Once you're comfortable moving around in Colab, create a new notebook and get started with the setup:



The next step is to install the libraries that we'll be using in this course. We'll use `pip` for the installation, which is the package manager for Python. In notebooks, you can run system commands by preceding them with the `!` character, so you can install the 🤖 Transformers library as follows:

```
!pip install transformers
```

You can make sure the package was correctly installed by importing it within your Python runtime:

```
import transformers
```

This installs a very light version of 🤖 Transformers. In particular, no specific machine learning frameworks (like PyTorch or TensorFlow) are installed. Since we'll be using a lot of different features of the library, we recommend installing the development version, which comes with all the required dependencies for pretty much any imaginable use case:

```
!pip install transformers[sentencepiece]
```

This will take a bit of time, but then you'll be ready to go for the rest of the course!

Using a Python virtual environment

If you prefer to use a Python virtual environment, the first step is to install Python on your system. We recommend following [this guide](#) to get started.

Once you have Python installed, you should be able to run Python commands in your terminal. You can start by running the following command to ensure that it is correctly installed before proceeding to the next steps: `python --version`. This should print out the Python version now available on your system.

When running a Python command in your terminal, such as `python --version`, you should think of the program running your command as the “main” Python on your system. We recommend keeping this main installation free of any packages, and using it to create separate environments for each application you work on — this way, each application can have its own dependencies and packages, and you won't need to worry about potential compatibility issues with other applications.

In Python this is done with [virtual environments](#), which are self-contained directory trees that each contain a Python installation with a particular Python version alongside all the packages the

application needs. Creating such a virtual environment can be done with a number of different tools, but we'll use the official Python package for that purpose, which is called [venv](#).

First, create the directory you'd like your application to live in — for example, you might want to make a new directory called `transformers-course` at the root of your home directory:

```
mkdir ~/transformers-course
```

```
cd ~/transformers-course
```

From inside this directory, create a virtual environment using the Python `venv` module:

```
python -m venv .env
```

You should now have a directory called `.env` in your otherwise empty folder:

```
ls -a
```

```
.  ..  .env
```

You can jump in and out of your virtual environment with the `activate` and `deactivate` scripts:

```
# Activate the virtual environment
```

```
source .env/bin/activate
```

```
# Deactivate the virtual environment
```

```
source .env/bin/deactivate
```

You can make sure that the environment is activated by running the `which python` command: if it points to the virtual environment, then you have successfully activated it!

```
which python
```

```
/home/<user>/transformers-course/.env/bin/python
```

Installing dependencies

As in the previous section on using Google Colab instances, you'll now need to install the packages required to continue. Again, you can install the development version of 🤖 Transformers using the `pip` package manager:

```
pip install "transformers[sentencepiece]"
```

You're now all set up and ready to go!

Transformers | Introduction

Welcome to the 😊 Course!

This course will teach you about natural language processing (NLP) using libraries from the [Hugging Face](#) ecosystem — [🤗 Transformers](#), [🤗 Datasets](#), [🤗 Tokenizers](#), and [🤗 Accelerate](#) — as well as the [Hugging Face Hub](#). It's completely free and without ads.

This course:

- Requires a good knowledge of Python
- Is better taken after an introductory deep learning course, such as [fast.ai's Practical Deep Learning for Coders](#) or one of the programs developed by [DeepLearning.AI](#)
- Does not expect prior [PyTorch](#) or [TensorFlow](#) knowledge, though some familiarity with either of those will help

Who are we?

About the authors:

[Abubakar Abid](#) completed his PhD at Stanford in applied machine learning. During his PhD, he founded [Gradio](#), an open-source Python library that has been used to build over 600,000 machine learning demos. Gradio was acquired by Hugging Face, which is where Abubakar now serves as a machine learning team lead.

[Matthew Carrigan](#) is a Machine Learning Engineer at Hugging Face. He lives in Dublin, Ireland and previously worked as an ML engineer at Parse.ly and before that as a post-doctoral researcher at Trinity College Dublin. He does not believe we're going to get to AGI by scaling existing architectures, but has high hopes for robot immortality regardless.

[Lysandre Debut](#) is a Machine Learning Engineer at Hugging Face and has been working on the 🤗 Transformers library since the very early development stages. His aim is to make NLP accessible for everyone by developing tools with a very simple API.

[Sylvain Gugger](#) is a Research Engineer at Hugging Face and one of the core maintainers of the 🤗 Transformers library. Previously he was a Research Scientist at fast.ai, and he co-wrote [Deep Learning for Coders with fastai and PyTorch](#) with Jeremy Howard. The main focus of his research is on making deep learning more accessible, by designing and improving techniques that allow models to train fast on limited resources.

[Dawood Khan](#) is a Machine Learning Engineer at Hugging Face. He's from NYC and graduated from New York University studying Computer Science. After working as an iOS Engineer for a few years, Dawood quit to start Gradio with his fellow co-founders. Gradio was eventually acquired by Hugging Face.

[Merve Noyan](#) is a developer advocate at Hugging Face, working on developing tools and building content around them to democratize machine learning for everyone.

[Lucile Saulnier](#) is a machine learning engineer at Hugging Face, developing and supporting the use of open source tools. She is also actively involved in many research projects in the field of Natural Language Processing such as collaborative training and BigScience.

[Lewis Tunstall](#) is a machine learning engineer at Hugging Face, focused on developing open-source tools and making them accessible to the wider community. He is also a co-author of the O'Reilly book [Natural Language Processing with Transformers](#).


[Leandro von Werra](#) is a machine learning engineer in the open-source team at Hugging Face and also a co-author of the O'Reilly book [Natural Language Processing with Transformers](#). He has several years of industry experience bringing NLP projects to production by working across the whole machine learning stack..

FAQ

Here are some answers to frequently asked questions:

- **Does taking this course lead to a certification?** Currently we do not have any certification for this course. However, we are working on a certification program for the Hugging Face ecosystem — stay tuned!
- **How much time should I spend on this course?** Each chapter in this course is designed to be completed in 1 week, with approximately 6-8 hours of work per week. However, you can take as much time as you need to complete the course.
- **Where can I ask a question if I have one?** If you have a question about any section of the course, just click on the "Ask a question" banner at the top of the page to be automatically redirected to the right section of the [Hugging Face forums](#):

Natural Language Processing



[Ask a question](#)

Before jumping into Transformer models, let's do a quick overview of what natural language processing is and why we care about it.

Note that a list of [project ideas](#) is also available on the forums if you wish to practice more once you have completed the course.

- **Where can I get the code for the course?** For each section, click on the banner at the top of the page to run the code in either Google Colab or Amazon SageMaker Studio Lab:

Transformers, what can they do?



In this section, we will look at what Transformer models can do and use our first tool from the 🤗 Transformers library: the `pipeline()` function.

The Jupyter notebooks containing all the code from the course are hosted on the [huggingface/notebooks](#) repo. If you wish to generate them locally, check out the instructions in the [course](#) repo on GitHub.

- **How can I contribute to the course?** There are many ways to contribute to the course! If you find a typo or a bug, please open an issue on the [course](#) repo. If you would like to help translate the course into your native language, check out the instructions [here](#).
- **What were the choices made for each translation?** Each translation has a glossary and `TRANSLATING.txt` file that details the choices that were made for machine learning jargon etc. You can find an example for German [here](#).
- **Can I reuse this course?** Of course! The course is released under the permissive [Apache 2 license](#). This means that you must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. If you would like to cite the course, please use the following BibTeX:

```
@misc{huggingfacecourse,  
  
  author = {Hugging Face},  
  
  title = {The Hugging Face Course, 2022},  
  
  howpublished = "\url{https://huggingface.co/course}",  
  
  year = {2022},
```



```
note = "[Online; accessed <today>]"
```

```
}
```

Let's Go

Are you ready to roll? In this chapter, you will learn:

- How to use the `pipeline()` function to solve NLP tasks such as text generation and classification
- About the Transformer architecture
- How to distinguish between encoder, decoder, and encoder-decoder architectures and use cases

Natural Language Processing

Before jumping into Transformer models, let's do a quick overview of what natural language processing is and why we care about it.

What is NLP?

NLP is a field of linguistics and machine learning focused on understanding everything related to human language. The aim of NLP tasks is not only to understand single words individually, but to be able to understand the context of those words.

The following is a list of common NLP tasks, with some examples of each:


- **Classifying whole sentences:** Getting the sentiment of a review, detecting if an email is spam, determining if a sentence is grammatically correct or whether two sentences are logically related or not
- **Classifying each word in a sentence:** Identifying the grammatical components of a sentence (noun, verb, adjective), or the named entities (person, location, organization)
- **Generating text content:** Completing a prompt with auto-generated text, filling in the blanks in a text with masked words
- **Extracting an answer from a text:** Given a question and a context, extracting the answer to the question based on the information provided in the context
- **Generating a new sentence from an input text:** Translating a text into another language, summarizing a text


NLP isn't limited to written text though. It also tackles complex challenges in speech recognition and computer vision, such as generating a transcript of an audio sample or a description of an image.

Why is it challenging?

Computers don't process information in the same way as humans. For example, when we read the sentence "I am hungry," we can easily understand its meaning. Similarly, given two sentences such as "I am hungry" and "I am sad," we're able to easily determine how similar they are. For machine learning (ML) models, such tasks are more difficult. The text needs to be processed in a way that enables the model to learn from it. And because language is complex, we need to think carefully about how this processing must be done. There has been a lot of research done on how to represent text, and we will look at some methods in the next chapter.

Transformers, what can they do?

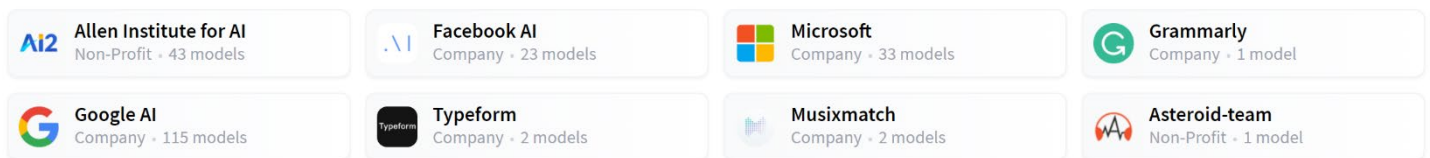
In this section, we will look at what Transformer models can do and use our first tool from the  Transformers library: the `pipeline()` function.

 See that *Open in Colab* button on the top right? Click on it to open a Google Colab notebook with all the code samples of this section. This button will be present in any section containing code examples. If you want to run the examples locally, we recommend taking a look at the [setup](#).


Transformers are everywhere!

Transformer models are used to solve all kinds of NLP tasks, like the ones mentioned in the previous section. Here are some of the companies and organizations using Hugging Face and Transformer models, who also contribute back to the community by sharing their models:

More than 2,000 organizations are using Hugging Face




The [Hugging Face Transformers library](#) provides the functionality to create and use those shared models. The [Model Hub](#) contains thousands of pretrained models that anyone can download and use. You can also upload your own models to the Hub!

 The Hugging Face Hub is not limited to Transformer models. Anyone can share any kind of models or datasets they want! [Create a huggingface.co](#) account to benefit from all available features!

Before diving into how Transformer models work under the hood, let's look at a few examples of how they can be used to solve some interesting NLP problems.

Working with pipelines

The most basic object in the  Transformers library is the `pipeline()` function. It connects a model with its necessary preprocessing and postprocessing steps, allowing us to directly input any text and get an intelligible answer:

```
from transformers import pipeline
```

```
classifier = pipeline("sentiment-analysis")
```

```
classifier("I've been waiting for a HuggingFace course my whole life.")
```

```
[{'label': 'POSITIVE', 'score': 0.9598047137260437}]
```

We can even pass several sentences!

```
classifier(  
    ["I've been waiting for a HuggingFace course my whole life.", "I hate this so  
much!"]  
)
```

```
[{'label': 'POSITIVE', 'score': 0.9598047137260437},  
 {'label': 'NEGATIVE', 'score': 0.9994558095932007}]
```

By default, this pipeline selects a particular pretrained model that has been fine-tuned for sentiment analysis in English. The model is downloaded and cached when you create the `classifier` object. If you rerun the command, the cached model will be used instead and there is no need to download the model again.

There are three main steps involved when you pass some text to a pipeline:

1. The text is preprocessed into a format the model can understand.

2. The preprocessed inputs are passed to the model.
3. The predictions of the model are post-processed, so you can make sense of them.

Some of the currently [available pipelines](#) are:

- `feature-extraction` (get the vector representation of a text)
- `fill-mask`
- `ner` (named entity recognition)
- `question-answering`
- `sentiment-analysis`
- `summarization`
- `text-generation`
- `translation`
- `zero-shot-classification`

Let's have a look at a few of these!

Zero-shot classification

We'll start by tackling a more challenging task where we need to classify texts that haven't been labelled. This is a common scenario in real-world projects because annotating text is usually time-consuming and requires domain expertise. For this use case, the `zero-shot-classification` pipeline is very powerful: it allows you to specify which labels to use for the classification, so you don't have to rely on the labels of the pretrained model. You've already seen how the model can classify a sentence as positive or negative using those two labels — but it can also classify the text using any other set of labels you like.

```
from transformers import pipeline

classifier = pipeline("zero-shot-classification")

classifier(
    "This is a course about the Transformers library",
    candidate_labels=["education", "politics", "business"],
)
```

```
{'sequence': 'This is a course about the Transformers library',  
'labels': ['education', 'business', 'politics'],  
'scores': [0.8445963859558105, 0.111976258456707, 0.043427448719739914]}
```

This pipeline is called *zero-shot* because you don't need to fine-tune the model on your data to use it. It can directly return probability scores for any list of labels you want!

 **Try it out!** Play around with your own sequences and labels and see how the model behaves.

Text generation


Now let's see how to use a pipeline to generate some text. The main idea here is that you provide a prompt and the model will auto-complete it by generating the remaining text. This is similar to the predictive text feature that is found on many phones. Text generation involves randomness, so it's normal if you don't get the same results as shown below.

```
from transformers import pipeline  
  
generator = pipeline("text-generation")  
  
generator("In this course, we will teach you how to")
```

```
[{'generated_text': 'In this course, we will teach you how to understand and use '  
'data flow and data interchange when handling user data. We '  
'will be working with one or more of the most commonly used '  
'data flows – data flows of various types, as seen by the '}
```

```
'HTTP' }]
```

You can control how many different sequences are generated with the argument `num_return_sequences` and the total length of the output text with the argument `max_length`.

 **Try it out!** Use the `num_return_sequences` and `max_length` arguments to generate two sentences of 15 words each.

Using any model from the Hub in a pipeline

The previous examples used the default model for the task at hand, but you can also choose a particular model from the Hub to use in a pipeline for a specific task — say, text generation. Go to the [Model Hub](#) and click on the corresponding tag on the left to display only the supported models for that task. You should get to a page like [this one](#).

Let's try the [distilgpt2](#) model! Here's how to load it in the same pipeline as before:

```
from transformers import pipeline

generator = pipeline("text-generation", model="distilgpt2")

generator(

    "In this course, we will teach you how to",

    max_length=30,

    num_return_sequences=2,


)
```

```
[{'generated_text': 'In this course, we will teach you how to manipulate the world and '}]
```

```
'move your mental and physical capabilities to your advantage.'}],
{'generated_text': 'In this course, we will teach you how to become an expert and '
                    'practice realtime, and with a hands on experience on both real '
                    'time and real'}]
```

You can refine your search for a model by clicking on the language tags, and pick a model that will generate text in another language. The Model Hub even contains checkpoints for multilingual models that support several languages.

Once you select a model by clicking on it, you'll see that there is a widget enabling you to try it directly online. This way you can quickly test the model's capabilities before downloading it.

 **Try it out!** Use the filters to find a text generation model for another language. Feel free to play with the widget and use it in a pipeline!

The Inference API

All the models can be tested directly through your browser using the Inference API, which is available on the Hugging Face [website](#). You can play with the model directly on this page by inputting custom text and watching the model process the input data.

The Inference API that powers the widget is also available as a paid product, which comes in handy if you need it for your workflows. See the [pricing page](#) for more details.

Mask filling

The next pipeline you'll try is `fill-mask`. The idea of this task is to fill in the blanks in a given text:

```
from transformers import pipeline


unmasker = pipeline("fill-mask")

unmasker("This course will teach you all about <mask> models.", top_k=2)
```



```
[{'sequence': 'This course will teach you all about mathematical models.',  
  'score': 0.19619831442832947,  
  'token': 30412,  
  'token_str': ' mathematical'},  
 {'sequence': 'This course will teach you all about computational models.',  
  'score': 0.04052725434303284,  
  'token': 38163,  
  'token_str': ' computational'}]
```

The `top_k` argument controls how many possibilities you want to be displayed. Note that here the model fills in the special `<mask>` word, which is often referred to as a *mask token*. Other mask-filling models might have different mask tokens, so it's always good to verify the proper mask word when exploring other models. One way to check it is by looking at the mask word used in the widget.

 **Try it out!** Search for the `bert-base-cased` model on the Hub and identify its mask word in the Inference API widget. What does this model predict for the sentence in our pipeline example above?

Named entity recognition


Named entity recognition (NER) is a task where the model has to find which parts of the input text correspond to entities such as persons, locations, or organizations. Let's look at an example:

```
from transformers import pipeline  
  
ner = pipeline("ner", grouped_entities=True)  
  
ner("My name is Sylvain and I work at Hugging Face in Brooklyn.")
```

```
[{'entity_group': 'PER', 'score': 0.99816, 'word': 'Sylvain', 'start': 11, 'end': 18},  
  
 {'entity_group': 'ORG', 'score': 0.97960, 'word': 'Hugging Face', 'start': 33, 'end': 45},  
  
 {'entity_group': 'LOC', 'score': 0.99321, 'word': 'Brooklyn', 'start': 49, 'end': 57}  
  
 ]
```

Here the model correctly identified that Sylvain is a person (PER), Hugging Face an organization (ORG), and Brooklyn a location (LOC).

We pass the option `grouped_entities=True` in the pipeline creation function to tell the pipeline to regroup together the parts of the sentence that correspond to the same entity: here the model correctly grouped “Hugging” and “Face” as a single organization, even though the name consists of multiple words. In fact, as we will see in the next chapter, the preprocessing even splits some words into smaller parts. For instance, `Sylvain` is split into four pieces: `S`, `##yl`, `##va`, and `##in`. In the post-processing step, the pipeline successfully regrouped those pieces.

 **Try it out!** Search the Model Hub for a model able to do part-of-speech tagging (usually abbreviated as POS) in English. What does this model predict for the sentence in the example above?

Question answering

The `question-answering` pipeline answers questions using information from a given context:

```
from transformers import pipeline  
  
question_answerer = pipeline("question-answering")  
  
question_answerer(  
    question="Where do I work?",
```

```
context="My name is Sylvain and I work at Hugging Face in Brooklyn",
```

```
)
```

```
{'score': 0.6385916471481323, 'start': 33, 'end': 45, 'answer': 'Hugging Face'}
```

Note that this pipeline works by extracting information from the provided context; it does not generate the answer.

Summarization

Summarization is the task of reducing a text into a shorter text while keeping all (or most) of the important aspects referenced in the text. Here's an example:

```
from transformers import pipeline
```

```
summarizer = pipeline("summarization")
```

```
summarizer(
```

```
    """
```

```
    America has changed dramatically during recent years. Not only has the number of
```

```
    graduates in traditional engineering disciplines such as mechanical, civil,
```

```
    electrical, chemical, and aeronautical engineering declined, but in most of
```

```
    the premier American universities engineering curricula now concentrate on
```

```
    and encourage largely the study of engineering science. As a result, there
```

```
    are declining offerings in engineering subjects dealing with infrastructure,
```

the environment, and related issues, and greater concentration on high technology subjects, largely supporting increasingly complex scientific developments. While the latter is important, it should not be at the expense of more traditional engineering.

Rapidly developing economies such as China and India, as well as other industrial countries in Europe and Asia, continue to encourage and advance the teaching of engineering. Both China and India, respectively, graduate six and eight times as many traditional engineers as does the United States.

Other industrial countries at minimum maintain their output, while America suffers an increasingly serious decline in the number of engineering graduates and a lack of well-educated engineers.

"""

)

```
[{'summary_text': ' America has changed dramatically during recent years . The '
                    'number of engineering graduates in the U.S. has declined in '
                    'traditional engineering disciplines such as mechanical, civil '
                    ', electrical, chemical, and aeronautical engineering . Rapidly '
                    'developing economies such as China and India, as well as other '}
```

```
'industrial countries in Europe and Asia, continue to encourage '  
'and advance engineering .'}]
```

Like with text generation, you can specify a `max_length` or a `min_length` for the result.


Translation

For translation, you can use a default model if you provide a language pair in the task name (such as `"translation_en_to_fr"`), but the easiest way is to pick the model you want to use on the [Model Hub](#). Here we'll try translating from French to English:

```
from transformers import pipeline  
  
translator = pipeline("translation", model="Helsinki-NLP/opus-mt-fr-en")  
  
translator("Ce cours est produit par Hugging Face.")
```

```
[{'translation_text': 'This course is produced by Hugging Face.'}]
```

Like with text generation and summarization, you can specify a `max_length` or a `min_length` for the result.

 **Try it out!** Search for translation models in other languages and try to translate the previous sentence into a few different languages.

The pipelines shown so far are mostly for demonstrative purposes. They were programmed for specific tasks and cannot perform variations of them. In the next chapter, you'll learn what's inside a `pipeline()` function and how to customize its behavior.

How do Transformers work?

In this section, we will take a high-level look at the architecture of Transformer models.

A bit of Transformer history

Here are some reference points in the (short) history of Transformer models:

The [Transformer architecture](#) was introduced in June 2017. The focus of the original research was on translation tasks. This was followed by the introduction of several influential models, including:

- **June 2018:** [GPT](#), the first pretrained Transformer model, used for fine-tuning on various NLP tasks and obtained state-of-the-art results
- **October 2018:** [BERT](#), another large pretrained model, this one designed to produce better summaries of sentences (more on this in the next chapter!)
- **February 2019:** [GPT-2](#), an improved (and bigger) version of GPT that was not immediately publicly released due to ethical concerns
- **October 2019:** [DistilBERT](#), a distilled version of BERT that is 60% faster, 40% lighter in memory, and still retains 97% of BERT's performance
- **October 2019:** [BART](#) and [T5](#), two large pretrained models using the same architecture as the original Transformer model (the first to do so)
- **May 2020,** [GPT-3](#), an even bigger version of GPT-2 that is able to perform well on a variety of tasks without the need for fine-tuning (called *zero-shot learning*)

This list is far from comprehensive, and is just meant to highlight a few of the different kinds of Transformer models. Broadly, they can be grouped into three categories:

- GPT-like (also called *auto-regressive* Transformer models)
- BERT-like (also called *auto-encoding* Transformer models)
- BART/T5-like (also called *sequence-to-sequence* Transformer models)

We will dive into these families in more depth later on.

Transformers are language models

All the Transformer models mentioned above (GPT, BERT, BART, T5, etc.) have been trained as *language models*. This means they have been trained on large amounts of raw text in a self-supervised fashion. Self-supervised learning is a type of training in which the objective is automatically computed from the inputs of the model. That means that humans are not needed to label the data!

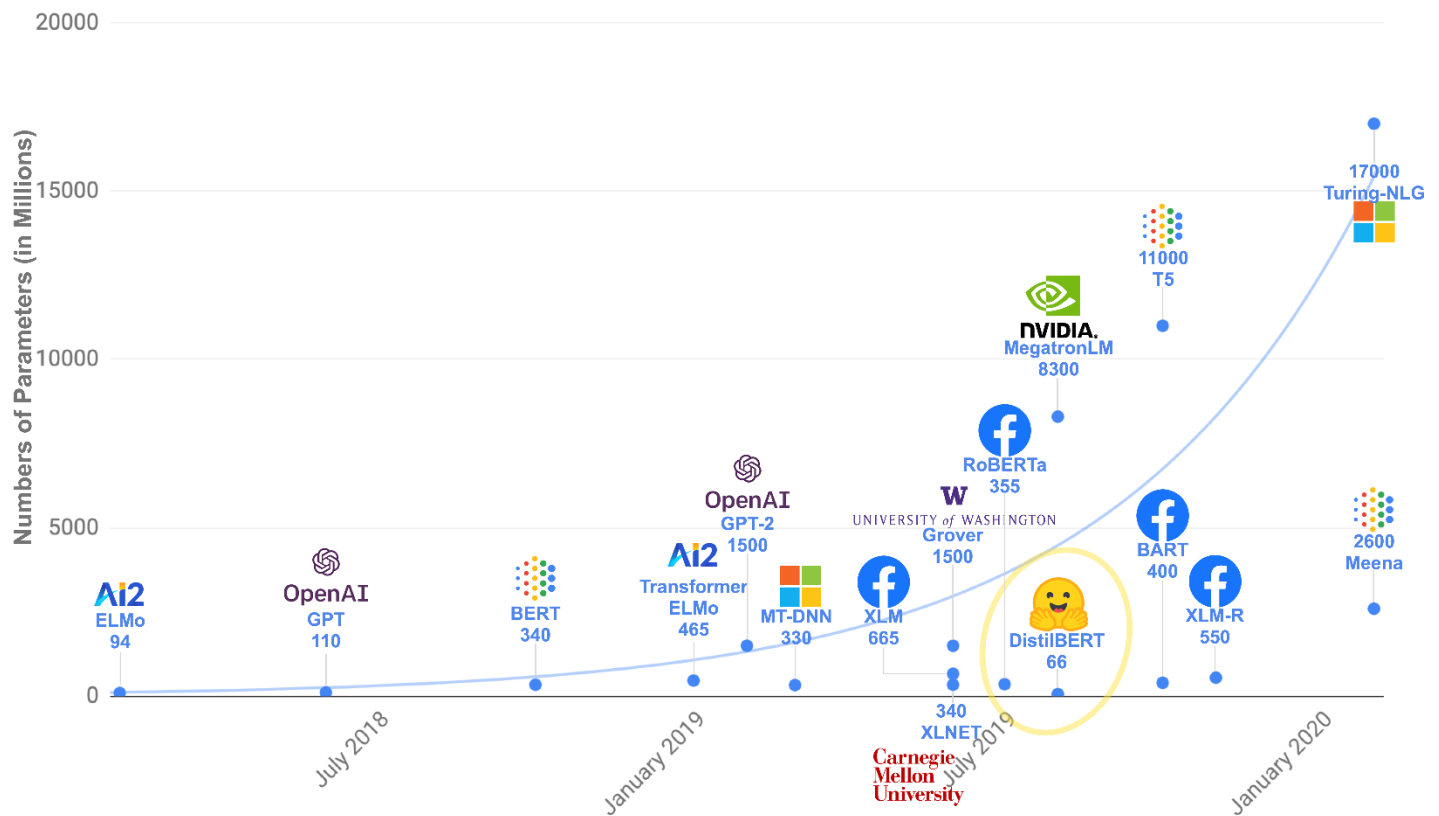
This type of model develops a statistical understanding of the language it has been trained on, but it's not very useful for specific practical tasks. Because of this, the general pretrained model then goes through a process called *transfer learning*. During this process, the model is fine-tuned in a supervised way — that is, using human-annotated labels — on a given task.

An example of a task is predicting the next word in a sentence having read the n previous words. This is called *causal language modeling* because the output depends on the past and present inputs, but not the future ones.

Another example is *masked language modeling*, in which the model predicts a masked word in the sentence.

Transformers are big models

Apart from a few outliers (like DistilBERT), the general strategy to achieve better performance is by increasing the models' sizes as well as the amount of data they are pretrained on.



Unfortunately, training a model, especially a large one, requires a large amount of data. This becomes very costly in terms of time and compute resources. It even translates to environmental impact, as can be seen in the following graph.

And this is showing a project for a (very big) model led by a team consciously trying to reduce the environmental impact of pretraining. The footprint of running lots of trials to get the best hyperparameters would be even higher.

Imagine if each time a research team, a student organization, or a company wanted to train a model, it did so from scratch. This would lead to huge, unnecessary global costs!

This is why sharing language models is paramount: sharing the trained weights and building on top of already trained weights reduces the overall compute cost and carbon footprint of the community.

By the way, you can evaluate the carbon footprint of your models' training through several tools. For example [ML CO2 Impact](#) or [Code Carbon](#) which is integrated in 🤗 Transformers. To learn more about this, you can read this [blog post](#) which will show you how to generate an `emissions.csv` file with an estimate of the footprint of your training, as well as the [documentation](#) of 🤗 Transformers addressing this topic.

Transfer Learning

Pretraining is the act of training a model from scratch: the weights are randomly initialized, and the training starts without any prior knowledge.

This pretraining is usually done on very large amounts of data. Therefore, it requires a very large corpus of data, and training can take up to several weeks.

Fine-tuning, on the other hand, is the training done **after** a model has been pretrained. To perform fine-tuning, you first acquire a pretrained language model, then perform additional training with a dataset specific to your task. Wait — why not simply train directly for the final task? There are a couple of reasons:

- The pretrained model was already trained on a dataset that has some similarities with the fine-tuning dataset. The fine-tuning process is thus able to take advantage of knowledge acquired by the initial model during pretraining (for instance, with NLP problems, the pretrained model will have some kind of statistical understanding of the language you are using for your task).
- Since the pretrained model was already trained on lots of data, the fine-tuning requires way less data to get decent results.
- For the same reason, the amount of time and resources needed to get good results are much lower.

For example, one could leverage a pretrained model trained on the English language and then fine-tune it on an arXiv corpus, resulting in a science/research-based model. The fine-tuning will only require a limited amount of data: the knowledge the pretrained model has acquired is “transferred,” hence the term *transfer learning*.

Fine-tuning a model therefore has lower time, data, financial, and environmental costs. It is also quicker and easier to iterate over different fine-tuning schemes, as the training is less constraining than a full pretraining.

This process will also achieve better results than training from scratch (unless you have lots of data), which is why you should always try to leverage a pretrained model — one as close as possible to the task you have at hand — and fine-tune it.

General architecture

In this section, we’ll go over the general architecture of the Transformer model. Don’t worry if you don’t understand some of the concepts; there are detailed sections later covering each of the components.

Introduction

The model is primarily composed of two blocks:

- **Encoder (left):** The encoder receives an input and builds a representation of it (its features). This means that the model is optimized to acquire understanding from the input.
- **Decoder (right):** The decoder uses the encoder's representation (features) along with other inputs to generate a target sequence. This means that the model is optimized for generating outputs.

Each of these parts can be used independently, depending on the task:

- **Encoder-only models:** Good for tasks that require understanding of the input, such as sentence classification and named entity recognition.
- **Decoder-only models:** Good for generative tasks such as text generation.
- **Encoder-decoder models** or **sequence-to-sequence models:** Good for generative tasks that require an input, such as translation or summarization.

We will dive into those architectures independently in later sections.

Attention layers

A key feature of Transformer models is that they are built with special layers called *attention layers*. In fact, the title of the paper introducing the Transformer architecture was [“Attention Is All You Need”](#)! We will explore the details of attention layers later in the course; for now, all you need to know is that this layer will tell the model to pay specific attention to certain words in the sentence you passed it (and more or less ignore the others) when dealing with the representation of each word.

To put this into context, consider the task of translating text from English to French. Given the input “You like this course”, a translation model will need to also attend to the adjacent word “You” to get the proper translation for the word “like”, because in French the verb “like” is conjugated differently depending on the subject. The rest of the sentence, however, is not useful for the translation of that word. In the same vein, when translating “this” the model will also need to pay attention to the word “course”, because “this” translates differently depending on whether the associated noun is masculine or feminine. Again, the other words in the sentence will not matter for the translation of “this”. With more complex sentences (and more complex grammar rules), the model would need to pay special attention to words that might appear farther away in the sentence to properly translate each word.

The same concept applies to any task associated with natural language: a word by itself has a meaning, but that meaning is deeply affected by the context, which can be any other word (or words) before or after the word being studied.

Now that you have an idea of what attention layers are all about, let's take a closer look at the Transformer architecture.

The original architecture

The Transformer architecture was originally designed for translation. During training, the encoder receives inputs (sentences) in a certain language, while the decoder receives the same sentences in the desired target language. In the encoder, the attention layers can use all the words in a sentence (since, as we just saw, the translation of a given word can be dependent on what is after as well as before it in the sentence). The decoder, however, works sequentially and can only pay attention to the words in the sentence that it has already translated (so, only the words before the word currently being generated). For example, when we have predicted the first three words of the translated target, we give them to the decoder which then uses all the inputs of the encoder to try to predict the fourth word.

To speed things up during training (when the model has access to target sentences), the decoder is fed the whole target, but it is not allowed to use future words (if it had access to the word at position 2 when trying to predict the word at position 2, the problem would not be very hard!). For instance, when trying to predict the fourth word, the attention layer will only have access to the words in positions 1 to 3.

The original Transformer architecture looked like this, with the encoder on the left and the decoder on the right:

Note that the first attention layer in a decoder block pays attention to all (past) inputs to the decoder, but the second attention layer uses the output of the encoder. It can thus access the whole input sentence to best predict the current word. This is very useful as different languages can have grammatical rules that put the words in different orders, or some context provided later in the sentence may be helpful to determine the best translation of a given word.

The *attention mask* can also be used in the encoder/decoder to prevent the model from paying attention to some special words — for instance, the special padding word used to make all the inputs the same length when batching together sentences.

Architectures vs. checkpoints

As we dive into Transformer models in this course, you'll see mentions of *architectures* and *checkpoints* as well as *models*. These terms all have slightly different meanings:

- **Architecture:** This is the skeleton of the model — the definition of each layer and each operation that happens within the model.
- **Checkpoints:** These are the weights that will be loaded in a given architecture.
- **Model:** This is an umbrella term that isn't as precise as "architecture" or "checkpoint": it can mean both. This course will specify *architecture* or *checkpoint* when it matters to reduce ambiguity.

For example, BERT is an architecture while bert-base-cased, a set of weights trained by the Google team for the first release of BERT, is a checkpoint. However, one can say “the BERT model” and “the bert-base-cased model.”

Encoder models



Encoder models use only the encoder of a Transformer model. At each stage, the attention layers can access all the words in the initial sentence. These models are often characterized as having “bi-directional” attention, and are often called *auto-encoding models*.

The pretraining of these models usually revolves around somehow corrupting a given sentence (for instance, by masking random words in it) and tasking the model with finding or reconstructing the initial sentence.

Encoder models are best suited for tasks requiring an understanding of the full sentence, such as sentence classification, named entity recognition (and more generally word classification), and extractive question answering.

Representatives of this family of models include:

- [ALBERT](#)
- [BERT](#)
- [DistilBERT](#)
- [ELECTRA](#)
- [RoBERTa](#)

Decoder models



Decoder models use only the decoder of a Transformer model. At each stage, for a given word the attention layers can only access the words positioned before it in the sentence. These models are often called *auto-regressive models*.

The pretraining of decoder models usually revolves around predicting the next word in the sentence.

These models are best suited for tasks involving text generation.

Representatives of this family of models include:

- [CTRL](#)
- [GPT](#)
- [GPT-2](#)
- [Transformer XL](#)

Sequence-to-sequence models[sequence-to-sequence-models]



Encoder-decoder models (also called *sequence-to-sequence models*) use both parts of the Transformer architecture. At each stage, the attention layers of the encoder can access all the words in the initial sentence, whereas the attention layers of the decoder can only access the words positioned before a given word in the input.

The pretraining of these models can be done using the objectives of encoder or decoder models, but usually involves something a bit more complex. For instance, [T5](#) is pretrained by replacing random spans of text (that can contain several words) with a single mask special word, and the objective is then to predict the text that this mask word replaces.

Sequence-to-sequence models are best suited for tasks revolving around generating new sentences depending on a given input, such as summarization, translation, or generative question answering.

Representatives of this family of models include:

- [BART](#)
- [mBART](#)
- [Marian](#)
- [T5](#)

Bias and limitations

If your intent is to use a pretrained model or a fine-tuned version in production, please be aware that, while these models are powerful tools, they come with limitations. The biggest of these is that, to enable pretraining on large amounts of data, researchers often scrape all the content they can find, taking the best as well as the worst of what is available on the internet.

To give a quick illustration, let's go back the example of a `fill-mask` pipeline with the BERT model:

```
from transformers import pipeline

unmasker = pipeline("fill-mask", model="bert-base-uncased")

result = unmasker("This man works as a [MASK].")

print([r["token_str"] for r in result])

result = unmasker("This woman works as a [MASK].")

print([r["token_str"] for r in result])
```

```
['lawyer', 'carpenter', 'doctor', 'waiter', 'mechanic']


['nurse', 'waitress', 'teacher', 'maid', 'prostitute']
```

When asked to fill in the missing word in these two sentences, the model gives only one gender-free answer (waiter/waitress). The others are work occupations usually associated with one specific gender — and yes, prostitute ended up in the top 5 possibilities the model associates with “woman” and “work.” This happens even though BERT is one of the rare Transformer models not built by scraping

data from all over the internet, but rather using apparently neutral data (it's trained on the [English Wikipedia](#) and [BookCorpus](#) datasets).

When you use these tools, you therefore need to keep in the back of your mind that the original model you are using could very easily generate sexist, racist, or homophobic content. Fine-tuning the model on your data won't make this intrinsic bias disappear.

Summary

In this chapter, you saw how to approach different NLP tasks using the high-level `pipeline()` function from  Transformers. You also saw how to search for and use models in the Hub, as well as how to use the Inference API to test the models directly in your browser.

Using Transformers | Introduction

As you saw in [Chapter 1](#), Transformer models are usually very large. With millions to tens of *billions* of parameters, training and deploying these models is a complicated undertaking. Furthermore, with new models being released on a near-daily basis and each having its own implementation, trying them all out is no easy task.

The 🤖 Transformers library was created to solve this problem. Its goal is to provide a single API through which any Transformer model can be loaded, trained, and saved. The library's main features are:

- **Ease of use:** Downloading, loading, and using a state-of-the-art NLP model for inference can be done in just two lines of code.
- **Flexibility:** At their core, all models are simple PyTorch `nn.Module` or TensorFlow `tf.keras.Model` classes and can be handled like any other models in their respective machine learning (ML) frameworks.
- **Simplicity:** Hardly any abstractions are made across the library. The “All in one file” is a core concept: a model's forward pass is entirely defined in a single file, so that the code itself is understandable and hackable.

This last feature makes 🤖 Transformers quite different from other ML libraries. The models are not built on modules that are shared across files; instead, each model has its own layers. In addition to making the models more approachable and understandable, this allows you to easily experiment on one model without affecting others.

This chapter will begin with an end-to-end example where we use a model and a tokenizer together to replicate the `pipeline()` function introduced in [Chapter 1](#). Next, we'll discuss the model API: we'll dive into the model and configuration classes, and show you how to load a model and how it processes numerical inputs to output predictions.

Then we'll look at the tokenizer API, which is the other main component of the `pipeline()` function. Tokenizers take care of the first and last processing steps, handling the conversion from text to numerical inputs for the neural network, and the conversion back to text when it is needed. Finally, we'll show you how to handle sending multiple sentences through a model in a prepared batch, then wrap it all up with a closer look at the high-level `tokenizer()` function.

⚠ In order to benefit from all features available with the Model Hub and 🤖 Transformers, we recommend [creating an account](#).

Behind the pipeline

This is the first section where the content is slightly different depending on whether you use PyTorch or TensorFlow. Toggle the switch on top of the title to select the platform you prefer!

Let's start with a complete example, taking a look at what happened behind the scenes when we executed the following code in [Chapter 1](#):

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")

classifier(
    [
        "I've been waiting for a HuggingFace course my whole life.",
        "I hate this so much!",
    ]
)
```

and obtained:

```
[{'label': 'POSITIVE', 'score': 0.9598047137260437},
 {'label': 'NEGATIVE', 'score': 0.9994558095932007}]
```

As we saw in [Chapter 1](#), this pipeline groups together three steps: preprocessing, passing the inputs through the model, and postprocessing:

Let's quickly go over each of these.

Preprocessing with a tokenizer

Like other neural networks, Transformer models can't process raw text directly, so the first step of our pipeline is to convert the text inputs into numbers that the model can make sense of. To do this we use a *tokenizer*, which will be responsible for:

- Splitting the input into words, subwords, or symbols (like punctuation) that are called *tokens*
- Mapping each token to an integer
- Adding additional inputs that may be useful to the model

All this preprocessing needs to be done in exactly the same way as when the model was pretrained, so we first need to download that information from the [Model Hub](#). To do this, we use the `AutoTokenizer` class and its `from_pretrained()` method. Using the checkpoint name of our model, it will automatically fetch the data associated with the model's tokenizer and cache it (so it's only downloaded the first time you run the code below).

Since the default checkpoint of the `sentiment-analysis` pipeline is `distilbert-base-uncased-finetuned-sst-2-english` (you can see its model card [here](#)), we run the following:

```
from transformers import AutoTokenizer

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

Once we have the tokenizer, we can directly pass our sentences to it and we'll get back a dictionary that's ready to feed to our model! The only thing left to do is to convert the list of input IDs to tensors.

You can use 🤖 Transformers without having to worry about which ML framework is used as a backend; it might be PyTorch or TensorFlow, or Flax for some models. However, Transformer models only

accept `tensors` as input. If this is your first time hearing about tensors, you can think of them as NumPy arrays instead. A NumPy array can be a scalar (0D), a vector (1D), a matrix (2D), or have more dimensions. It's effectively a tensor; other ML frameworks' tensors behave similarly, and are usually as simple to instantiate as NumPy arrays.

To specify the type of tensors we want to get back (PyTorch, TensorFlow, or plain NumPy), we use the `return_tensors` argument:

```
raw_inputs = [
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!",
]

inputs = tokenizer(raw_inputs, padding=True, truncation=True, return_tensors="pt")

print(inputs)
```

Don't worry about padding and truncation just yet; we'll explain those later. The main things to remember here are that you can pass one sentence or a list of sentences, as well as specifying the type of tensors you want to get back (if no type is passed, you will get a list of lists as a result).

Here's what the results look like as PyTorch tensors:

```
{
  'input_ids': tensor([
    [ 101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607,
    2026, 2878, 2166, 1012, 102],
    [ 101, 1045, 5223, 2023, 2061, 2172, 999, 102, 0, 0,
    0, 0, 0, 0, 0]
  ]),
```

```
'attention_mask': tensor([
  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
  [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
])
}
```

The output itself is a dictionary containing two keys, `input_ids` and `attention_mask`. `input_ids` contains two rows of integers (one for each sentence) that are the unique identifiers of the tokens in each sentence. We'll explain what the `attention_mask` is later in this chapter.

Going through the model

We can download our pretrained model the same way we did with our tokenizer. 🤖 Transformers provides an `AutoModel` class which also has a `from_pretrained()` method:

```
from transformers import AutoModel

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

model = AutoModel.from_pretrained(checkpoint)
```

In this code snippet, we have downloaded the same checkpoint we used in our pipeline before (it should actually have been cached already) and instantiated a model with it.

This architecture contains only the base Transformer module: given some inputs, it outputs what we'll call *hidden states*, also known as *features*. For each model input, we'll retrieve a high-dimensional vector representing the **contextual understanding of that input by the Transformer model**.

If this doesn't make sense, don't worry about it. We'll explain it all later.

While these hidden states can be useful on their own, they're usually inputs to another part of the model, known as the *head*. In [Chapter 1](#), the different tasks could have been performed with the same architecture, but each of these tasks will have a different head associated with it.

A high-dimensional vector?

The vector output by the Transformer module is usually large. It generally has three dimensions:

- **Batch size:** The number of sequences processed at a time (2 in our example).
- **Sequence length:** The length of the numerical representation of the sequence (16 in our example).
- **Hidden size:** The vector dimension of each model input.

It is said to be “high dimensional” because of the last value. The hidden size can be very large (768 is common for smaller models, and in larger models this can reach 3072 or more).

We can see this if we feed the inputs we preprocessed to our model:

```
outputs = model(**inputs)

print(outputs.last_hidden_state.shape)
```

```
torch.Size([2, 16, 768])
```

Note that the outputs of 🤖 Transformers models behave like `namedtuples` or dictionaries. You can access the elements by attributes (like we did) or by key (`outputs["last_hidden_state"]`), or even by index if you know exactly where the thing you are looking for is (`outputs[0]`).

Model heads: Making sense out of numbers

The model heads take the high-dimensional vector of hidden states as input and project them onto a different dimension. They are usually composed of one or a few linear layers:

The output of the Transformer model is sent directly to the model head to be processed.

In this diagram, the model is represented by its embeddings layer and the subsequent layers. The embeddings layer converts each input ID in the tokenized input into a vector that represents the associated token. The subsequent layers manipulate those vectors using the attention mechanism to produce the final representation of the sentences.

There are many different architectures available in 🤖 Transformers, with each one designed around tackling a specific task. Here is a non-exhaustive list:

- `*Model` (retrieve the hidden states)
- `*ForCausalLM`
- `*ForMaskedLM`
- `*ForMultipleChoice`
- `*ForQuestionAnswering`
- `*ForSequenceClassification`
- `*ForTokenClassification`
- and others 🤖

For our example, we will need a model with a sequence classification head (to be able to classify the sentences as positive or negative). So, we won't actually use the `AutoModel` class, but `AutoModelForSequenceClassification`:

```
from transformers import AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

model = AutoModelForSequenceClassification.from_pretrained(checkpoint)

outputs = model(**inputs)
```

Now if we look at the shape of our outputs, the dimensionality will be much lower: the model head takes as input the high-dimensional vectors we saw before, and outputs vectors containing two values (one per label):

```
print(outputs.logits.shape)
```

```
torch.Size([2, 2])
```

Since we have just two sentences and two labels, the result we get from our model is of shape 2 x 2.

Postprocessing the output

The values we get as output from our model don't necessarily make sense by themselves. Let's take a look:

```
print(outputs.logits)
```

```
tensor([[ -1.5607,  1.6123],  
        [ 4.1692, -3.3464]], grad_fn=<AddmmBackward>)
```

Our model predicted `[-1.5607, 1.6123]` for the first sentence and `[4.1692, -3.3464]` for the second one. Those are not probabilities but *logits*, the raw, unnormalized scores outputted by the last layer of the model. To be converted to probabilities, they need to go through a [SoftMax](#) layer (all Transformers models output the logits, as the loss function for training will generally fuse the last activation function, such as SoftMax, with the actual loss function, such as cross entropy):

```
import torch  
  
predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)  
  
print(predictions)
```

```
tensor([[4.0195e-02, 9.5980e-01],  
        [9.9946e-01, 5.4418e-04]], grad_fn=<SoftmaxBackward>)
```

Now we can see that the model predicted [0.0402, 0.9598] for the first sentence and [0.9995, 0.0005] for the second one. These are recognizable probability scores.

To get the labels corresponding to each position, we can inspect the `id2label` attribute of the model config (more on this in the next section):


```
model.config.id2label
```

```
{0: 'NEGATIVE', 1: 'POSITIVE'}
```

Now we can conclude that the model predicted the following:

- First sentence: NEGATIVE: 0.0402, POSITIVE: 0.9598
- Second sentence: NEGATIVE: 0.9995, POSITIVE: 0.0005

We have successfully reproduced the three steps of the pipeline: preprocessing with tokenizers, passing the inputs through the model, and postprocessing! Now let's take some time to dive deeper into each of those steps.

 **Try it out!** Choose two (or more) texts of your own and run them through the `sentiment-analysis` pipeline. Then replicate the steps you saw here yourself and check that you obtain the same results!

Models

In this section we'll take a closer look at creating and using a model. We'll use the `AutoModel` class, which is handy when you want to instantiate any model from a checkpoint.

The `AutoModel` class and all of its relatives are actually simple wrappers over the wide variety of models available in the library. It's a clever wrapper as it can automatically guess the appropriate model architecture for your checkpoint, and then instantiates a model with this architecture.

However, if you know the type of model you want to use, you can use the class that defines its architecture directly. Let's take a look at how this works with a BERT model.

Creating a Transformer

The first thing we'll need to do to initialize a BERT model is load a configuration object:

```
from transformers import BertConfig, BertModel
```

```
# Building the config
```

```
config = BertConfig()
```

```
# Building the model from the config
```

```
model = BertModel(config)
```

The configuration contains many attributes that are used to build the model:

```
print(config)
```

```
BertConfig {  
  
    [...]  
  
    "hidden_size": 768,  
  
    "intermediate_size": 3072,  
  
    "max_position_embeddings": 512,  
  
    "num_attention_heads": 12,  
  
    "num_hidden_layers": 12,  
  
    [...]  
  
}
```

While you haven't seen what all of these attributes do yet, you should recognize some of them: the `hidden_size` attribute defines the size of the `hidden_states` vector, and `num_hidden_layers` defines the number of layers the Transformer model has.

Different loading methods

Creating a model from the default configuration initializes it with random values:

```
from transformers import BertConfig, BertModel  
  
config = BertConfig()  
  
model = BertModel(config)
```

```
# Model is randomly initialized!
```

The model can be used in this state, but it will output gibberish; it needs to be trained first. We could train the model from scratch on the task at hand, but as you saw in [Chapter 1](#), this would require a long time and a lot of data, and it would have a non-negligible environmental impact. To avoid unnecessary and duplicated effort, it's imperative to be able to share and reuse models that have already been trained.

Loading a Transformer model that is already trained is simple — we can do this using the `from_pretrained()` method:

```
from transformers import BertModel

model = BertModel.from_pretrained("bert-base-cased")
```

As you saw earlier, we could replace `BertModel` with the equivalent `AutoModel` class. We'll do this from now on as this produces checkpoint-agnostic code; if your code works for one checkpoint, it should work seamlessly with another. This applies even if the architecture is different, as long as the checkpoint was trained for a similar task (for example, a sentiment analysis task).

In the code sample above we didn't use `BertConfig`, and instead loaded a pretrained model via the `bert-base-cased` identifier. This is a model checkpoint that was trained by the authors of BERT themselves; you can find more details about it in its [model card](#).

This model is now initialized with all the weights of the checkpoint. It can be used directly for inference on the tasks it was trained on, and it can also be fine-tuned on a new task. By training with pretrained weights rather than from scratch, we can quickly achieve good results.

The weights have been downloaded and cached (so future calls to the `from_pretrained()` method won't re-download them) in the cache folder, which defaults to `~/.cache/huggingface/transformers`. You can customize your cache folder by setting the `HF_HOME` environment variable.

The identifier used to load the model can be the identifier of any model on the Model Hub, as long as it is compatible with the BERT architecture. The entire list of available BERT checkpoints can be found [here](#).

Saving methods

Saving a model is as easy as loading one — we use the `save_pretrained()` method, which is analogous to the `from_pretrained()` method:

```
model.save_pretrained("directory_on_my_computer")
```

This saves two files to your disk:

```
ls directory_on_my_computer  
  
config.json pytorch_model.bin
```

If you take a look at the `config.json` file, you'll recognize the attributes necessary to build the model architecture. This file also contains some metadata, such as where the checkpoint originated and what 🤖 Transformers version you were using when you last saved the checkpoint.

The `pytorch_model.bin` file is known as the *state dictionary*; it contains all your model's weights. The two files go hand in hand; the configuration is necessary to know your model's architecture, while the model weights are your model's parameters.

Using a Transformer model for inference

Now that you know how to load and save a model, let's try using it to make some predictions. Transformer models can only process numbers — numbers that the tokenizer generates. But before we discuss tokenizers, let's explore what inputs the model accepts.

Tokenizers can take care of casting the inputs to the appropriate framework's tensors, but to help you understand what's going on, we'll take a quick look at what must be done before sending the inputs to the model.

Let's say we have a couple of sequences:

```
sequences = ["Hello!", "Cool.", "Nice!"]
```

The tokenizer converts these to vocabulary indices which are typically called *input IDs*. Each sequence is now a list of numbers! The resulting output is:

```
encoded_sequences = [  
    [101, 7592, 999, 102],  
    [101, 4658, 1012, 102],  
    [101, 3835, 999, 102],  
]
```

This is a list of encoded sequences: a list of lists. Tensors only accept rectangular shapes (think matrices). This “array” is already of rectangular shape, so converting it to a tensor is easy:

```
import torch  
  
model_inputs = torch.tensor(encoded_sequences)
```

Using the tensors as inputs to the model

Making use of the tensors with the model is extremely simple — we just call the model with the inputs:

```
output = model(model_inputs)
```


While the model accepts a lot of different arguments, only the input IDs are necessary. We'll explain what the other arguments do and when they are required later, but first we need to take a closer look at the tokenizers that build the inputs that a Transformer model can understand.

Tokenizers

Tokenizers are one of the core components of the NLP pipeline. They serve one purpose: to translate text into data that can be processed by the model. Models can only process numbers, so tokenizers need to convert our text inputs to numerical data. In this section, we'll explore exactly what happens in the tokenization pipeline.

In NLP tasks, the data that is generally processed is raw text. Here's an example of such text:

```
Jim Henson was a puppeteer
```

However, models can only process numbers, so we need to find a way to convert the raw text to numbers. That's what the tokenizers do, and there are a lot of ways to go about this. The goal is to find the most meaningful representation — that is, the one that makes the most sense to the model — and, if possible, the smallest representation.

Let's take a look at some examples of tokenization algorithms, and try to answer some of the questions you may have about tokenization.

Word-based

The first type of tokenizer that comes to mind is *word-based*. It's generally very easy to set up and use with only a few rules, and it often yields decent results. For example, in the image below, the goal is to split the raw text into words and find a numerical representation for each of them:

There are different ways to split the text. For example, we could use whitespace to tokenize the text into words by applying Python's `split()` function:

```
tokenized_text = "Jim Henson was a puppeteer".split()

print(tokenized_text)
```

```
['Jim', 'Henson', 'was', 'a', 'puppeteer']
```

There are also variations of word tokenizers that have extra rules for punctuation. With this kind of tokenizer, we can end up with some pretty large “vocabularies,” where a vocabulary is defined by the total number of independent tokens that we have in our corpus.

Each word gets assigned an ID, starting from 0 and going up to the size of the vocabulary. The model uses these IDs to identify each word.

If we want to completely cover a language with a word-based tokenizer, we’ll need to have an identifier for each word in the language, which will generate a huge amount of tokens. For example, there are over 500,000 words in the English language, so to build a map from each word to an input ID we’d need to keep track of that many IDs. Furthermore, words like “dog” are represented differently from words like “dogs”, and the model will initially have no way of knowing that “dog” and “dogs” are similar: it will identify the two words as unrelated. The same applies to other similar words, like “run” and “running”, which the model will not see as being similar initially.

Finally, we need a custom token to represent words that are not in our vocabulary. This is known as the “unknown” token, often represented as “[UNK]” or “”. It’s generally a bad sign if you see that the tokenizer is producing a lot of these tokens, as it wasn’t able to retrieve a sensible representation of a word and you’re losing information along the way. The goal when crafting the vocabulary is to do it in such a way that the tokenizer tokenizes as few words as possible into the unknown token.

One way to reduce the amount of unknown tokens is to go one level deeper, using a *character-based* tokenizer.

Character-based

Character-based tokenizers split the text into characters, rather than words. This has two primary benefits:

- The vocabulary is much smaller.
- There are much fewer out-of-vocabulary (unknown) tokens, since every word can be built from characters.

But here too some questions arise concerning spaces and punctuation:

This approach isn't perfect either. Since the representation is now based on characters rather than words, one could argue that, intuitively, it's less meaningful: each character doesn't mean a lot on its own, whereas that is the case with words. However, this again differs according to the language; in Chinese, for example, each character carries more information than a character in a Latin language.

Another thing to consider is that we'll end up with a very large amount of tokens to be processed by our model: whereas a word would only be a single token with a word-based tokenizer, it can easily turn into 10 or more tokens when converted into characters.

To get the best of both worlds, we can use a third technique that combines the two approaches: *subword tokenization*.

Subword tokenization

Subword tokenization algorithms rely on the principle that frequently used words should not be split into smaller subwords, but rare words should be decomposed into meaningful subwords.

For instance, "annoyingly" might be considered a rare word and could be decomposed into "annoying" and "ly". These are both likely to appear more frequently as standalone subwords, while at the same time the meaning of "annoyingly" is kept by the composite meaning of "annoying" and "ly".

Here is an example showing how a subword tokenization algorithm would tokenize the sequence "Let's do tokenization!":

These subwords end up providing a lot of semantic meaning: for instance, in the example above "tokenization" was split into "token" and "ization", two tokens that have a semantic meaning while being space-efficient (only two tokens are needed to represent a long word). This allows us to have relatively good coverage with small vocabularies, and close to no unknown tokens.

This approach is especially useful in agglutinative languages such as Turkish, where you can form (almost) arbitrarily long complex words by stringing together subwords.

And more!

Unsurprisingly, there are many more techniques out there. To name a few:

- Byte-level BPE, as used in GPT-2
- WordPiece, as used in BERT
- SentencePiece or Unigram, as used in several multilingual models

You should now have sufficient knowledge of how tokenizers work to get started with the API.

Loading and saving

Loading and saving tokenizers is as simple as it is with models. Actually, it's based on the same two methods: `from_pretrained()` and `save_pretrained()`. These methods will load or save the algorithm used by the tokenizer (a bit like the *architecture* of the model) as well as its vocabulary (a bit like the *weights* of the model).

Loading the BERT tokenizer trained with the same checkpoint as BERT is done the same way as loading the model, except we use the `BertTokenizer` class:

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-cased")
```

Similar to `AutoModel`, the `AutoTokenizer` class will grab the proper tokenizer class in the library based on the checkpoint name, and can be used directly with any checkpoint:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

We can now use the tokenizer as shown in the previous section:

```
tokenizer("Using a Transformer network is simple")
```

```
{'input_ids': [101, 7993, 170, 11303, 1200, 2443, 1110, 3014, 102],
```

```
'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

Saving a tokenizer is identical to saving a model:

```
tokenizer.save_pretrained("directory_on_my_computer")
```

We'll talk more about `token_type_ids` in [Chapter 3](#), and we'll explain the `attention_mask` key a little later. First, let's see how the `input_ids` are generated. To do this, we'll need to look at the intermediate methods of the tokenizer.

Encoding

Translating text to numbers is known as *encoding*. Encoding is done in a two-step process: the tokenization, followed by the conversion to input IDs.

As we've seen, the first step is to split the text into words (or parts of words, punctuation symbols, etc.), usually called *tokens*. There are multiple rules that can govern that process, which is why we need to instantiate the tokenizer using the name of the model, to make sure we use the same rules that were used when the model was pretrained.

The second step is to convert those tokens into numbers, so we can build a tensor out of them and feed them to the model. To do this, the tokenizer has a *vocabulary*, which is the part we download when we instantiate it with the `from_pretrained()` method. Again, we need to use the same vocabulary used when the model was pretrained.

To get a better understanding of the two steps, we'll explore them separately. Note that we will use some methods that perform parts of the tokenization pipeline separately to show you the intermediate results of those steps, but in practice, you should call the tokenizer directly on your inputs (as shown in the section 2).

Tokenization

The tokenization process is done by the `tokenize()` method of the tokenizer:

```
from transformers import AutoTokenizer
```

```
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

```
sequence = "Using a Transformer network is simple"
```

```
tokens = tokenizer.tokenize(sequence)
```

```
print(tokens)
```

The output of this method is a list of strings, or tokens:

```
['Using', 'a', 'transform', '##er', 'network', 'is', 'simple']
```

This tokenizer is a subword tokenizer: it splits the words until it obtains tokens that can be represented by its vocabulary. That's the case here with `transformer`, which is split into two tokens: `transform` and `##er`.

From tokens to input IDs


The conversion to input IDs is handled by the `convert_tokens_to_ids()` tokenizer method:

```
ids = tokenizer.convert_tokens_to_ids(tokens)
```

```
print(ids)
```

```
[7993, 170, 11303, 1200, 2443, 1110, 3014]
```

These outputs, once converted to the appropriate framework tensor, can then be used as inputs to a model as seen earlier in this chapter.

 **Try it out!** Replicate the two last steps (tokenization and conversion to input IDs) on the input sentences we used in section 2 (“I’ve been waiting for a HuggingFace course my whole life.” and “I hate this so much!”). Check that you get the same input IDs we got earlier!

Decoding

Decoding is going the other way around: from vocabulary indices, we want to get a string. This can be done with the `decode()` method as follows:

```
decoded_string = tokenizer.decode([7993, 170, 11303, 1200, 2443, 1110, 3014])  
  
print(decoded_string)
```

```
'Using a Transformer network is simple'
```

Note that the `decode` method not only converts the indices back to tokens, but also groups together the tokens that were part of the same words to produce a readable sentence. This behavior will be extremely useful when we use models that predict new text (either text generated from a prompt, or for sequence-to-sequence problems like translation or summarization).

By now you should understand the atomic operations a tokenizer can handle: tokenization, conversion to IDs, and converting IDs back to a string. However, we’ve just scraped the tip of the iceberg. In the following section, we’ll take our approach to its limits and take a look at how to overcome them.

Handling multiple sequences

In the previous section, we explored the simplest of use cases: doing inference on a single sequence of a small length. However, some questions emerge already:

- How do we handle multiple sequences?
- How do we handle multiple sequences *of different lengths*?
- Are vocabulary indices the only inputs that allow a model to work well?
- Is there such a thing as too long a sequence?

Let's see what kinds of problems these questions pose, and how we can solve them using the 🤖 Transformers API.

Models expect a batch of inputs

In the previous exercise you saw how sequences get translated into lists of numbers. Let's convert this list of numbers to a tensor and send it to the model:

```
import torch

from transformers import AutoTokenizer, AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

tokenizer = AutoTokenizer.from_pretrained(checkpoint)

model = AutoModelForSequenceClassification.from_pretrained(checkpoint)

sequence = "I've been waiting for a HuggingFace course my whole life."
```

```
tokens = tokenizer.tokenize(sequence)

ids = tokenizer.convert_tokens_to_ids(tokens)

input_ids = torch.tensor(ids)

# This line will fail.

model(input_ids)
```

IndexError: Dimension out of range (expected to be in range of [-1, 0], but got 1)

Oh no! Why did this fail? “We followed the steps from the pipeline in section 2.

The problem is that we sent a single sequence to the model, whereas 🤖 Transformers models expect multiple sentences by default. Here we tried to do everything the tokenizer did behind the scenes when we applied it to a `sequence`. But if you look closely, you’ll see that the tokenizer didn’t just convert the list of input IDs into a tensor, it added a dimension on top of it:

```
tokenized_inputs = tokenizer(sequence, return_tensors="pt")

print(tokenized_inputs["input_ids"])
```

```
tensor([[ 101,  1045,  1005,  2310,  2042,  3403,  2005,  1037, 17662, 12172,
         2607,  2026,  2878,  2166,  1012,  102]])
```

Let’s try again and add a new dimension:

```
import torch

from transformers import AutoTokenizer, AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

tokenizer = AutoTokenizer.from_pretrained(checkpoint)

model = AutoModelForSequenceClassification.from_pretrained(checkpoint)

sequence = "I've been waiting for a HuggingFace course my whole life."

tokens = tokenizer.tokenize(sequence)

ids = tokenizer.convert_tokens_to_ids(tokens)

input_ids = torch.tensor([ids])

print("Input IDs:", input_ids)

output = model(input_ids)

print("Logits:", output.logits)
```

We print the input IDs as well as the resulting logits — here's the output:


```
Input IDs: [[ 1045,  1005,  2310,  2042,  3403,  2005,  1037, 17662, 12172,  2607,
 2026,  2878,  2166,  1012]]
```

```
Logits: [[-2.7276, 2.8789]]
```

Batching is the act of sending multiple sentences through the model, all at once. If you only have one sentence, you can just build a batch with a single sequence:

```
batched_ids = [ids, ids]
```

This is a batch of two identical sequences!

 **Try it out!** Convert this `batched_ids` list into a tensor and pass it through your model. Check that you obtain the same logits as before (but twice)!

Batching allows the model to work when you feed it multiple sentences. Using multiple sequences is just as simple as building a batch with a single sequence. There's a second issue, though. When you're trying to batch together two (or more) sentences, they might be of different lengths. If you've ever worked with tensors before, you know that they need to be of rectangular shape, so you won't be able to convert the list of input IDs into a tensor directly. To work around this problem, we usually *pad* the inputs.

Padding the inputs

The following list of lists cannot be converted to a tensor:

```
batched_ids = [  
    [200, 200, 200],  
    [200, 200]  
]
```

In order to work around this, we'll use *padding* to make our tensors have a rectangular shape. Padding makes sure all our sentences have the same length by adding a special word called the *padding token* to the sentences with fewer values. For example, if you have 10 sentences with 10 words and 1 sentence with 20 words, padding will ensure all the sentences have 20 words. In our example, the resulting tensor looks like this:

```
padding_id = 100

batched_ids = [

    [200, 200, 200],

    [200, 200, padding_id],

]
```

The padding token ID can be found in `tokenizer.pad_token_id`. Let's use it and send our two sentences through the model individually and batched together:

```
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)

sequence1_ids = [[200, 200, 200]]

sequence2_ids = [[200, 200]]

batched_ids = [

    [200, 200, 200],

    [200, 200, tokenizer.pad_token_id],

]

print(model(torch.tensor(sequence1_ids)).logits)

print(model(torch.tensor(sequence2_ids)).logits)
```

```
print(model(torch.tensor(batched_ids)).logits)
```

```
tensor([[ 1.5694, -1.3895]], grad_fn=<AddmmBackward>)
```

```
tensor([[ 0.5803, -0.4125]], grad_fn=<AddmmBackward>)
```

```
tensor([[ 1.5694, -1.3895],  
        [ 1.3373, -1.2163]], grad_fn=<AddmmBackward>)
```

There's something wrong with the logits in our batched predictions: the second row should be the same as the logits for the second sentence, but we've got completely different values!

This is because the key feature of Transformer models is attention layers that *contextualize* each token. These will take into account the padding tokens since they attend to all of the tokens of a sequence. To get the same result when passing individual sentences of different lengths through the model or when passing a batch with the same sentences and padding applied, we need to tell those attention layers to ignore the padding tokens. This is done by using an attention mask.

Attention masks

Attention masks are tensors with the exact same shape as the input IDs tensor, filled with 0s and 1s: 1s indicate the corresponding tokens should be attended to, and 0s indicate the corresponding tokens should not be attended to (i.e., they should be ignored by the attention layers of the model).

Let's complete the previous example with an attention mask:

```
batched_ids = [  
    [200, 200, 200],  
    [200, 200, tokenizer.pad_token_id],  
]
```

```
attention_mask = [
    [1, 1, 1],
    [1, 1, 0],
]


outputs = model(torch.tensor(batched_ids),
attention_mask=torch.tensor(attention_mask))

print(outputs.logits)
```

```
tensor([[ 1.5694, -1.3895],
        [ 0.5803, -0.4125]], grad_fn=<AddmmBackward>)
```

Now we get the same logits for the second sentence in the batch.

Notice how the last value of the second sequence is a padding ID, which is a 0 value in the attention mask.

 **Try it out!** Apply the tokenization manually on the two sentences used in section 2 (“I’ve been waiting for a HuggingFace course my whole life.” and “I hate this so much!”). Pass them through the model and check that you get the same logits as in section 2. Now batch them together using the padding token, then create the proper attention mask. Check that you obtain the same results when going through the model!

Longer sequences

With Transformer models, there is a limit to the lengths of the sequences we can pass the models. Most models handle sequences of up to 512 or 1024 tokens, and will crash when asked to process longer sequences. There are two solutions to this problem:

- Use a model with a longer supported sequence length.
- Truncate your sequences.

Models have different supported sequence lengths, and some specialize in handling very long sequences. [Longformer](#) is one example, and another is [LED](#). If you're working on a task that requires very long sequences, we recommend you take a look at those models.

Otherwise, we recommend you truncate your sequences by specifying the `max_sequence_length` parameter:

```
sequence = sequence[:max_sequence_length]
```


Putting it all together

In the last few sections, we've been trying our best to do most of the work by hand. We've explored how tokenizers work and looked at tokenization, conversion to input IDs, padding, truncation, and attention masks.

However, as we saw in section 2, the 🤖 Transformers API can handle all of this for us with a high-level function that we'll dive into here. When you call your `tokenizer` directly on the sentence, you get back inputs that are ready to pass through your model:

```
from transformers import AutoTokenizer

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

tokenizer = AutoTokenizer.from_pretrained(checkpoint)

sequence = "I've been waiting for a HuggingFace course my whole life."

model_inputs = tokenizer(sequence)
```

Here, the `model_inputs` variable contains everything that's necessary for a model to operate well. For DistilBERT, that includes the input IDs as well as the attention mask. Other models that accept additional inputs will also have those output by the `tokenizer` object.

As we'll see in some examples below, this method is very powerful. First, it can tokenize a single sequence:

```
sequence = "I've been waiting for a HuggingFace course my whole life."
```

```
model_inputs = tokenizer(sequence)
```

It also handles multiple sequences at a time, with no change in the API:

```
sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]
```

```
model_inputs = tokenizer(sequences)
```

It can pad according to several objectives:

```
# Will pad the sequences up to the maximum sequence length
```

```
model_inputs = tokenizer(sequences, padding="longest")
```

```
# Will pad the sequences up to the model max length
```

```
# (512 for BERT or DistilBERT)
```

```
model_inputs = tokenizer(sequences, padding="max_length")
```

```
# Will pad the sequences up to the specified max length
```

```
model_inputs = tokenizer(sequences, padding="max_length", max_length=8)
```

It can also truncate sequences:

```
sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]
```

```
# Will truncate the sequences that are longer than the model max length
```

```
# (512 for BERT or DistilBERT)
```

```
model_inputs = tokenizer(sequences, truncation=True)
```

```
# Will truncate the sequences that are longer than the specified max length
```

```
model_inputs = tokenizer(sequences, max_length=8, truncation=True)
```

The `tokenizer` object can handle the conversion to specific framework tensors, which can then be directly sent to the model. For example, in the following code sample we are prompting the tokenizer to return tensors from the different frameworks — `"pt"` returns PyTorch tensors, `"tf"` returns TensorFlow tensors, and `"np"` returns NumPy arrays:

```
sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]
```

```
# Returns PyTorch tensors
```

```
model_inputs = tokenizer(sequences, padding=True, return_tensors="pt")
```

```
# Returns TensorFlow tensors
```

```
model_inputs = tokenizer(sequences, padding=True, return_tensors="tf")
```

```
# Returns NumPy arrays
```

```
model_inputs = tokenizer(sequences, padding=True, return_tensors="np")
```

Special tokens

If we take a look at the input IDs returned by the tokenizer, we will see they are a tiny bit different from what we had earlier:

```
sequence = "I've been waiting for a HuggingFace course my whole life."
```

```
model_inputs = tokenizer(sequence)
```

```
print(model_inputs["input_ids"])
```

```
tokens = tokenizer.tokenize(sequence)
```

```
ids = tokenizer.convert_tokens_to_ids(tokens)
```

```
print(ids)
```

```
[101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607, 2026, 2878,  
2166, 1012, 102]
```

```
[1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607, 2026, 2878, 2166,  
1012]
```

One token ID was added at the beginning, and one at the end. Let's decode the two sequences of IDs above to see what this is about:

```
print(tokenizer.decode(model_inputs["input_ids"]))
```

```
print(tokenizer.decode(ids))
```

```
"[CLS] i've been waiting for a huggingface course my whole life. [SEP]"
```

```
"i've been waiting for a huggingface course my whole life."
```

The tokenizer added the special word [CLS] at the beginning and the special word [SEP] at the end. This is because the model was pretrained with those, so to get the same results for inference we need to add them as well. Note that some models don't add special words, or add different ones; models may also add these special words only at the beginning, or only at the end. In any case, the tokenizer knows which ones are expected and will deal with this for you.

Wrapping up: From tokenizer to model

Now that we've seen all the individual steps the `tokenizer` object uses when applied on texts, let's see one final time how it can handle multiple sequences (padding!), very long sequences (truncation!), and multiple types of tensors with its main API:

```
import torch
```

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification
```

```
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
```

```
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

```
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
```

```
sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]
```

```
tokens = tokenizer(sequences, padding=True, truncation=True, return_tensors="pt")
```

```
output = model(**tokens)
```

Basic usage completed!



Great job following the course up to here! To recap, in this chapter you:

- Learned the basic building blocks of a Transformer model.
- Learned what makes up a tokenization pipeline.
- Saw how to use a Transformer model in practice.
- Learned how to leverage a tokenizer to convert text to tensors that are understandable by the model.
- Set up a tokenizer and a model together to get from text to predictions.
- Learned the limitations of input IDs, and learned about attention masks.
- Played around with versatile and configurable tokenizer methods.

From now on, you should be able to freely navigate the 📖 Transformers docs: the vocabulary will sound familiar, and you've already seen the methods that you'll use the majority of the time.

Fine-Tuning a Pretrained Model | Introduction

In [Chapter 2](#) we explored how to use tokenizers and pretrained models to make predictions. But what if you want to fine-tune a pretrained model for your own dataset? That's the topic of this chapter! You will learn:

- How to prepare a large dataset from the Hub
- How to use the high-level `Trainer` API to fine-tune a model
- How to use a custom training loop
- How to leverage the 🦘 Accelerate library to easily run that custom training loop on any distributed setup

In order to upload your trained checkpoints to the Hugging Face Hub, you will need a huggingface.co account: [create an account](#)

Processing the data

Continuing with the example from the [previous chapter](#), here is how we would train a sequence classifier on one batch in PyTorch:

```
import torch

from transformers import AdamW, AutoTokenizer, AutoModelForSequenceClassification

# Same as before

checkpoint = "bert-base-uncased"

tokenizer = AutoTokenizer.from_pretrained(checkpoint)

model = AutoModelForSequenceClassification.from_pretrained(checkpoint)

sequences = [

    "I've been waiting for a HuggingFace course my whole life.",

    "This course is amazing!",

]

batch = tokenizer(sequences, padding=True, truncation=True, return_tensors="pt")

# This is new

batch["labels"] = torch.tensor([1, 1])
```

```
optimizer = AdamW(model.parameters())

loss = model(**batch).loss

loss.backward()

optimizer.step()
```

Of course, just training the model on two sentences is not going to yield very good results. To get better results, you will need to prepare a bigger dataset.

In this section we will use as an example the MRPC (Microsoft Research Paraphrase Corpus) dataset, introduced in a [paper](#) by William B. Dolan and Chris Brockett. The dataset consists of 5,801 pairs of sentences, with a label indicating if they are paraphrases or not (i.e., if both sentences mean the same thing). We've selected it for this chapter because it's a small dataset, so it's easy to experiment with training on it.

Loading a dataset from the Hub

The Hub doesn't just contain models; it also has multiple datasets in lots of different languages. You can browse the datasets [here](#), and we recommend you try to load and process a new dataset once you have gone through this section (see the general documentation [here](#)). But for now, let's focus on the MRPC dataset! This is one of the 10 datasets composing the [GLUE benchmark](#), which is an academic benchmark that is used to measure the performance of ML models across 10 different text classification tasks.

The 🤗 Datasets library provides a very simple command to download and cache a dataset on the Hub. We can download the MRPC dataset like this:

```
from datasets import load_dataset

raw_datasets = load_dataset("glue", "mrpc")

raw_datasets
```

```
DatasetDict({
  train: Dataset({
    features: ['sentence1', 'sentence2', 'label', 'idx'],
    num_rows: 3668
  })
  validation: Dataset({
    features: ['sentence1', 'sentence2', 'label', 'idx'],
    num_rows: 408
  })
  test: Dataset({
    features: ['sentence1', 'sentence2', 'label', 'idx'],
    num_rows: 1725
  })
})
```

As you can see, we get a `DatasetDict` object which contains the training set, the validation set, and the test set. Each of those contains several columns (`sentence1`, `sentence2`, `label`, and `idx`) and a variable number of rows, which are the number of elements in each set (so, there are 3,668 pairs of sentences in the training set, 408 in the validation set, and 1,725 in the test set).

This command downloads and caches the dataset, by default in `~/.cache/huggingface/datasets`. Recall from Chapter 2 that you can customize your cache folder by setting the `HF_HOME` environment variable.

We can access each pair of sentences in our `raw_datasets` object by indexing, like with a dictionary:

```
raw_train_dataset = raw_datasets["train"]

raw_train_dataset[0]
```

```
{'idx': 0,

 'label': 1,

 'sentence1': 'Amrozi accused his brother , whom he called " the witness " , of
deliberately distorting his evidence .',

 'sentence2': 'Referring to him as only " the witness " , Amrozi accused his brother
of deliberately distorting his evidence .'} }
```

We can see the labels are already integers, so we won't have to do any preprocessing there. To know which integer corresponds to which label, we can inspect the `features` of our `raw_train_dataset`. This will tell us the type of each column:

```
raw_train_dataset.features
```


```
{'sentence1': Value(dtype='string', id=None),

 'sentence2': Value(dtype='string', id=None),

 'label': ClassLabel(num_classes=2, names=['not_equivalent', 'equivalent'],
names_file=None, id=None),

 'idx': Value(dtype='int32', id=None)}
```

Behind the scenes, `label` is of type `ClassLabel`, and the mapping of integers to label name is stored in the `names` folder. `0` corresponds to `not_equivalent`, and `1` corresponds to `equivalent`.

 **Try it out!** Look at element 15 of the training set and element 87 of the validation set. What are their labels?

Preprocessing a dataset

To preprocess the dataset, we need to convert the text to numbers the model can make sense of. As you saw in the [previous chapter](#), this is done with a tokenizer. We can feed the tokenizer one sentence or a list of sentences, so we can directly tokenize all the first sentences and all the second sentences of each pair like this:

```
from transformers import AutoTokenizer

checkpoint = "bert-base-uncased"

tokenizer = AutoTokenizer.from_pretrained(checkpoint)

tokenized_sentences_1 = tokenizer(raw_datasets["train"]["sentence1"])

tokenized_sentences_2 = tokenizer(raw_datasets["train"]["sentence2"])
```

However, we can't just pass two sequences to the model and get a prediction of whether the two sentences are paraphrases or not. We need to handle the two sequences as a pair, and apply the appropriate preprocessing. Fortunately, the tokenizer can also take a pair of sequences and prepare it the way our BERT model expects:

```
inputs = tokenizer("This is the first sentence.", "This is the second one.")

inputs
```

```
{
```

```

: [101, 2023, 2003, 1996, 2034, 6251, 1012, 102, 2023, 2003, 1996,
2117, 2028, 1012, 102],


'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],

'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

}

```

We discussed the `input_ids` and `attention_mask` keys in [Chapter 2](#), but we put off talking about `token_type_ids`. In this example, this is what tells the model which part of the input is the first sentence and which is the second sentence.

 **Try it out!** Take element 15 of the training set and tokenize the two sentences separately and as a pair. What's the difference between the two results?

If we decode the IDs inside `input_ids` back to words:

```
tokenizer.convert_ids_to_tokens(inputs["input_ids"])
```

we will get:

```
['[CLS]', 'this', 'is', 'the', 'first', 'sentence', '.', '[SEP]', 'this', 'is',
'the', 'second', 'one', '.', '[SEP]']
```

So we see the model expects the inputs to be of the form `[CLS] sentence1 [SEP] sentence2 [SEP]` when there are two sentences. Aligning this with the `token_type_ids` gives us:

```

['[CLS]', 'this', 'is', 'the', 'first', 'sentence', '.', '[SEP]', 'this', 'is',
'the', 'second', 'one', '.', '[SEP]']

[ 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
1, 1, 1, 1, 1]

```

As you can see, the parts of the input corresponding to `[CLS] sentence1 [SEP]` all have a token type ID of `0`, while the other parts, corresponding to `sentence2 [SEP]`, all have a token type ID of `1`.

Note that if you select a different checkpoint, you won't necessarily have the `token_type_ids` in your tokenized inputs (for instance, they're not returned if you use a DistilBERT model). They are only returned when the model will know what to do with them, because it has seen them during its pretraining.

Here, BERT is pretrained with token type IDs, and on top of the masked language modeling objective we talked about in [Chapter 1](#), it has an additional objective called *next sentence prediction*. The goal with this task is to model the relationship between pairs of sentences.

With next sentence prediction, the model is provided pairs of sentences (with randomly masked tokens) and asked to predict whether the second sentence follows the first. To make the task non-trivial, half of the time the sentences follow each other in the original document they were extracted from, and the other half of the time the two sentences come from two different documents.

In general, you don't need to worry about whether or not there are `token_type_ids` in your tokenized inputs: as long as you use the same checkpoint for the tokenizer and the model, everything will be fine as the tokenizer knows what to provide to its model.

Now that we have seen how our tokenizer can deal with one pair of sentences, we can use it to tokenize our whole dataset: like in the [previous chapter](#), we can feed the tokenizer a list of pairs of sentences by giving it the list of first sentences, then the list of second sentences. This is also compatible with the padding and truncation options we saw in [Chapter 2](#). So, one way to preprocess the training dataset is:


```
tokenized_dataset = tokenizer(  
  
    raw_datasets["train"]["sentence1"],  
  
    raw_datasets["train"]["sentence2"],  
  
    padding=True,  
  
    truncation=True,  
  
)
```

This works well, but it has the disadvantage of returning a dictionary (with our keys, `input_ids`, `attention_mask`, and `token_type_ids`, and values that are lists of lists). It will also only work if you have enough RAM to store your whole dataset during the tokenization (whereas the

datasets from the  Datasets library are [Apache Arrow](#) files stored on the disk, so you only keep the samples you ask for loaded in memory).

To keep the data as a dataset, we will use the `Dataset.map()` method. This also allows us some extra flexibility, if we need more preprocessing done than just tokenization. The `map()` method works by applying a function on each element of the dataset, so let's define a function that tokenizes our inputs:


```
def tokenize_function(example):  
  
    return tokenizer(example["sentence1"], example["sentence2"], truncation=True)
```

This function takes a dictionary (like the items of our dataset) and returns a new dictionary with the keys `input_ids`, `attention_mask`, and `token_type_ids`. Note that it also works if the `example` dictionary contains several samples (each key as a list of sentences) since the `tokenizer` works on lists of pairs of sentences, as seen before. This will allow us to use the option `batched=True` in our call to `map()`, which will greatly speed up the tokenization. The `tokenizer` is backed by a tokenizer written in Rust from the  [Tokenizers](#) library. This tokenizer can be very fast, but only if we give it lots of inputs at once.

Note that we've left the `padding` argument out in our tokenization function for now. This is because padding all the samples to the maximum length is not efficient: it's better to pad the samples when we're building a batch, as then we only need to pad to the maximum length in that batch, and not the maximum length in the entire dataset. This can save a lot of time and processing power when the inputs have very variable lengths!

Here is how we apply the tokenization function on all our datasets at once. We're using `batched=True` in our call to `map` so the function is applied to multiple elements of our dataset at once, and not on each element separately. This allows for faster preprocessing.

```
tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)  
  
tokenized_datasets
```

The way the  Datasets library applies this preprocessing is by adding new fields to the datasets, one for each key in the dictionary returned by the preprocessing function:


```

DatasetDict({
  train: Dataset({
    features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1',
'sentence2', 'token_type_ids'],
    num_rows: 3668
  })
  validation: Dataset({
    features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1',
'sentence2', 'token_type_ids'],
    num_rows: 408
  })
  test: Dataset({
    features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1',
'sentence2', 'token_type_ids'],
    num_rows: 1725
  })
})

```

You can even use multiprocessing when applying your preprocessing function with `map()` by passing along a `num_proc` argument. We didn't do this here because the 🤖 Tokenizers library already uses multiple threads to tokenize our samples faster, but if you are not using a fast tokenizer backed by this library, this could speed up your preprocessing.

Our `tokenize_function` returns a dictionary with the keys `input_ids`, `attention_mask`, and `token_type_ids`, so those three fields are added to all splits of our dataset. Note that we could also have changed existing fields if our preprocessing function returned a new value for an existing key in the dataset to which we applied `map()`.

The last thing we will need to do is pad all the examples to the length of the longest element when we batch elements together — a technique we refer to as *dynamic padding*.

Dynamic padding

The function that is responsible for putting together samples inside a batch is called a *collate function*. It's an argument you can pass when you build a `DataLoader`, the default being a function that will just convert your samples to PyTorch tensors and concatenate them (recursively if your elements are lists, tuples, or dictionaries). This won't be possible in our case since the inputs we have won't all be of the same size. We have deliberately postponed the padding, to only apply it as necessary on each batch and avoid having over-long inputs with a lot of padding. This will speed up training by quite a bit, but note that if you're training on a TPU it can cause problems — TPUs prefer fixed shapes, even when that requires extra padding.

To do this in practice, we have to define a collate function that will apply the correct amount of padding to the items of the dataset we want to batch together. Fortunately, the 🤖 Transformers library provides us with such a function via `DataCollatorWithPadding`. It takes a tokenizer when you instantiate it (to know which padding token to use, and whether the model expects padding to be on the left or on the right of the inputs) and will do everything you need:

```
from transformers import DataCollatorWithPadding

data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

To test this new toy, let's grab a few samples from our training set that we would like to batch together. Here, we remove the columns `idx`, `sentence1`, and `sentence2` as they won't be needed and contain strings (and we can't create tensors with strings) and have a look at the lengths of each entry in the batch:

```
samples = tokenized_datasets["train"][:8]

samples = {k: v for k, v in samples.items() if k not in ["idx", "sentence1",
"sentence2"]}

[ len(x) for x in samples["input_ids"] ]
```

```
[50, 59, 47, 67, 59, 50, 62, 32]
```


No surprise, we get samples of varying length, from 32 to 67. Dynamic padding means the samples in this batch should all be padded to a length of 67, the maximum length inside the batch. Without dynamic padding, all of the samples would have to be padded to the maximum length in the whole dataset, or the maximum length the model can accept. Let's double-check that our `data_collator` is dynamically padding the batch properly:

```
batch = data_collator(samples)

{k: v.shape for k, v in batch.items()}
```

```
{'attention_mask': torch.Size([8, 67]),
 'input_ids': torch.Size([8, 67]),
 'token_type_ids': torch.Size([8, 67]),
 'labels': torch.Size([8])}
```

Looking good! Now that we've gone from raw text to batches our model can deal with, we're ready to fine-tune it!

 **Try it out!** Replicate the preprocessing on the GLUE SST-2 dataset. It's a little bit different since it's composed of single sentences instead of pairs, but the rest of what we did should look the same. For a harder challenge, try to write a preprocessing function that works on any of the GLUE tasks.

Fine-tuning a model with the Trainer API

🤖 Transformers provides a `Trainer` class to help you fine-tune any of the pretrained models it provides on your dataset. Once you've done all the data preprocessing work in the last section, you have just a few steps left to define the `Trainer`. The hardest part is likely to be preparing the environment to run `Trainer.train()`, as it will run very slowly on a CPU. If you don't have a GPU set up, you can get access to free GPUs or TPUs on [Google Colab](#).

The code examples below assume you have already executed the examples in the previous section. Here is a short summary recapping what you need:

```
from datasets import load_dataset

from transformers import AutoTokenizer, DataCollatorWithPadding

raw_datasets = load_dataset("glue", "mrpc")

checkpoint = "bert-base-uncased"

tokenizer = AutoTokenizer.from_pretrained(checkpoint)

def tokenize_function(example):

    return tokenizer(example["sentence1"], example["sentence2"], truncation=True)

tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
```

```
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Training

The first step before we can define our `Trainer` is to define a `TrainingArguments` class that will contain all the hyperparameters the `Trainer` will use for training and evaluation. The only argument you have to provide is a directory where the trained model will be saved, as well as the checkpoints along the way. For all the rest, you can leave the defaults, which should work pretty well for a basic fine-tuning.

```
from transformers import TrainingArguments

training_args = TrainingArguments("test-trainer")
```

💡 If you want to automatically upload your model to the Hub during training, pass along `push_to_hub=True` in the `TrainingArguments`. We will learn more about this in [Chapter 4](#)

The second step is to define our model. As in the [previous chapter](#), we will use the `AutoModelForSequenceClassification` class, with two labels:

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
```

You will notice that unlike in [Chapter 2](#), you get a warning after instantiating this pretrained model. This is because BERT has not been pretrained on classifying pairs of sentences, so the head of the pretrained model has been discarded and a new head suitable for sequence classification has been added instead. The warnings indicate that some weights were not used (the ones corresponding to the dropped pretraining head) and that some others were randomly initialized (the ones for the new head). It concludes by encouraging you to train the model, which is exactly what we are going to do now.

Once we have our model, we can define a `Trainer` by passing it all the objects constructed up to now – the `model`, the `training_args`, the training and validation datasets, our `data_collator`, and our `tokenizer`:

```
from transformers import Trainer

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

Note that when you pass the `tokenizer` as we did here, the default `data_collator` used by the `Trainer` will be a `DataCollatorWithPadding` as defined previously, so you can skip the line `data_collator=data_collator` in this call. It was still important to show you this part of the processing in section 2!

To fine-tune the model on our dataset, we just have to call the `train()` method of our `Trainer`:

```
trainer.train()
```

This will start the fine-tuning (which should take a couple of minutes on a GPU) and report the training loss every 500 steps. It won't, however, tell you how well (or badly) your model is performing. This is because:

1. We didn't tell the `Trainer` to evaluate during training by setting `evaluation_strategy` to either `"steps"` (evaluate every `eval_steps`) or `"epoch"` (evaluate at the end of each epoch).
2. We didn't provide the `Trainer` with a `compute_metrics()` function to calculate a metric during said evaluation (otherwise the evaluation would just have printed the loss, which is not a very intuitive number).

Evaluation

Let's see how we can build a useful `compute_metrics()` function and use it the next time we train. The function must take an `EvalPrediction` object (which is a named tuple with a `predictions` field and a `label_ids` field) and will return a dictionary mapping strings to floats (the strings being the names of the metrics returned, and the floats their values). To get some predictions from our model, we can use the `Trainer.predict()` command:

```
predictions = trainer.predict(tokenized_datasets["validation"])  
  
print(predictions.predictions.shape, predictions.label_ids.shape)
```


```
(408, 2) (408,)
```

The output of the `predict()` method is another named tuple with three fields: `predictions`, `label_ids`, and `metrics`. The `metrics` field will just contain the loss on the dataset passed, as well as some time metrics (how long it took to predict, in total and on average). Once we complete our `compute_metrics()` function and pass it to the `Trainer`, that field will also contain the metrics returned by `compute_metrics()`.

As you can see, `predictions` is a two-dimensional array with shape `408 x 2` (408 being the number of elements in the dataset we used). Those are the logits for each element of the dataset we passed to `predict()` (as you saw in the [previous chapter](#), all Transformer models return logits). To transform them into predictions that we can compare to our labels, we need to take the index with the maximum value on the second axis:

```
import numpy as np
```

```
preds = np.argmax(predictions.predictions, axis=-1)
```

We can now compare those `preds` to the labels. To build our `compute_metric()` function, we will rely on the metrics from the  [Evaluate](#) library. We can load the metrics associated with the MRPC dataset as easily as we loaded the dataset, this time with the `evaluate.load()` function. The object returned has a `compute()` method we can use to do the metric calculation:

```
import evaluate

metric = evaluate.load("glue", "mrpc")

metric.compute(predictions=preds, references=predictions.label_ids)
```

```
{'accuracy': 0.8578431372549019, 'f1': 0.8996539792387542}
```

The exact results you get may vary, as the random initialization of the model head might change the metrics it achieved. Here, we can see our model has an accuracy of 85.78% on the validation set and an F1 score of 89.97. Those are the two metrics used to evaluate results on the MRPC dataset for the GLUE benchmark. The table in the [BERT paper](#) reported an F1 score of 88.9 for the base model. That was the `uncased` model while we are currently using the `cased` model, which explains the better result.

Wrapping everything together, we get our `compute_metrics()` function:

```
def compute_metrics(eval_preds):

    metric = evaluate.load("glue", "mrpc")

    logits, labels = eval_preds
```



```
predictions = np.argmax(logits, axis=-1)
```

```
return metric.compute(predictions=predictions, references=labels)
```

And to see it used in action to report metrics at the end of each epoch, here is how we define a new `Trainer` with this `compute_metrics()` function:

```
training_args = TrainingArguments("test-trainer", evaluation_strategy="epoch")

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)

trainer = Trainer(

    model,

    training_args,

    train_dataset=tokenized_datasets["train"],

    eval_dataset=tokenized_datasets["validation"],

    data_collator=data_collator,

    tokenizer=tokenizer,

    compute_metrics=compute_metrics,

)
```


Note that we create a new `TrainingArguments` with its `evaluation_strategy` set to "epoch" and a new model — otherwise, we would just be continuing the training of the model we have already trained. To launch a new training run, we execute:

```
trainer.train()
```

This time, it will report the validation loss and metrics at the end of each epoch on top of the training loss. Again, the exact accuracy/F1 score you reach might be a bit different from what we found, because of the random head initialization of the model, but it should be in the same ballpark.

The `Trainer` will work out of the box on multiple GPUs or TPUs and provides lots of options, like mixed-precision training (use `fp16 = True` in your training arguments). We will go over everything it supports in Chapter 10.

This concludes the introduction to fine-tuning using the `Trainer` API. An example of doing this for most common NLP tasks will be given in [Chapter 7](#), but for now let's look at how to do the same thing in pure PyTorch.

 **Try it out!** Fine-tune a model on the GLUE SST-2 dataset, using the data processing you did in section 2.

A full training

Now we'll see how to achieve the same results as we did in the last section without using the `Trainer` class. Again, we assume you have done the data processing in section 2. Here is a short summary covering everything you will need:

```
from datasets import load_dataset

from transformers import AutoTokenizer, DataCollatorWithPadding

raw_datasets = load_dataset("glue", "mrpc")

checkpoint = "bert-base-uncased"

tokenizer = AutoTokenizer.from_pretrained(checkpoint)

def tokenize_function(example):

    return tokenizer(example["sentence1"], example["sentence2"], truncation=True)

tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)

data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Prepare for training

Before actually writing our training loop, we will need to define a few objects. The first ones are the dataloaders we will use to iterate over batches. But before we can define those dataloaders, we need to apply a bit of postprocessing to our `tokenized_datasets`, to take care of some things that the `Trainer` did for us automatically. Specifically, we need to:

- Remove the columns corresponding to values the model does not expect (like the `sentence1` and `sentence2` columns).
- Rename the column `label` to `labels` (because the model expects the argument to be named `labels`).
- Set the format of the datasets so they return PyTorch tensors instead of lists.

Our `tokenized_datasets` has one method for each of those steps:

```
tokenized_datasets = tokenized_datasets.remove_columns(["sentence1", "sentence2",
"idx"])

tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

tokenized_datasets.set_format("torch")

tokenized_datasets["train"].column_names
```

We can then check that the result only has columns that our model will accept:

```
["attention_mask", "input_ids", "labels", "token_type_ids"]
```

Now that this is done, we can easily define our dataloaders:

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(
```

```
    tokenized_datasets["train"], shuffle=True, batch_size=8, collate_fn=data_collator
)

eval_dataloader = DataLoader(

    tokenized_datasets["validation"], batch_size=8, collate_fn=data_collator
)
```

To quickly check there is no mistake in the data processing, we can inspect a batch like this:

```
for batch in train_dataloader:

    break

{k: v.shape for k, v in batch.items()}
```

```
{'attention_mask': torch.Size([8, 65]),
 'input_ids': torch.Size([8, 65]),
 'labels': torch.Size([8]),
 'token_type_ids': torch.Size([8, 65])}
```

Note that the actual shapes will probably be slightly different for you since we set `shuffle=True` for the training dataloader and we are padding to the maximum length inside the batch.

Now that we're completely finished with data preprocessing (a satisfying yet elusive goal for any ML practitioner), let's turn to the model. We instantiate it exactly as we did in the previous section:

```
from transformers import AutoModelForSequenceClassification
```

```
model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
```

To make sure that everything will go smoothly during training, we pass our batch to this model:

```
outputs = model(**batch)
```

```
print(outputs.loss, outputs.logits.shape)
```

```
tensor(0.5441, grad_fn=<NllLossBackward>) torch.Size([8, 2])
```

All 🤖 Transformers models will return the loss when `labels` are provided, and we also get the logits (two for each input in our batch, so a tensor of size 8 x 2).

We're almost ready to write our training loop! We're just missing two things: an optimizer and a learning rate scheduler. Since we are trying to replicate what the `Trainer` was doing by hand, we will use the same defaults. The optimizer used by the `Trainer` is AdamW, which is the same as Adam, but with a twist for weight decay regularization (see [“Decoupled Weight Decay Regularization”](#) by Ilya Loshchilov and Frank Hutter):

```
from transformers import AdamW
```

```
optimizer = AdamW(model.parameters(), lr=5e-5)
```

Finally, the learning rate scheduler used by default is just a linear decay from the maximum value (5e-5) to 0. To properly define it, we need to know the number of training steps we will take, which is the

number of epochs we want to run multiplied by the number of training batches (which is the length of our training dataloader). The `Trainer` uses three epochs by default, so we will follow that:

```
from transformers import get_scheduler

num_epochs = 3

num_training_steps = num_epochs * len(train_dataloader)

lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps,
)

print(num_training_steps)
```

1377

The training loop

One last thing: we will want to use the GPU if we have access to one (on a CPU, training might take several hours instead of a couple of minutes). To do this, we define a `device` we will put our model and our batches on:

```
import torch
```

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

```
model.to(device)
```

```
device
```

```
device(type='cuda')
```

We are now ready to train! To get some sense of when training will be finished, we add a progress bar over our number of training steps, using the `tqdm` library:

```
from tqdm.auto import tqdm
```

```
progress_bar = tqdm(range(num_training_steps))
```

```
model.train()
```

```
for epoch in range(num_epochs):
```

```
    for batch in train_dataloader:
```

```
        batch = {k: v.to(device) for k, v in batch.items()}
```

```
        outputs = model(**batch)
```

```
        loss = outputs.loss
```



```
loss.backward()

optimizer.step()


lr_scheduler.step()

optimizer.zero_grad()

progress_bar.update(1)
```

You can see that the core of the training loop looks a lot like the one in the introduction. We didn't ask for any reporting, so this training loop will not tell us anything about how the model fares. We need to add an evaluation loop for that.

The evaluation loop

As we did earlier, we will use a metric provided by the  Evaluate library. We've already seen the `metric.compute()` method, but metrics can actually accumulate batches for us as we go over the prediction loop with the method `add_batch()`. Once we have accumulated all the batches, we can get the final result with `metric.compute()`. Here's how to implement all of this in an evaluation loop:

```
import evaluate

metric = evaluate.load("glue", "mrpc")

model.eval()

for batch in eval_dataloader:

    batch = {k: v.to(device) for k, v in batch.items()}

    with torch.no_grad():

        outputs = model(**batch)
```

```
logits = outputs.logits


predictions = torch.argmax(logits, dim=-1)

metric.add_batch(predictions=predictions, references=batch["labels"])


metric.compute()
```

```
{'accuracy': 0.8431372549019608, 'f1': 0.8907849829351535}
```

Again, your results will be slightly different because of the randomness in the model head initialization and the data shuffling, but they should be in the same ballpark.

 **Try it out!** Modify the previous training loop to fine-tune your model on the SST-2 dataset.

Supercharge your training loop with Accelerate

The training loop we defined earlier works fine on a single CPU or GPU. But using the  [Accelerate](#) library, with just a few adjustments we can enable distributed training on multiple GPUs or TPUs. Starting from the creation of the training and validation dataloaders, here is what our manual training loop looks like:

```
from transformers import AdamW, AutoModelForSequenceClassification, get_scheduler

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)

optimizer = AdamW(model.parameters(), lr=3e-5)
```

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

model.to(device)

num_epochs = 3

num_training_steps = num_epochs * len(train_dataloader)

lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps,
)

progress_bar = tqdm(range(num_training_steps))

model.train()

for epoch in range(num_epochs):
    for batch in train_dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}

        outputs = model(**batch)

        loss = outputs.loss

        loss.backward()
```

```
optimizer.step()

lr_scheduler.step()

optimizer.zero_grad()

progress_bar.update(1)
```

And here are the changes:

```
+ from accelerate import Accelerator

from transformers import AdamW, AutoModelForSequenceClassification, get_scheduler

+ accelerator = Accelerator()

model = AutoModelForSequenceClassification.from_pretrained(checkpoint,
num_labels=2)

optimizer = AdamW(model.parameters(), lr=3e-5)

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

- model.to(device)

+ train_dataloader, eval_dataloader, model, optimizer = accelerator.prepare(
+     train_dataloader, eval_dataloader, model, optimizer
```

+)

```
num_epochs = 3

num_training_steps = num_epochs * len(train_dataloader)

lr_scheduler = get_scheduler(

    "linear",

    optimizer=optimizer,

    num_warmup_steps=0,

    num_training_steps=num_training_steps

)
```

```
progress_bar = tqdm(range(num_training_steps))
```

```
model.train()
```

```
for epoch in range(num_epochs):
```

```
    for batch in train_dataloader:
```

```
-         batch = {k: v.to(device) for k, v in batch.items()}
```

```
        outputs = model(**batch)
```

```
        loss = outputs.loss
```

```
-         loss.backward()
```

```
+         accelerator.backward(loss)
```

```
optimizer.step()

lr_scheduler.step()

optimizer.zero_grad()

progress_bar.update(1)
```

The first line to add is the import line. The second line instantiates an `Accelerator` object that will look at the environment and initialize the proper distributed setup. 🤖 Accelerate handles the device placement for you, so you can remove the lines that put the model on the device (or, if you prefer, change them to use `accelerator.device` instead of `device`).

Then the main bulk of the work is done in the line that sends the dataloaders, the model, and the optimizer to `accelerator.prepare()`. This will wrap those objects in the proper container to make sure your distributed training works as intended. The remaining changes to make are removing the line that puts the batch on the `device` (again, if you want to keep this you can just change it to use `accelerator.device`) and replacing `loss.backward()` with `accelerator.backward(loss)`.

⚠ In order to benefit from the speed-up offered by Cloud TPUs, we recommend padding your samples to a fixed length with the ``padding="max_length"`` and ``max_length`` arguments of the tokenizer.

If you'd like to copy and paste it to play around, here's what the complete training loop looks like with 🤖 Accelerate:

```
from accelerate import Accelerator

from transformers import AdamW, AutoModelForSequenceClassification, get_scheduler

accelerator = Accelerator()

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
```

```
optimizer = AdamW(model.parameters(), lr=3e-5)

train_dl, eval_dl, model, optimizer = accelerator.prepare(

    train_dataloader, eval_dataloader, model, optimizer

)

num_epochs = 3

num_training_steps = num_epochs * len(train_dl)

lr_scheduler = get_scheduler(

    "linear",

    optimizer=optimizer,

    num_warmup_steps=0,

    num_training_steps=num_training_steps,

)

progress_bar = tqdm(range(num_training_steps))

model.train()

for epoch in range(num_epochs):

    for batch in train_dl:

        outputs = model(**batch)
```

```
loss = outputs.loss

accelerator.backward(loss)

optimizer.step()

lr_scheduler.step()

optimizer.zero_grad()

progress_bar.update(1)
```

Putting this in a `train.py` script will make that script runnable on any kind of distributed setup. To try it out in your distributed setup, run the command:

```
accelerate config
```

which will prompt you to answer a few questions and dump your answers in a configuration file used by this command:

```
accelerate launch train.py
```

which will launch the distributed training.

If you want to try this in a Notebook (for instance, to test it with TPUs on Colab), just paste the code in a `training_function()` and run a last cell with:

```
from accelerate import notebook_launcher
```



```
notebook_launcher(training_function)
```

You can find more examples in the [📖 Accelerate repo](#).

Fine-tuning, Check!



That was fun! In the first two chapters you learned about models and tokenizers, and now you know how to fine-tune them for your own data. To recap, in this chapter you:

- Learned about datasets in the [Hub](#)
- Learned how to load and preprocess datasets, including using dynamic padding and collators
- Implemented your own fine-tuning and evaluation of a model
- Implemented a lower-level training loop
- Used 🤖 Accelerate to easily adapt your training loop so it works for multiple GPUs or TPUs

Sharing Models and Transformers | The Hugging Face Hub



The [Hugging Face Hub](#) -- our main website -- is a central platform that enables anyone to discover, use, and contribute new state-of-the-art models and datasets. It hosts a wide variety of models, with more than 10,000 publicly available. We'll focus on the models in this chapter, and take a look at the datasets in Chapter 5.

The models in the Hub are not limited to 🤖 Transformers or even NLP. There are models from [Flair](#) and [AllenNLP](#) for NLP, [Asteroid](#) and [pyannote](#) for speech, and [timm](#) for vision, to name a few.

Each of these models is hosted as a Git repository, which allows versioning and reproducibility. Sharing a model on the Hub means opening it up to the community and making it accessible to anyone looking to easily use it, in turn eliminating their need to train a model on their own and simplifying sharing and usage.

Additionally, sharing a model on the Hub automatically deploys a hosted Inference API for that model. Anyone in the community is free to test it out directly on the model's page, with custom inputs and appropriate widgets.

The best part is that sharing and using any public model on the Hub is completely free! [Paid plans](#) also exist if you wish to share models privately.

The video below shows how to navigate the Hub.

Having a huggingface.co account is required to follow along this part, as we'll be creating and managing repositories on the Hugging Face Hub: [create an account](#)

Using pretrained models

The Model Hub makes selecting the appropriate model simple, so that using it in any downstream library can be done in a few lines of code. Let's take a look at how to actually use one of these models, and how to contribute back to the community.

We select the `camembert-base` checkpoint to try it out. The identifier `camembert-base` is all we need to start using it! As you've seen in previous chapters, we can instantiate it using the `pipeline()` function:

```
from transformers import pipeline

camembert_fill_mask = pipeline("fill-mask", model="camembert-base")

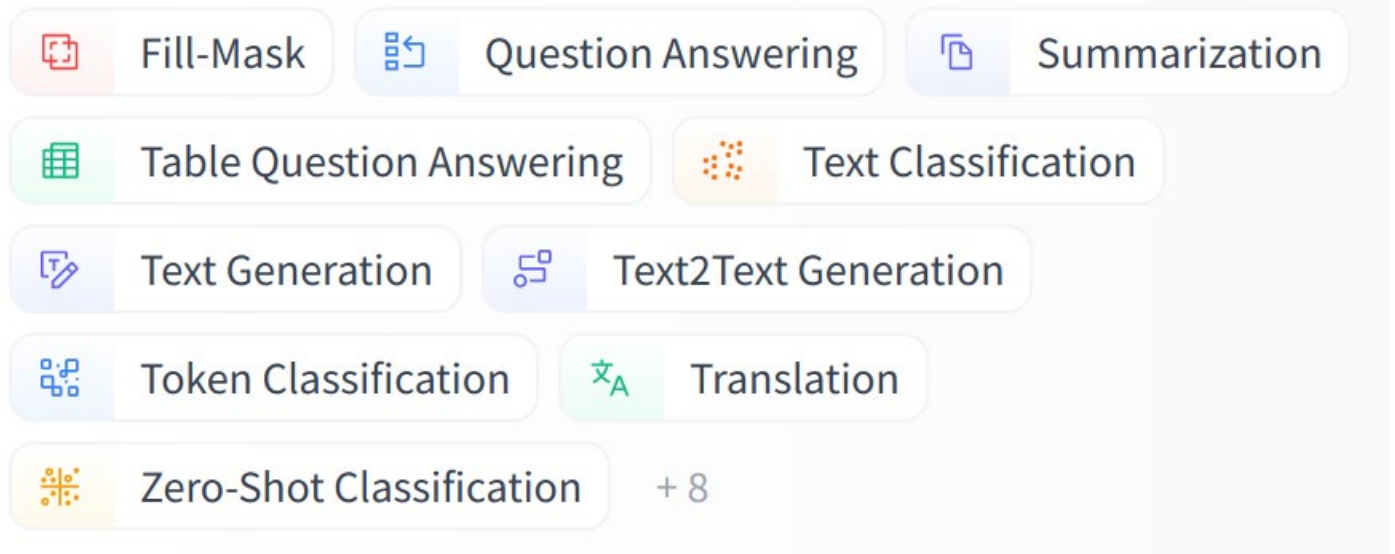
results = camembert_fill_mask("Le camembert est <mask> :)")
```

```
[
  {'sequence': 'Le camembert est délicieux :)', 'score': 0.49091005325317383,
  'token': 7200, 'token_str': 'délicieux'},
  {'sequence': 'Le camembert est excellent :)', 'score': 0.1055697426199913, 'token':
  2183, 'token_str': 'excellent'},
  {'sequence': 'Le camembert est succulent :)', 'score': 0.03453313186764717,
  'token': 26202, 'token_str': 'succulent'},
  {'sequence': 'Le camembert est meilleur :)', 'score': 0.0330314114689827, 'token':
  528, 'token_str': 'meilleur'},
```

```
{'sequence': 'Le camembert est parfait :)', 'score': 0.03007650189101696, 'token': 1654, 'token_str': 'parfait'}  
]
```

As you can see, loading a model within a pipeline is extremely simple. The only thing you need to watch out for is that the chosen checkpoint is suitable for the task it's going to be used for. For example, here we are loading the `camembert-base` checkpoint in the `fill-mask` pipeline, which is completely fine. But if we were to load this checkpoint in the `text-classification` pipeline, the results would not make any sense because the head of `camembert-base` is not suitable for this task! We recommend using the task selector in the Hugging Face Hub interface in order to select the appropriate checkpoints:

Tasks



You can also instantiate the checkpoint using the model architecture directly:

```
from transformers import CamembertTokenizer, CamembertForMaskedLM  
  
tokenizer = CamembertTokenizer.from_pretrained("camembert-base")  
  
model = CamembertForMaskedLM.from_pretrained("camembert-base")
```

However, we recommend using the [Auto* classes](#) instead, as these are by design architecture-agnostic. While the previous code sample limits users to checkpoints loadable in the CamemBERT architecture, using the Auto* classes makes switching checkpoints simple:


```
from transformers import AutoTokenizer, AutoModelForMaskedLM

tokenizer = AutoTokenizer.from_pretrained("camembert-base")

model = AutoModelForMaskedLM.from_pretrained("camembert-base")
```

When using a pretrained model, make sure to check how it was trained, on which datasets, its limits, and its biases. All of this information should be indicated on its model card.

Sharing pretrained models

In the steps below, we'll take a look at the easiest ways to share pretrained models to the  Hub. There are tools and utilities available that make it simple to share and update models directly on the Hub, which we will explore below.

We encourage all users that train models to contribute by sharing them with the community — sharing models, even when trained on very specific datasets, will help others, saving them time and compute resources and providing access to useful trained artifacts. In turn, you can benefit from the work that others have done!

There are three ways to go about creating new model repositories:

- Using the `push_to_hub` API
- Using the `huggingface_hub` Python library
- Using the web interface

Once you've created a repository, you can upload files to it via `git` and `git-lfs`. We'll walk you through creating model repositories and uploading files to them in the following sections.

Using the `push_to_hub` API

The simplest way to upload files to the Hub is by leveraging the `push_to_hub` API.

Before going further, you'll need to generate an authentication token so that the `huggingface_hub` API knows who you are and what namespaces you have write access to. Make sure you are in an environment where you have `transformers` installed (see [Setup](#)). If you are in a notebook, you can use the following function to login:

```
from huggingface_hub import notebook_login
```

```
notebook_login()
```

In a terminal, you can run:

```
huggingface-cli login
```

In both cases, you should be prompted for your username and password, which are the same ones you use to log in to the Hub. If you do not have a Hub profile yet, you should create one [here](#).

Great! You now have your authentication token stored in your cache folder. Let's create some repositories!

If you have played around with the `Trainer` API to train a model, the easiest way to upload it to the Hub is to set `push_to_hub=True` when you define your `TrainingArguments`:

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    "bert-finetuned-mrpc", save_strategy="epoch", push_to_hub=True
)
```

When you call `trainer.train()`, the `Trainer` will then upload your model to the Hub each time it is saved (here every epoch) in a repository in your namespace. That repository will be named like the output directory you picked (here `bert-finetuned-mrpc`) but you can choose a different name with `hub_model_id = "a_different_name"`.

To upload your model to an organization you are a member of, just pass it with `hub_model_id = "my_organization/my_repo_name"`.

Once your training is finished, you should do a final `trainer.push_to_hub()` to upload the last version of your model. It will also generate a model card with all the relevant metadata, reporting the hyperparameters used and the evaluation results! Here is an example of the content you might find in a such a model card:

Training procedure

Training hyperparameters

The following hyperparameters were used during training:

- learning_rate: 5e-05
- train_batch_size: 32
- eval_batch_size: 32
- seed: 42
- distributed_type: multi-GPU
- num_devices: 2
- total_train_batch_size: 64
- total_eval_batch_size: 64
- optimizer: Adam with betas=(0.9,0.999) and epsilon=1e-08
- lr_scheduler_type: linear
- num_epochs: 3.0
- mixed_precision_training: Native AMP

Training results

Training Loss	Epoch	Step	Validation Loss	Accuracy	F1	Combined Score
0.6093	1.0	58	0.542145	0.727941	0.833083	0.780512
0.4107	2.0	116	0.391782	0.816176	0.859813	0.837995
0.2512	3.0	174	0.419669	0.852941	0.896907	0.874924

At a lower level, accessing the Model Hub can be done directly on models, tokenizers, and configuration objects via their `push_to_hub()` method. This method takes care of both the repository creation and

pushing the model and tokenizer files directly to the repository. No manual handling is required, unlike with the API we'll see below.

To get an idea of how it works, let's first initialize a model and a tokenizer:

```
from transformers import AutoModelForMaskedLM, AutoTokenizer

checkpoint = "camembert-base"

model = AutoModelForMaskedLM.from_pretrained(checkpoint)

tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

You're free to do whatever you want with these — add tokens to the tokenizer, train the model, fine-tune it. Once you're happy with the resulting model, weights, and tokenizer, you can leverage the `push_to_hub()` method directly available on the `model` object:

```
model.push_to_hub("dummy-model")
```

This will create the new repository `dummy-model` in your profile, and populate it with your model files. Do the same with the tokenizer, so that all the files are now available in this repository:

```
tokenizer.push_to_hub("dummy-model")
```

If you belong to an organization, simply specify the `organization` argument to upload to that organization's namespace:

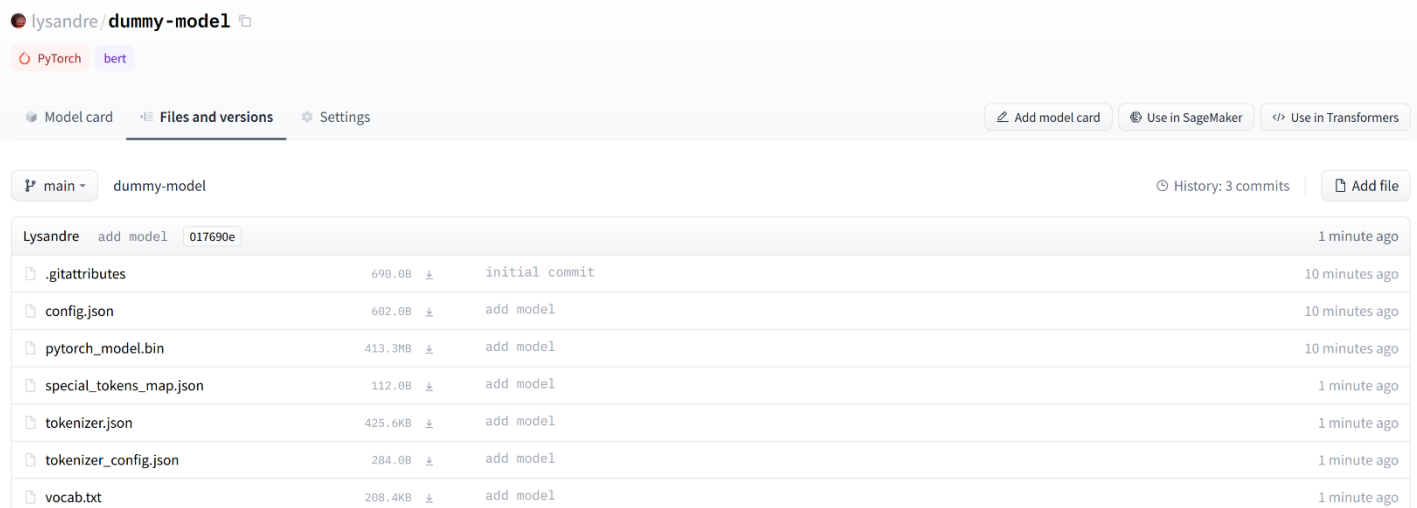
```
tokenizer.push_to_hub("dummy-model", organization="huggingface")
```

If you wish to use a specific Hugging Face token, you're free to specify it to the `push_to_hub()` method as well:

```
tokenizer.push_to_hub("dummy-model", organization="huggingface",  
use_auth_token="<TOKEN>")
```

Now head to the Model Hub to find your newly uploaded model: <https://huggingface.co/user-or-organization/dummy-model>.

Click on the “Files and versions” tab, and you should see the files visible in the following screenshot:



lysandre / dummy-model

PyTorch bert

Model card Files and versions Settings Add model card Use in SageMaker Use in Transformers

main dummy-model History: 3 commits Add file

File	Size	Commit	Time
lysandre add model 017690e			1 minute ago
.gitattributes	690.0B	initial commit	10 minutes ago
config.json	602.0B	add model	10 minutes ago
pytorch_model.bin	413.3MB	add model	10 minutes ago
special_tokens_map.json	112.0B	add model	1 minute ago
tokenizer.json	425.6KB	add model	1 minute ago
tokenizer_config.json	284.0B	add model	1 minute ago
vocab.txt	288.4KB	add model	1 minute ago

Try it out! Take the model and tokenizer associated with the `bert-base-cased` checkpoint and upload them to a repo in your namespace using the `push_to_hub()` method. Double-check that the repo appears properly on your page before deleting it.

As you've seen, the `push_to_hub()` method accepts several arguments, making it possible to upload to a specific repository or organization namespace, or to use a different API token. We recommend you take a look at the method specification available directly in the [Transformers documentation](#) to get an idea of what is possible.

The `push_to_hub()` method is backed by the [huggingface_hub](#) Python package, which offers a direct API to the Hugging Face Hub. It's integrated within [Transformers](#) and several other machine learning libraries, like [allenlp](#). Although we focus on the [Transformers](#) integration in this chapter, integrating it into your own code or library is simple.

Jump to the last section to see how to upload files to your newly created repository!

Using the `huggingface_hub` Python library

The `huggingface_hub` Python library is a package which offers a set of tools for the model and datasets hubs. It provides simple methods and classes for common tasks like getting information about repositories on the hub and managing them. It provides simple APIs that work on top of git to manage those repositories' content and to integrate the Hub in your projects and libraries.

Similarly to using the `push_to_hub` API, this will require you to have your API token saved in your cache. In order to do this, you will need to use the `login` command from the CLI, as mentioned in the previous section (again, make sure to prepend these commands with the `!` character if running in Google Colab):

```
huggingface-cli login
```

The `huggingface_hub` package offers several methods and classes which are useful for our purpose. Firstly, there are a few methods to manage repository creation, deletion, and others:

```
from huggingface_hub import (  
  
    ## User management  
  
    login,  
  
    logout,  
  
    whoami,  
  
    ## Repository creation and management  
  
    create_repo,  
  
    delete_repo,
```

```
update_repo_visibility,  
  
# And some methods to retrieve/change information about the content  
  
list_models,  
  
list_datasets,  
  
list_metrics,  
  
list_repo_files,  
  
upload_file,  
  
delete_file,  
  
)
```

Additionally, it offers the very powerful `Repository` class to manage a local repository. We will explore these methods and that class in the next few sections to understand how to leverage them.

The `create_repo` method can be used to create a new repository on the hub:

```
from huggingface_hub import create_repo  
  
create_repo("dummy-model")
```

This will create the repository `dummy-model` in your namespace. If you like, you can specify which organization the repository should belong to using the `organization` argument:

```
from huggingface_hub import create_repo
```

```
create_repo("dummy-model", organization="huggingface")
```

This will create the `dummy-model` repository in the `huggingface` namespace, assuming you belong to that organization. Other arguments which may be useful are:

- `private`, in order to specify if the repository should be visible from others or not.
- `token`, if you would like to override the token stored in your cache by a given token.
- `repo_type`, if you would like to create a `dataset` or a `space` instead of a model. Accepted values are `"dataset"` and `"space"`.

Once the repository is created, we should add files to it! Jump to the next section to see the three ways this can be handled.

Using the web interface

The web interface offers tools to manage repositories directly in the Hub. Using the interface, you can easily create repositories, add files (even large ones!), explore models, visualize diffs, and much more.

To create a new repository, visit huggingface.co/new:



Create a new model repository

A repository contains all model files, including the revision history.

Owner

Model name



Public

Anyone on the internet can see this model. Only you (personal model) or members of your organization (organization model) can commit.



Private

Only you (personal model) or members of your organization (organization model) can see and commit to this model.

Create model

First, specify the owner of the repository: this can be either you or any of the organizations you're affiliated with. If you choose an organization, the model will be featured on the organization's page and every member of the organization will have the ability to contribute to the repository.

Next, enter your model's name. This will also be the name of the repository. Finally, you can specify whether you want your model to be public or private. Private models are hidden from public view.

After creating your model repository, you should see a page like this:

lysandre / dummy

Model card Files and versions Settings

Add model card How to clone

No model card

New: Create and edit your model card directly on the website!

Create Model Card

Downloads last month
0

Hosted inference API

Unable to determine this model's pipeline type. Check the docs.

This is where your model will be hosted. To start populating it, you can add a README file directly from the web interface.

lysandre / dummy

Model card Files and versions Settings

Edit model card How to clone

dummy / README.md

Edit Preview

```

1 # Dummy model
2
3 This is a dummy model

```

Commit changes

Update README.md

Add an extended description...

Commit changes Cancel

The README file is in Markdown — feel free to go wild with it! The third part of this chapter is dedicated to building a model card. These are of prime importance in bringing value to your model, as they're where you tell others what it can do.

If you look at the “Files and versions” tab, you’ll see that there aren’t many files there yet — just the `README.md` you just created and the `.gitattributes` file that keeps track of large files.

main dummy History: 2 commits Add file

lysandre	Create README.md	891b41d
.gitattributes	690.0B	initial commit 6 minutes ago
README.md	36.0B	Create README.md 2 minutes ago


We'll take a look at how to add some new files next.

Uploading the model files

The system to manage files on the Hugging Face Hub is based on git for regular files, and git-lfs (which stands for [Git Large File Storage](#)) for larger files.

In the next section, we go over three different ways of uploading files to the Hub: through `huggingface_hub` and through git commands.

The `upload_file` approach

Using `upload_file` does not require git and git-lfs to be installed on your system. It pushes files directly to the  Hub using HTTP POST requests. A limitation of this approach is that it doesn't handle files that are larger than 5GB in size. If your files are larger than 5GB, please follow the two other methods detailed below.

The API may be used as follows:

```
from huggingface_hub import upload_file

upload_file(
    "<path_to_file>/config.json",
    path_in_repo="config.json",
    repo_id="<namespace>/dummy-model",
)
```

This will upload the file `config.json` available at `<path_to_file>` to the root of the repository as `config.json`, to the `dummy-model` repository. Other arguments which may be useful are:

- `token`, if you would like to override the token stored in your cache by a given token.
- `repo_type`, if you would like to upload to a `dataset` or a `space` instead of a model. Accepted values are `"dataset"` and `"space"`.

The Repository class

The `Repository` class manages a local repository in a git-like manner. It abstracts most of the pain points one may have with git to provide all features that we require.

Using this class requires having git and git-lfs installed, so make sure you have git-lfs installed (see [here](#) for installation instructions) and set up before you begin.

In order to start playing around with the repository we have just created, we can start by initialising it into a local folder by cloning the remote repository:

```
from huggingface_hub import Repository

repo = Repository("<path_to_dummy_folder>", clone_from="<namespace>/dummy-model")
```

This created the folder `<path_to_dummy_folder>` in our working directory. This folder only contains the `.gitattributes` file as that's the only file created when instantiating the repository through `create_repo`.

From this point on, we may leverage several of the traditional git methods:

```
repo.git_pull()

repo.git_add()

repo.git_commit()

repo.git_push()

repo.git_tag()
```

And others! We recommend taking a look at the `Repository` documentation available [here](#) for an overview of all available methods.

At present, we have a model and a tokenizer that we would like to push to the hub. We have successfully cloned the repository, we can therefore save the files within that repository.

We first make sure that our local clone is up to date by pulling the latest changes:

```
repo.git_pull()
```

Once that is done, we save the model and tokenizer files:

```
model.save_pretrained("<path_to_dummy_folder>")  
tokenizer.save_pretrained("<path_to_dummy_folder>")
```

The `<path_to_dummy_folder>` now contains all the model and tokenizer files. We follow the usual git workflow by adding files to the staging area, committing them and pushing them to the hub:

```
repo.git_add()  
repo.git_commit("Add model and tokenizer files")  
repo.git_push()
```

Congratulations! You just pushed your first files on the hub.

The git-based approach

This is the very barebones approach to uploading files: we'll do so with git and git-lfs directly. Most of the difficulty is abstracted away by previous approaches, but there are a few caveats with the following method so we'll follow a more complex use-case.

Using this class requires having git and git-lfs installed, so make sure you have [git-lfs](#) installed (see here for installation instructions) and set up before you begin.

First start by initializing git-lfs:

```
git lfs install
```

Updated git hooks.

Git LFS initialized.

Once that's done, the first step is to clone your model repository:

```
git clone https://huggingface.co/<namespace>/<your-model-id>
```

My username is `lysandre` and I've used the model name `dummy`, so for me the command ends up looking like the following:

```
git clone https://huggingface.co/lysandre/dummy
```

I now have a folder named `dummy` in my working directory. I can `cd` into the folder and have a look at the contents:

```
cd dummy && ls
```

README.md

If you just created your repository using Hugging Face Hub's `create_repo` method, this folder should only contain a hidden `.gitattributes` file. If you followed the instructions in the previous section to create a repository using the web interface, the folder should contain a single `README.md` file alongside the hidden `.gitattributes` file, as shown here.

Adding a regular-sized file, such as a configuration file, a vocabulary file, or basically any file under a few megabytes, is done exactly as one would do it in any git-based system. However, bigger files must be registered through git-lfs in order to push them to `huggingface.co`.

Let's go back to Python for a bit to generate a model and tokenizer that we'd like to commit to our dummy repository:

```
from transformers import AutoModelForMaskedLM, AutoTokenizer
```

```
checkpoint = "camembert-base"
```

```
model = AutoModelForMaskedLM.from_pretrained(checkpoint)
```

```
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

```
# Do whatever with the model, train it, fine-tune it...
```

```
model.save_pretrained("<path_to_dummy_folder>")
```


```
tokenizer.save_pretrained("<path_to_dummy_folder>")
```

Now that we've saved some model and tokenizer artifacts, let's take another look at the `dummy` folder:

```
ls
```

```
config.json  pytorch_model.bin  README.md  sentencepiece.bpe.model  
special_tokens_map.json  tokenizer_config.json  tokenizer.json
```

If you look at the file sizes (for example, with `ls -lh`), you should see that the model state dict file (`pytorch_model.bin`) is the only outlier, at more than 400 MB.

 When creating the repository from the web interface, the `.gitattributes` file is automatically set up to consider files with certain extensions, such as `*.bin*` and `*.h5*`, as large files, and git-lfs will track them with no necessary setup on your side.

We can now go ahead and proceed like we would usually do with traditional Git repositories. We can add all the files to Git's staging environment using the `git add` command:

```
git add .
```

We can then have a look at the files that are currently staged:

```
git status
```

```
On branch main
```

```
Your branch is up to date with 'origin/main'.
```

Changes to be committed:

(use `"git restore --staged <file>..."` to unstage)

modified: .gitattributes

new file: config.json

new file: pytorch_model.bin

new file: sentencepiece.bpe.model

new file: special_tokens_map.json

new file: tokenizer.json

new file: tokenizer_config.json

Similarly, we can make sure that git-lfs is tracking the correct files by using its `status` command:

```
git lfs status
```

On branch main

Objects to be pushed to origin/main:

Objects to be committed:

config.json (Git: bc20ff2)

```
pytorch_model.bin (LFS: 35686c2)

sentencepiece.bpe.model (LFS: 988bc5a)

special_tokens_map.json (Git: cb23931)

tokenizer.json (Git: 851ff3e)

tokenizer_config.json (Git: f0f7783)
```

Objects not staged **for** commit:

We can see that all files have `Git` as a handler, except `pytorch_model.bin` and `sentencepiece.bpe.model`, which have `LFS`. Great!

Let's proceed to the final steps, committing and pushing to the `huggingface.co` remote repository:

```
git commit -m "First model version"
```

```
[main b08aab1] First model version

7 files changed, 29027 insertions(+)

6 files changed, 36 insertions(+)

create mode 100644 config.json

create mode 100644 pytorch_model.bin

create mode 100644 sentencepiece.bpe.model
```



```
create mode 100644 special_tokens_map.json
```

```
create mode 100644 tokenizer.json
```

```
create mode 100644 tokenizer_config.json
```

Pushing can take a bit of time, depending on the speed of your internet connection and the size of your files:

```
git push
```

```
Uploading LFS objects: 100% (1/1), 433 MB | 1.3 MB/s, done.
```

```
Enumerating objects: 11, done.
```

```
Counting objects: 100% (11/11), done.
```

```
Delta compression using up to 12 threads
```

```
Compressing objects: 100% (9/9), done.
```

```
Writing objects: 100% (9/9), 288.27 KiB | 6.27 MiB/s, done.
```

```
Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
```

```
To https://huggingface.co/lysandre/dummy
```

```
891b41d..b08aab1 main -> main
```

If we take a look at the model repository when this is finished, we can see all the recently added files:

















main

dummy

History: 3 commits

Add file

Lysandre First model version **b08aab1**

 .gitattributes	744.0B		First model version	10 minutes ago
 README.md	36.0B		Create README.md	2 days ago
 config.json	602.0B		First model version	10 minutes ago
 pytorch_mor	413.3MB		First model version	10 minutes ago
 special_toke	112.0B		First model version	10 minutes ago
 tokenizer.js	425.6KB		First model version	10 minutes ago
 tokenizer_co	284.0B		First model version	10 minutes ago
 vocab.txt	208.4KB		First model version	10 minutes ago

Building a model card



The model card is a file which is arguably as important as the model and tokenizer files in a model repository. It is the central definition of the model, ensuring reusability by fellow community members and reproducibility of results, and providing a platform on which other members may build their artifacts.

Documenting the training and evaluation process helps others understand what to expect of a model — and providing sufficient information regarding the data that was used and the preprocessing and postprocessing that were done ensures that the limitations, biases, and contexts in which the model is and is not useful can be identified and understood.

Therefore, creating a model card that clearly defines your model is a very important step. Here, we provide some tips that will help you with this. Creating the model card is done through the `README.md` file you saw earlier, which is a Markdown file.

The “model card” concept originates from a research direction from Google, first shared in the paper [“Model Cards for Model Reporting”](#) by Margaret Mitchell et al. A lot of information contained here is based on that paper, and we recommend you take a look at it to understand why model cards are so important in a world that values reproducibility, reusability, and fairness.

The model card usually starts with a very brief, high-level overview of what the model is for, followed by additional details in the following sections:

- Model description
- Intended uses & limitations
- How to use
- Limitations and bias
- Training data
- Training procedure
- Evaluation results

Let’s take a look at what each of these sections should contain.

Model description

The model description provides basic details about the model. This includes the architecture, version, if it was introduced in a paper, if an original implementation is available, the author, and general information about the model. Any copyright should be attributed here. General information about training procedures, parameters, and important disclaimers can also be mentioned in this section.

Intended uses & limitations

Here you describe the use cases the model is intended for, including the languages, fields, and domains where it can be applied. This section of the model card can also document areas that are known to be out of scope for the model, or where it is likely to perform suboptimally.

How to use

This section should include some examples of how to use the model. This can showcase usage of the `pipeline()` function, usage of the model and tokenizer classes, and any other code you think might be helpful.

Training data

This part should indicate which dataset(s) the model was trained on. A brief description of the dataset(s) is also welcome.

Training procedure

In this section you should describe all the relevant aspects of training that are useful from a reproducibility perspective. This includes any preprocessing and postprocessing that were done on the data, as well as details such as the number of epochs the model was trained for, the batch size, the learning rate, and so on.

Variable and metrics

Here you should describe the metrics you use for evaluation, and the different factors you are measuring. Mentioning which metric(s) were used, on which dataset and which dataset split, makes it easy to compare your model's performance compared to that of other models. These should be informed by the previous sections, such as the intended users and use cases.

Evaluation results

Finally, provide an indication of how well the model performs on the evaluation dataset. If the model uses a decision threshold, either provide the decision threshold used in the evaluation, or provide details on evaluation at different thresholds for the intended uses.

Example

Check out the following for a few examples of well-crafted model cards:

- [bert-base-cased](#)
- [gpt2](#)
- [distilbert](#)

More examples from different organizations and companies are available [here](#).

Note

Model cards are not a requirement when publishing models, and you don't need to include all of the sections described above when you make one. However, explicit documentation of the model can only benefit future users, so we recommend that you fill in as many of the sections as possible to the best of your knowledge and ability.

Model card metadata

If you have done a little exploring of the Hugging Face Hub, you should have seen that some models belong to certain categories: you can filter them by tasks, languages, libraries, and more. The categories a model belongs to are identified according to the metadata you add in the model card header.

For example, if you take a look at the [camembert-base model card](#), you should see the following lines in the model card header:

```
---  
  
language: fr  
  
license: mit  
  
datasets:  
- oscar  
  
---
```

This metadata is parsed by the Hugging Face Hub, which then identifies this model as being a French model, with an MIT license, trained on the Oscar dataset.

The [full model card specification](#) allows specifying languages, licenses, tags, datasets, metrics, as well as the evaluation results the model obtained when training.

The DataSets Library | Introduction

In [Chapter 3](#) you got your first taste of the 🤖 Datasets library and saw that there were three main steps when it came to fine-tuning a model:


1. Load a dataset from the Hugging Face Hub.
2. Preprocess the data with `Dataset.map()`.
3. Load and compute metrics.

But this is just scratching the surface of what 🤖 Datasets can do! In this chapter, we will take a deep dive into the library. Along the way, we'll find answers to the following questions:


- What do you do when your dataset is not on the Hub?
- How can you slice and dice a dataset? (And what if you *really* need to use Pandas?)
- What do you do when your dataset is huge and will melt your laptop's RAM?
- What the heck are “memory mapping” and Apache Arrow?
- How can you create your own dataset and push it to the Hub?

The techniques you learn here will prepare you for the advanced tokenization and fine-tuning tasks in [Chapter 6](#) and [Chapter 7](#) — so grab a coffee and let's get started!

What if my dataset isn't on the Hub?

You know how to use the [Hugging Face Hub](#) to download datasets, but you'll often find yourself working with data that is stored either on your laptop or on a remote server. In this section we'll show you how  Datasets can be used to load datasets that aren't available on the Hugging Face Hub.

Working with local and remote datasets

 Datasets provides loading scripts to handle the loading of local and remote datasets. It supports several common data formats, such as:

Data format	Loading script	Example
CSV & TSV	<code>csv</code>	<code>load_dataset("csv", data_files="my_file.csv")</code>
Text files	<code>text</code>	<code>load_dataset("text", data_files="my_file.txt")</code>
JSON & JSON Lines	<code>json</code>	<code>load_dataset("json", data_files="my_file.jsonl")</code>
Pickled DataFrames	<code>pandas</code>	<code>load_dataset("pandas", data_files="my_dataframe.pkl")</code>

As shown in the table, for each data format we just need to specify the type of loading script in the `load_dataset()` function, along with a `data_files` argument that specifies the path to one or more files. Let's start by loading a dataset from local files; later we'll see how to do the same with remote files.

Loading a local dataset

For this example we'll use the [SQuAD-it dataset](#), which is a large-scale dataset for question answering in Italian.

The training and test splits are hosted on GitHub, so we can download them with a simple `wget` command:

```
!wget https://github.com/crux82/squad-it/raw/master/SQuAD_it-train.json.gz  
  
!wget https://github.com/crux82/squad-it/raw/master/SQuAD_it-test.json.gz
```

This will download two compressed files called `SQuAD_it-train.json.gz` and `SQuAD_it-test.json.gz`, which we can decompress with the Linux `gzip` command:

```
!gzip -dkv SQuAD_it-*.json.gz
```

```
SQuAD_it-test.json.gz:      87.4% -- replaced with SQuAD_it-test.json  
SQuAD_it-train.json.gz:   82.2% -- replaced with SQuAD_it-train.json
```

We can see that the compressed files have been replaced with `SQuAD_it-train.json` and `SQuAD_it-test.json`, and that the data is stored in the JSON format.

🔪 If you're wondering why there's a `!` character in the above shell commands, that's because we're running them within a Jupyter notebook. Simply remove the prefix if you want to download and unzip the dataset within a terminal.

To load a JSON file with the `load_dataset()` function, we just need to know if we're dealing with ordinary JSON (similar to a nested dictionary) or JSON Lines (line-separated JSON). Like many question answering datasets, SQuAD-it uses the nested format, with all the text stored in a `data` field. This means we can load the dataset by specifying the `field` argument as follows:


```
from datasets import load_dataset
```

```
squad_it_dataset = load_dataset("json", data_files="SQuAD_it-train.json",  
field="data")
```

By default, loading local files creates a `DatasetDict` object with a `train` split. We can see this by inspecting the `squad_it_dataset` object:

```
squad_it_dataset
```

```
DatasetDict({  
  
  train: Dataset({  
  
    features: ['title', 'paragraphs'],  
  
    num_rows: 442  
  
  })  
  
})
```

This shows us the number of rows and the column names associated with the training set. We can view one of the examples by indexing into the `train` split as follows:

```
squad_it_dataset["train"][0]
```

```

{
  "title": "Terremoto del Sichuan del 2008",
  "paragraphs": [
    {
      "context": "Il terremoto del Sichuan del 2008 o il terremoto...",
      "qas": [
        {
          "answers": [{"answer_start": 29, "text": "2008"}],
          "id": "56cdca7862d2951400fa6826",
          "question": "In quale anno si è verificato il terremoto nel
Sichuan?",
        },
        ...
      ],
    },
    ...
  ],
}

```

Great, we've loaded our first local dataset! But while this worked for the training set, what we really want is to include both the `train` and `test` splits in a single `DatasetDict` object so we can

apply `Dataset.map()` functions across both splits at once. To do this, we can provide a dictionary to the `data_files` argument that maps each split name to a file associated with that split:

```
data_files = {"train": "SQuAD_it-train.json", "test": "SQuAD_it-test.json"}

squad_it_dataset = load_dataset("json", data_files=data_files, field="data")

squad_it_dataset
```

```
DatasetDict({
  train: Dataset({
    features: ['title', 'paragraphs'],
    num_rows: 442
  })
  test: Dataset({
    features: ['title', 'paragraphs'],
    num_rows: 48
  })
})
```

This is exactly what we wanted. Now, we can apply various preprocessing techniques to clean up the data, tokenize the reviews, and so on.

The `data_files` argument of the `load_dataset()` function is quite flexible and can be either a single file path, a list of file paths, or a dictionary that maps split names to file paths. You can also glob files that match a specified pattern according to the rules used by the Unix shell (e.g., you can glob all the

JSON files in a directory as a single split by setting `data_files="*.json"`). See the 😊 Datasets [documentation](#) for more details.

The loading scripts in 🤖 Datasets actually support automatic decompression of the input files, so we could have skipped the use of `gzip` by pointing the `data_files` argument directly to the compressed files:

```
data_files = {"train": "SQuAD_it-train.json.gz", "test": "SQuAD_it-test.json.gz"}  
  
squad_it_dataset = load_dataset("json", data_files=data_files, field="data")
```

This can be useful if you don't want to manually decompress many GZIP files. The automatic decompression also applies to other common formats like ZIP and TAR, so you just need to point `data_files` to the compressed files and you're good to go!

Now that you know how to load local files on your laptop or desktop, let's take a look at loading remote files.


Loading a remote dataset

If you're working as a data scientist or coder in a company, there's a good chance the datasets you want to analyze are stored on some remote server. Fortunately, loading remote files is just as simple as loading local ones! Instead of providing a path to local files, we point the `data_files` argument of `load_dataset()` to one or more URLs where the remote files are stored. For example, for the SQuAD-it dataset hosted on GitHub, we can just point `data_files` to the `SQuAD_it-*.json.gz` URLs as follows:


```
url = "https://github.com/crux82/squad-it/raw/master/"  
  
data_files = {  
  
    "train": url + "SQuAD_it-train.json.gz",  
  
    "test": url + "SQuAD_it-test.json.gz",  
  
}
```

```
squad_it_dataset = load_dataset("json", data_files=data_files, field="data")
```


This returns the same `DatasetDict` object obtained above, but saves us the step of manually downloading and decompressing the `SQuAD_it-*.json.gz` files. This wraps up our foray into the various ways to load datasets that aren't hosted on the Hugging Face Hub. Now that we've got a dataset to play with, let's get our hands dirty with various data-wrangling techniques!

 **Try it out!** Pick another dataset hosted on GitHub or the [UCI Machine Learning Repository](#) and try loading it both locally and remotely using the techniques introduced above. For bonus points, try loading a dataset that's stored in a CSV or text format (see the [documentation](#) for more information on these formats).

Time to slice and dice

Most of the time, the data you work with won't be perfectly prepared for training models. In this section we'll explore the various features that  Datasets provides to clean up your datasets.

Slicing and dicing our data

Similar to Pandas,  Datasets provides several functions to manipulate the contents of `Dataset` and `DatasetDict` objects. We already encountered the `Dataset.map()` method in [Chapter 3](#), and in this section we'll explore some of the other functions at our disposal.

For this example we'll use the [Drug Review Dataset](#) that's hosted on the [UC Irvine Machine Learning Repository](#), which contains patient reviews on various drugs, along with the condition being treated and a 10-star rating of the patient's satisfaction.

First we need to download and extract the data, which can be done with the `wget` and `unzip` commands:

```
!wget "https://archive.ics.uci.edu/ml/machine-learning-databases/00462/drugsCom_raw.zip"
```

```
!unzip drugsCom_raw.zip
```

Since TSV is just a variant of CSV that uses tabs instead of commas as the separator, we can load these files by using the `csv` loading script and specifying the `delimiter` argument in the `load_dataset()` function as follows:

```
from datasets import load_dataset
```

```
data_files = {"train": "drugsComTrain_raw.tsv", "test": "drugsComTest_raw.tsv"}
```

```
# \t is the tab character in Python
```

```
drug_dataset = load_dataset("csv", data_files=data_files, delimiter="\t")
```

A good practice when doing any sort of data analysis is to grab a small random sample to get a quick feel for the type of data you're working with. In [Dask Datasets](#), we can create a random sample by chaining the `Dataset.shuffle()` and `Dataset.select()` functions together:

```
drug_sample = drug_dataset["train"].shuffle(seed=42).select(range(1000))
```

```
# Peek at the first few examples
```

```
drug_sample[:3]
```

```
{'Unnamed: 0': [87571, 178045, 80482],
```

```
 'drugName': ['Naproxen', 'Duloxetine', 'Mobic'],
```

```
 'condition': ['Gout, Acute', 'fibromyalgia', 'Inflammatory Conditions'],
```

```
 'review': ['"like the previous person mention, I;m a strong believer of aleve, it works faster for my gout than the prescription meds I take. No more going to the doctor for refills.....Aleve works!"',
```

```
  '"I have taken Cymbalta for about a year and a half for fibromyalgia pain. It is great\r\nas a pain reducer and an anti-depressant, however, the side effects outweighed \r\nany benefit I got from it. I had trouble with restlessness, being tired constantly,\r\nndizziness, dry mouth, numbness and tingling in my feet, and horrible sweating. I am\r\nbeing weaned off of it now. Went from 60 mg to 30mg and now to 15 mg. I will be\r\noff completely in about a week. The fibro pain is coming back, but I would rather deal with it than the side effects."',
```


```
  '"I have been taking Mobic for over a year with no side effects other than an elevated blood pressure. I had severe knee and ankle pain which completely went away after taking Mobic. I attempted to stop the medication however pain returned after a few days."'],
```

```
 'rating': [9.0, 3.0, 10.0],
```

```
'date': ['September 2, 2015', 'November 7, 2011', 'June 5, 2013'],  
'usefulCount': [36, 13, 128]}
```

Note that we've fixed the seed in `Dataset.shuffle()` for reproducibility purposes. `Dataset.select()` expects an iterable of indices, so we've passed `range(1000)` to grab the first 1,000 examples from the shuffled dataset. From this sample we can already see a few quirks in our dataset:

- The `Unnamed: 0` column looks suspiciously like an anonymized ID for each patient.
- The `condition` column includes a mix of uppercase and lowercase labels.
- The reviews are of varying length and contain a mix of Python line separators (`\r\n`) as well as HTML character codes like `&\#039;`.

Let's see how we can use  Datasets to deal with each of these issues. To test the patient ID hypothesis for the `Unnamed: 0` column, we can use the `Dataset.unique()` function to verify that the number of IDs matches the number of rows in each split:


```
for split in drug_dataset.keys():  
  
    assert len(drug_dataset[split]) == len(drug_dataset[split].unique("Unnamed: 0"))
```

This seems to confirm our hypothesis, so let's clean up the dataset a bit by renaming the `Unnamed: 0` column to something a bit more interpretable. We can use the `DatasetDict.rename_column()` function to rename the column across both splits in one go:

```
drug_dataset = drug_dataset.rename_column(  
  
    original_column_name="Unnamed: 0", new_column_name="patient_id"  
  
)  
  
drug_dataset
```



```
DatasetDict({
  train: Dataset({
    features: ['patient_id', 'drugName', 'condition', 'review', 'rating', 'date',
              'usefulCount'],
    num_rows: 161297
  })
  test: Dataset({
    features: ['patient_id', 'drugName', 'condition', 'review', 'rating', 'date',
              'usefulCount'],
    num_rows: 53766
  })
})
```

 **Try it out!** Use the `Dataset.unique()` function to find the number of unique drugs and conditions in the training and test sets.

Next, let's normalize all the `condition` labels using `Dataset.map()`. As we did with tokenization in [Chapter 3](#), we can define a simple function that can be applied across all the rows of each split in `drug_dataset`:

```
def lowercase_condition(example):
    return {"condition": example["condition"].lower()}

drug_dataset.map(lowercase_condition)
```

```
AttributeError: 'NoneType' object has no attribute 'lower'
```

Oh no, we've run into a problem with our map function! From the error we can infer that some of the entries in the `condition` column are `None`, which cannot be lowercased as they're not strings. Let's drop these rows using `Dataset.filter()`, which works in a similar way to `Dataset.map()` and expects a function that receives a single example of the dataset. Instead of writing an explicit function like:

```
def filter_nones(x):  
  
    return x["condition"] is not None
```

and then running `drug_dataset.filter(filter_nones)`, we can do this in one line using a *lambda function*. In Python, lambda functions are small functions that you can define without explicitly naming them. They take the general form:

```
lambda <arguments> : <expression>
```

where `lambda` is one of Python's special [keywords](#), `<arguments>` is a list/set of comma-separated values that define the inputs to the function, and `<expression>` represents the operations you wish to execute. For example, we can define a simple lambda function that squares a number as follows:

```
lambda x : x * x
```

To apply this function to an input, we need to wrap it and the input in parentheses:


```
(lambda x: x * x)(3)
```

```
9
```

Similarly, we can define lambda functions with multiple arguments by separating them with commas. For example, we can compute the area of a triangle as follows:

```
(lambda base, height: 0.5 * base * height)(4, 8)
```

```
16.0
```

Lambda functions are handy when you want to define small, single-use functions (for more information about them, we recommend reading the excellent [Real Python tutorial](#) by Andre Burgaud). In the  Datasets context, we can use lambda functions to define simple map and filter operations, so let's use this trick to eliminate the `None` entries in our dataset:

```
drug_dataset = drug_dataset.filter(lambda x: x["condition"] is not None)
```

With the `None` entries removed, we can normalize our `condition` column:

```
drug_dataset = drug_dataset.map(lowercase_condition)
```

```
# Check that lowercasing worked
```

```
drug_dataset["train"]["condition"][:3]
```

```
['left ventricular dysfunction', 'adhd', 'birth control']
```

It works! Now that we've cleaned up the labels, let's take a look at cleaning up the reviews themselves.

Creating new columns

Whenever you're dealing with customer reviews, a good practice is to check the number of words in each review. A review might be just a single word like "Great!" or a full-blown essay with thousands of words, and depending on the use case you'll need to handle these extremes differently. To compute the number of words in each review, we'll use a rough heuristic based on splitting each text by whitespace.

Let's define a simple function that counts the number of words in each review:

```
def compute_review_length(example):  
  
    return {"review_length": len(example["review"].split())}
```

Unlike our `lowercase_condition()` function, `compute_review_length()` returns a dictionary whose key does not correspond to one of the column names in the dataset. In this case, when `compute_review_length()` is passed to `Dataset.map()`, it will be applied to all the rows in the dataset to create a new `review_length` column:

```
drug_dataset = drug_dataset.map(compute_review_length)  
  
# Inspect the first training example  
  
drug_dataset["train"][0]
```

```
{'patient_id': 206461,  
  
  'drugName': 'Valsartan',  
  
  'condition': 'left ventricular dysfunction',  
  
  'review': '"It has no side effect, I take it in combination of Bystolic 5 Mg and Fish Oil"',  
  
  'rating': 9.0,  
  
  'date': 'May 20, 2012',  
  
  'usefulCount': 27,  
  
  'review_length': 17}
```


As expected, we can see a `review_length` column has been added to our training set. We can sort this new column with `Dataset.sort()` to see what the extreme values look like:

```
drug_dataset["train"].sort("review_length")[:3]
```

```
{'patient_id': [103488, 23627, 20558],  
  
  'drugName': ['Loestrin 21 1 / 20', 'Chlorzoxazone', 'Nucynta'],  
  
  'condition': ['birth control', 'muscle spasm', 'pain'],  
  
  'review': ['"Excellent."', '"useless"', '"ok"'],  
  
  'rating': [10.0, 1.0, 6.0],  
  
  'date': ['November 4, 2008', 'March 24, 2017', 'August 20, 2016'],
```

```
'usefulCount': [5, 2, 10],  
  
'review_length': [1, 1, 1]}
```

As we suspected, some reviews contain just a single word, which, although it may be okay for sentiment analysis, would not be informative if we want to predict the condition.


 An alternative way to add new columns to a dataset is with the `Dataset.add_column()` function. This allows you to provide the column as a Python list or NumPy array and can be handy in situations where `Dataset.map()` is not well suited for your analysis.

Let's use the `Dataset.filter()` function to remove reviews that contain fewer than 30 words. Similarly to what we did with the `condition` column, we can filter out the very short reviews by requiring that the reviews have a length above this threshold:

```
drug_dataset = drug_dataset.filter(lambda x: x["review_length"] > 30)  
  
print(drug_dataset.num_rows)
```

```
{'train': 138514, 'test': 46108}
```

As you can see, this has removed around 15% of the reviews from our original training and test sets.

 **Try it out!** Use the `Dataset.sort()` function to inspect the reviews with the largest numbers of words. See the [documentation](#) to see which argument you need to use sort the reviews by length in descending order.

The last thing we need to deal with is the presence of HTML character codes in our reviews. We can use Python's `html` module to unescape these characters, like so:

```
import html
```

```
text = "I&#039;m a transformer called BERT"
```

```
html.unescape(text)
```

```
"I'm a transformer called BERT"
```

We'll use `Dataset.map()` to unescape all the HTML characters in our corpus:

```
drug_dataset = drug_dataset.map(lambda x: {"review": html.unescape(x["review"])})
```

As you can see, the `Dataset.map()` method is quite useful for processing data — and we haven't even scratched the surface of everything it can do!

The `map()` method's superpowers

The `Dataset.map()` method takes a `batched` argument that, if set to `True`, causes it to send a batch of examples to the map function at once (the batch size is configurable but defaults to 1,000). For instance, the previous map function that unescaped all the HTML took a bit of time to run (you can read the time taken from the progress bars). We can speed this up by processing several elements at the same time using a list comprehension.

When you specify `batched=True` the function receives a dictionary with the fields of the dataset, but each value is now a *list of values*, and not just a single value. The return value of `Dataset.map()` should be the same: a dictionary with the fields we want to update or add to our dataset, and a list of values. For example, here is another way to unescape all HTML characters, but using `batched=True`:

```
new_drug_dataset = drug_dataset.map(  
    lambda x: {"review": [html.unescape(o) for o in x["review"]]}, batched=True  
)
```

If you're running this code in a notebook, you'll see that this command executes way faster than the previous one. And it's not because our reviews have already been HTML-unesaped — if you re-execute the instruction from the previous section (without `batched=True`), it will take the same amount of time as before. This is because list comprehensions are usually faster than executing the same code in a `for` loop, and we also gain some performance by accessing lots of elements at the same time instead of one by one.

Using `Dataset.map()` with `batched=True` will be essential to unlock the speed of the “fast” tokenizers that we'll encounter in [Chapter 6](#), which can quickly tokenize big lists of texts. For instance, to tokenize all the drug reviews with a fast tokenizer, we could use a function like this:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")


def tokenize_function(examples):

    return tokenizer(examples["review"], truncation=True)
```

As you saw in [Chapter 3](#), we can pass one or several examples to the tokenizer, so we can use this function with or without `batched=True`. Let's take this opportunity to compare the performance of the different options. In a notebook, you can time a one-line instruction by adding `%time` before the line of code you wish to measure:

```
%time tokenized_dataset = drug_dataset.map(tokenize_function, batched=True)
```

You can also time a whole cell by putting `%%time` at the beginning of the cell. On the hardware we executed this on, it showed 10.8s for this instruction (it's the number written after “Wall time”).

 **Try it out!** Execute the same instruction with and without `batched=True`, then try it with a slow tokenizer (add `use_fast=False` in the `AutoTokenizer.from_pretrained()` method) so you can see what numbers you get on your hardware.

Here are the results we obtained with and without batching, with a fast and a slow tokenizer:

Options	Fast tokenizer	Slow tokenizer
<code>batched=True</code>	10.8s	4min41s
<code>batched=False</code>	59.2s	5min3s

This means that using a fast tokenizer with the `batched=True` option is 30 times faster than its slow counterpart with no batching — this is truly amazing! That’s the main reason why fast tokenizers are the default when using `AutoTokenizer` (and why they are called “fast”). They’re able to achieve such a speedup because behind the scenes the tokenization code is executed in Rust, which is a language that makes it easy to parallelize code execution.

Parallelization is also the reason for the nearly 6x speedup the fast tokenizer achieves with batching: you can’t parallelize a single tokenization operation, but when you want to tokenize lots of texts at the same time you can just split the execution across several processes, each responsible for its own texts.

`Dataset.map()` also has some parallelization capabilities of its own. Since they are not backed by Rust, they won’t let a slow tokenizer catch up with a fast one, but they can still be helpful (especially if you’re using a tokenizer that doesn’t have a fast version). To enable multiprocessing, use the `num_proc` argument and specify the number of processes to use in your call to `Dataset.map()`:

```
slow_tokenizer = AutoTokenizer.from_pretrained("bert-base-cased", use_fast=False)
```

```
def slow_tokenize_function(examples):
```

```
    return slow_tokenizer(examples["review"], truncation=True)
```

```
tokenized_dataset = drug_dataset.map(slow_tokenize_function, batched=True,  
num_proc=8)
```

You can experiment a little with timing to determine the optimal number of processes to use; in our case 8 seemed to produce the best speed gain. Here are the numbers we got with and without multiprocessing:

Options	Fast tokenizer	Slow tokenizer
<code>batched=True</code>	10.8s	4min41s
<code>batched=False</code>	59.2s	5min3s
<code>batched=True, num_proc=8</code>	6.52s	41.3s
<code>batched=False, num_proc=8</code>	9.49s	45.2s

Those are much more reasonable results for the slow tokenizer, but the performance of the fast tokenizer was also substantially improved. Note, however, that won't always be the case — for values of `num_proc` other than 8, our tests showed that it was faster to use `batched=True` without that

option. In general, we don't recommend using Python multiprocessing for fast tokenizers with `batched=True`.

Using `num_proc` to speed up your processing is usually a great idea, as long as the function you are using is not already doing some kind of multiprocessing of its own.

All of this functionality condensed into a single method is already pretty amazing, but there's more! With `Dataset.map()` and `batched=True` you can change the number of elements in your dataset. This is super useful in many situations where you want to create several training features from one example, and we will need to do this as part of the preprocessing for several of the NLP tasks we'll undertake in [Chapter 7](#).

💡 In machine learning, an *example* is usually defined as the set of *features* that we feed to the model. In some contexts, these features will be the set of columns in a `Dataset`, but in others (like here and for question answering), multiple features can be extracted from a single example and belong to a single column.

Let's have a look at how it works! Here we will tokenize our examples and truncate them to a maximum length of 128, but we will ask the tokenizer to return *all* the chunks of the texts instead of just the first one. This can be done with `return_overflowing_tokens=True`:

```
def tokenize_and_split(examples):  
  
    return tokenizer(  
  
        examples["review"],  
  
        truncation=True,  
  
        max_length=128,  
  
        return_overflowing_tokens=True,  
  
    )
```

Let's test this on one example before using `Dataset.map()` on the whole dataset:

```
result = tokenize_and_split(drug_dataset["train"][0])

[len(inp) for inp in result["input_ids"]]
```

```
[128, 49]
```

So, our first example in the training set became two features because it was tokenized to more than the maximum number of tokens we specified: the first one of length 128 and the second one of length 49. Now let's do this for all elements of the dataset!

```
tokenized_dataset = drug_dataset.map(tokenize_and_split, batched=True)
```

```
ArrowInvalid: Column 1 named condition expected length 1463 but got length 1000
```

Oh no! That didn't work! Why not? Looking at the error message will give us a clue: there is a mismatch in the lengths of one of the columns, one being of length 1,463 and the other of length 1,000. If you've looked at the `Dataset.map()` [documentation](#), you may recall that it's the number of samples passed to the function that we are mapping; here those 1,000 examples gave 1,463 new features, resulting in a shape error.

The problem is that we're trying to mix two different datasets of different sizes: the `drug_dataset` columns will have a certain number of examples (the 1,000 in our error), but the `tokenized_dataset` we are building will have more (the 1,463 in the error message; it is more than 1,000 because we are tokenizing long reviews into more than one example by using `return_overflowing_tokens=True`). That doesn't work for a `Dataset`, so we need to either remove the columns from the old dataset or make them the same size as they are in the new dataset. We can do the former with the `remove_columns` argument:

```
tokenized_dataset = drug_dataset.map(
```

```
    tokenize_and_split, batched=True,  
    remove_columns=drug_dataset["train"].column_names  
)
```

Now this works without error. We can check that our new dataset has many more elements than the original dataset by comparing the lengths:

```
len(tokenized_dataset["train"]), len(drug_dataset["train"])
```

```
(206772, 138514)
```

We mentioned that we can also deal with the mismatched length problem by making the old columns the same size as the new ones. To do this, we will need the `overflow_to_sample_mapping` field the tokenizer returns when we set `return_overflowing_tokens=True`. It gives us a mapping from a new feature index to the index of the sample it originated from. Using this, we can associate each key present in our original dataset with a list of values of the right size by repeating the values of each example as many times as it generates new features:

```
def tokenize_and_split(examples):  
  
    result = tokenizer(  
  
        examples["review"],  
  
        truncation=True,  
  
        max_length=128,  
  
        return_overflowing_tokens=True,  
  
    )
```

```

# Extract mapping between new and old indices

sample_map = result.pop("overflow_to_sample_mapping")

for key, values in examples.items():

    result[key] = [values[i] for i in sample_map]

return result

```

We can see it works with `Dataset.map()` without us needing to remove the old columns:

```

tokenized_dataset = drug_dataset.map(tokenize_and_split, batched=True)

tokenized_dataset

```

```

DatasetDict({

  train: Dataset({

    features: ['attention_mask', 'condition', 'date', 'drugName', 'input_ids',
'patient_id', 'rating', 'review', 'review_length', 'token_type_ids', 'usefulCount'],

    num_rows: 206772

  })

  test: Dataset({

    features: ['attention_mask', 'condition', 'date', 'drugName', 'input_ids',
'patient_id', 'rating', 'review', 'review_length', 'token_type_ids', 'usefulCount'],




    num_rows: 68876

  })


```

```
} )
```

We get the same number of training features as before, but here we've kept all the old fields. If you need them for some post-processing after applying your model, you might want to use this approach.

You've now seen how  Datasets can be used to preprocess a dataset in various ways. Although the processing functions of  Datasets will cover most of your model training needs, there may be times when you'll need to switch to Pandas to access more powerful features, like `DataFrame.groupby()` or high-level APIs for visualization. Fortunately,  Datasets is designed to be interoperable with libraries such as Pandas, NumPy, PyTorch, TensorFlow, and JAX. Let's take a look at how this works.

From Datasets to DataFrames and back

To enable the conversion between various third-party libraries,  Datasets provides a `Dataset.set_format()` function. This function only changes the *output format* of the dataset, so you can easily switch to another format without affecting the underlying *data format*, which is Apache Arrow. The formatting is done in place. To demonstrate, let's convert our dataset to Pandas:

```
drug_dataset.set_format("pandas")
```

Now when we access elements of the dataset we get a `pandas.DataFrame` instead of a dictionary:


```
drug_dataset["train"][:3]
```

	patient_id	drugName	condition	review	rating	date	usefulCount	review
0	95260	Guanfacine	adhd	"My son is halfway through his fourth week of Intuniv..."	8.0	April 27, 2010	192	141

	patient_id	drugName	condition	review	rating	date	usefulCount	review
1	92703	Lybrel	birth control	"I used to take another oral contraceptive, which had 21 pill cycle, and was very happy- very light periods, max 5 days, no other side effects..."	5.0	December 14, 2009	17	134
2	138000	Ortho Evra	birth control	"This is my first time using any form of birth control..."	8.0	November 3, 2015	10	89

Let's create a `pandas.DataFrame` for the whole training set by selecting all the elements of `drug_dataset["train"]`:

```
train_df = drug_dataset["train"][:]
```

 Under the hood, `Dataset.set_format()` changes the return format for the dataset's `__getitem__()` dunder method. This means that when we want to create a new object like `train_df` from a `Dataset` in the "pandas" format, we need to slice the whole dataset to obtain a `pandas.DataFrame`. You can verify for yourself that the type of `drug_dataset["train"]` is `Dataset`, irrespective of the output format.

From here we can use all the Pandas functionality that we want. For example, we can do fancy chaining to compute the class distribution among the `condition` entries:

```
frequencies = (
    train_df["condition"]
```



```
.value_counts()

.to_frame()

.reset_index()

.rename(columns={"index": "condition", "condition": "frequency"})

)

frequencies.head()
```

	condition	frequency
0	birth control	27655
1	depression	8023
2	acne	5209
3	anxiety	4991
4	pain	4744

And once we're done with our Pandas analysis, we can always create a new `Dataset` object by using the `Dataset.from_pandas()` function as follows:


```
from datasets import Dataset
```

```
freq_dataset = Dataset.from_pandas(frequencies)
```

```
freq_dataset
```

```
Dataset({  
  features: ['condition', 'frequency'],  
  num_rows: 819  
})
```

 **Try it out!** Compute the average rating per drug and store the result in a new `Dataset`.

This wraps up our tour of the various preprocessing techniques available in  `Datasets`. To round out the section, let's create a validation set to prepare the dataset for training a classifier on. Before doing so, we'll reset the output format of `drug_dataset` from "pandas" to "arrow":

```
drug_dataset.reset_format()
```

Creating a validation set

Although we have a test set we could use for evaluation, it's a good practice to leave the test set untouched and create a separate validation set during development. Once you are happy with the performance of your models on the validation set, you can do a final sanity check on the test set. This process helps mitigate the risk that you'll overfit to the test set and deploy a model that fails on real-world data.

🤖 Datasets provides a `Dataset.train_test_split()` function that is based on the famous functionality from `scikit-learn`. Let's use it to split our training set into `train` and `validation` splits (we set the `seed` argument for reproducibility):

```
drug_dataset_clean = drug_dataset["train"].train_test_split(train_size=0.8, seed=42)

# Rename the default "test" split to "validation"

drug_dataset_clean["validation"] = drug_dataset_clean.pop("test")

# Add the "test" set to our `DatasetDict`

drug_dataset_clean["test"] = drug_dataset["test"]



drug_dataset_clean
```

```
DatasetDict({
  train: Dataset({
    features: ['patient_id', 'drugName', 'condition', 'review', 'rating', 'date',
              'usefulCount', 'review_length', 'review_clean'],
    num_rows: 110811
  })
  validation: Dataset({
    features: ['patient_id', 'drugName', 'condition', 'review', 'rating', 'date',
              'usefulCount', 'review_length', 'review_clean'],
    num_rows: 27703
  })
})
```

```
test: Dataset({
    features: ['patient_id', 'drugName', 'condition', 'review', 'rating', 'date',
'usefulCount', 'review_length', 'review_clean'],
    num_rows: 46108
})
})
```

Great, we've now prepared a dataset that's ready for training some models on! In [section 5](#) we'll show you how to upload datasets to the Hugging Face Hub, but for now let's cap off our analysis by looking at a few ways you can save datasets on your local machine.

Saving a dataset

Although  Datasets will cache every downloaded dataset and the operations performed on it, there are times when you'll want to save a dataset to disk (e.g., in case the cache gets deleted). As shown in the table below,  Datasets provides three main functions to save your dataset in different formats:

Data format	Function
Arrow	<code>Dataset.save_to_disk()</code>
CSV	<code>Dataset.to_csv()</code>
JSON	<code>Dataset.to_json()</code>

For example, let's save our cleaned dataset in the Arrow format:

```
drug_dataset_clean.save_to_disk("drug-reviews")
```

This will create a directory with the following structure:

```
drug-reviews/  
├─ dataset_dict.json  
├─ test  
│   ├─ dataset.arrow  
│   ├─ dataset_info.json  
│   └─ state.json  
├─ train  
│   ├─ dataset.arrow  
│   ├─ dataset_info.json  
│   ├─ indices.arrow  
│   └─ state.json  
└─ validation  
    ├─ dataset.arrow  
    ├─ dataset_info.json  
    ├─ indices.arrow  
    └─ state.json
```

where we can see that each split is associated with its own `dataset.arrow` table, and some metadata in `dataset_info.json` and `state.json`. You can think of the Arrow format as a fancy table of columns and rows that is optimized for building high-performance applications that process and transport large datasets.

Once the dataset is saved, we can load it by using the `load_from_disk()` function as follows:

```
from datasets import load_from_disk
```

```
drug_dataset_reloaded = load_from_disk("drug-reviews")
```

```
drug_dataset_reloaded
```

```
DatasetDict({
  train: Dataset({
    features: ['patient_id', 'drugName', 'condition', 'review', 'rating', 'date',
              'usefulCount', 'review_length'],
    num_rows: 110811
  })
  validation: Dataset({
    features: ['patient_id', 'drugName', 'condition', 'review', 'rating', 'date',
              'usefulCount', 'review_length'],
    num_rows: 27703
  })
  test: Dataset({
```

```
features: ['patient_id', 'drugName', 'condition', 'review', 'rating', 'date',
'usefulCount', 'review_length'],

num_rows: 46108

})

})
```

For the CSV and JSON formats, we have to store each split as a separate file. One way to do this is by iterating over the keys and values in the `DatasetDict` object:

```
for split, dataset in drug_dataset_clean.items():

    dataset.to_json(f"drug-reviews-{split}.jsonl")
```

This saves each split in [JSON Lines format](#), where each row in the dataset is stored as a single line of JSON. Here's what the first example looks like:

```
!head -n 1 drug-reviews-train.jsonl
```

```
{"patient_id":141780,"drugName":"Escitalopram","condition":"depression","review":"\nI
seemed to experience the regular side effects of LEXAPRO, insomnia, low sex drive,
sleepiness during the day. I am taking it at night because my doctor said if it made
me tired to take it at night. I assumed it would and started out taking it at night.
Strange dreams, some pleasant. I was diagnosed with fibromyalgia. Seems to be helping
with the pain. Have had anxiety and depression in my family, and have tried quite a
few other medications that haven't worked. Only have been on it for two weeks but
feel more positive in my mind, want to accomplish more in my life. Hopefully the side
effects will dwindle away, worth it to stick with it from hearing others responses.
Great medication.\n","rating":9.0,"date":"May 29,
2011","usefulCount":10,"review_length":125}
```

We can then use the techniques from [section 2](#) to load the JSON files as follows:

```
data_files = {  
    "train": "drug-reviews-train.jsonl",  
    "validation": "drug-reviews-validation.jsonl",  
    "test": "drug-reviews-test.jsonl",  
}  
  
drug_dataset_reloaded = load_dataset("json", data_files=data_files)
```

And that's it for our excursion into data wrangling with 🤖 Datasets! Now that we have a cleaned dataset for training a model on, here are a few ideas that you could try out:

1. Use the techniques from [Chapter 3](#) to train a classifier that can predict the patient condition based on the drug review.
2. Use the `summarization` pipeline from [Chapter 1](#) to generate summaries of the reviews.

Next, we'll take a look at how 🤖 Datasets can enable you to work with huge datasets without blowing up your laptop!

ig data? 😊 Datasets to the rescue!

Nowadays it is not uncommon to find yourself working with multi-gigabyte datasets, especially if you're planning to pretrain a transformer like BERT or GPT-2 from scratch. In these cases, even *loading* the data can be a challenge. For example, the WebText corpus used to pretrain GPT-2 consists of over 8 million documents and 40 GB of text — loading this into your laptop's RAM is likely to give it a heart attack!

Fortunately, 🤖 Datasets has been designed to overcome these limitations. It frees you from memory management problems by treating datasets as *memory-mapped* files, and from hard drive limits by *streaming* the entries in a corpus.

In this section we'll explore these features of 🤖 Datasets with a huge 825 GB corpus known as [the Pile](#). Let's get started!

What is the Pile?

The Pile is an English text corpus that was created by [EleutherAI](#) for training large-scale language models. It includes a diverse range of datasets, spanning scientific articles, GitHub code repositories, and filtered web text. The training corpus is available in [14 GB chunks](#), and you can also download several of the [individual components](#). Let's start by taking a look at the PubMed Abstracts dataset, which is a corpus of abstracts from 15 million biomedical publications on [PubMed](#). The dataset is in [JSON Lines format](#) and is compressed using the `zstandard` library, so first we need to install that:

```
!pip install zstandard
```

Next, we can load the dataset using the method for remote files that we learned in [section 2](#):

```
from datasets import load_dataset
```

```
# This takes a few minutes to run, so go grab a tea or coffee while you wait :)
```

```
data_files = "https://the-eye.eu/public/AI/pile_preliminary_components/PUBMED_title_abstracts_2019_baseline.jsonl.zst"

pubmed_dataset = load_dataset("json", data_files=data_files, split="train")

pubmed_dataset
```

```
Dataset({
  features: ['meta', 'text'],
  num_rows: 15518009
})
```

We can see that there are 15,518,009 rows and 2 columns in our dataset — that's a lot!

🔪 By default, 😊 Datasets will decompress the files needed to load a dataset. If you want to preserve hard drive space, you can pass `DownloadConfig(delete_extracted=True)` to the `download_config` argument of `load_dataset()`. See the [documentation](#) for more details.

Let's inspect the contents of the first example:

```
pubmed_dataset[0]
```

```
{'meta': {'pmid': 11409574, 'language': 'eng'},
 'text': 'Epidemiology of hypoxaemia in children with acute lower respiratory infection.\n\nTo determine the prevalence of hypoxaemia in children aged under 5 years suffering acute lower respiratory infections (ALRI), the risk factors for hypoxaemia
```

```
in children under 5 years of age with ALRI, and the association of hypoxaemia with an increased risk of dying in children of the same age ...'}]
```

Okay, this looks like the abstract from a medical article. Now let's see how much RAM we've used to load the dataset!

The magic of memory mapping

A simple way to measure memory usage in Python is with the [psutil](#) library, which can be installed with `pip` as follows:

```
!pip install psutil
```

It provides a `Process` class that allows us to check the memory usage of the current process as follows:

```
import psutil

# Process.memory_info is expressed in bytes, so convert to megabytes

print(f"RAM used: {psutil.Process().memory_info().rss / (1024 * 1024):.2f} MB")
```

```
RAM used: 5678.33 MB
```

Here the `rss` attribute refers to the *resident set size*, which is the fraction of memory that a process occupies in RAM. This measurement also includes the memory used by the Python interpreter and the libraries we've loaded, so the actual amount of memory used to load the dataset is a bit smaller. For comparison, let's see how large the dataset is on disk, using the `dataset_size` attribute. Since the result is expressed in bytes like before, we need to manually convert it to gigabytes:

```
print(f"Number of files in dataset : {pubmed_dataset.dataset_size}")



size_gb = pubmed_dataset.dataset_size / (1024**3)



print(f"Dataset size (cache file) : {size_gb:.2f} GB")
```

```
Number of files in dataset : 20979437051
```

```
Dataset size (cache file) : 19.54 GB
```

Nice — despite it being almost 20 GB large, we're able to load and access the dataset with much less RAM!

 **Try it out!** Pick one of the [subsets](#) from the Pile that is larger than your laptop or desktop's RAM, load it with  Datasets, and measure the amount of RAM used. Note that to get an accurate measurement, you'll want to do this in a new process. You can find the decompressed sizes of each subset in Table 1 of [the Pile paper](#).

If you're familiar with Pandas, this result might come as a surprise because of Wes Kinney's famous [rule of thumb](#) that you typically need 5 to 10 times as much RAM as the size of your dataset. So how does  Datasets solve this memory management problem?  Datasets treats each dataset as a [memory-mapped file](#), which provides a mapping between RAM and filesystem storage that allows the library to access and operate on elements of the dataset without needing to fully load it into memory.

Memory-mapped files can also be shared across multiple processes, which enables methods like `Dataset.map()` to be parallelized without needing to move or copy the dataset. Under the hood, these capabilities are all realized by the [Apache Arrow](#) memory format and [pyarrow](#) library, which make the data loading and processing lightning fast. (For more details about Apache Arrow and comparisons to Pandas, check out [Dejan Simic's blog post](#).) To see this in action, let's run a little speed test by iterating over all the elements in the PubMed Abstracts dataset:

```
import timeit
```

```
code_snippet = """batch_size = 1000
```

```

for idx in range(0, len(pubmed_dataset), batch_size):

    _ = pubmed_dataset[idx:idx + batch_size]

"""

time = timeit.timeit(stmt=code_snippet, number=1, globals=globals())

print(

    f"Iterated over {len(pubmed_dataset)} examples (about {size_gb:.1f} GB) in "

    f"{time:.1f}s, i.e. {size_gb/time:.3f} GB/s"

)

```

```
'Iterated over 15518009 examples (about 19.5 GB) in 64.2s, i.e. 0.304 GB/s'
```

Here we've used Python's `timeit` module to measure the execution time taken by `code_snippet`. You'll typically be able to iterate over a dataset at speed of a few tenths of a GB/s to several GB/s. This works great for the vast majority of applications, but sometimes you'll have to work with a dataset that is too large to even store on your laptop's hard drive. For example, if we tried to download the Pile in its entirety, we'd need 825 GB of free disk space! To handle these cases, 🐼 Datasets provides a streaming feature that allows us to download and access elements on the fly, without needing to download the whole dataset. Let's take a look at how this works.

💡 In Jupyter notebooks you can also time cells using the [%%timeit magic function](#).

Streaming datasets

To enable dataset streaming you just need to pass the `streaming=True` argument to the `load_dataset()` function. For example, let's load the PubMed Abstracts dataset again, but in streaming mode:

```
pubmed_dataset_streamed = load_dataset(  
    "json", data_files=data_files, split="train", streaming=True  
)
```

Instead of the familiar `Dataset` that we've encountered elsewhere in this chapter, the object returned with `streaming=True` is an `IterableDataset`. As the name suggests, to access the elements of an `IterableDataset` we need to iterate over it. We can access the first element of our streamed dataset as follows:

```
next(iter(pubmed_dataset_streamed))
```

```
{'meta': {'pmid': 11409574, 'language': 'eng'},  
 'text': 'Epidemiology of hypoxaemia in children with acute lower respiratory  
infection.\n\nTo determine the prevalence of hypoxaemia in children aged under 5 years  
suffering acute lower respiratory infections (ALRI), the risk factors for hypoxaemia  
in children under 5 years of age with ALRI, and the association of hypoxaemia with an  
increased risk of dying in children of the same age ...'}
```

The elements from a streamed dataset can be processed on the fly using `IterableDataset.map()`, which is useful during training if you need to tokenize the inputs. The process is exactly the same as the one we used to tokenize our dataset in [Chapter 3](#), with the only difference being that outputs are returned one by one:

```
from transformers import AutoTokenizer
```

```
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")

tokenized_dataset = pubmed_dataset_streamed.map(lambda x: tokenizer(x["text"]))

next(iter(tokenized_dataset))
```

```
{'input_ids': [101, 4958, 5178, 4328, 6779, ...], 'attention_mask': [1, 1, 1, 1, 1, ...]}
```

💡 To speed up tokenization with streaming you can pass `batched=True`, as we saw in the last section. It will process the examples batch by batch; the default batch size is 1,000 and can be specified with the `batch_size` argument.

You can also shuffle a streamed dataset using `IterableDataset.shuffle()`, but unlike `Dataset.shuffle()` this only shuffles the elements in a predefined `buffer_size`:

```
shuffled_dataset = pubmed_dataset_streamed.shuffle(buffer_size=10_000, seed=42)

next(iter(shuffled_dataset))
```

```
{'meta': {'pmid': 11410799, 'language': 'eng'},

 'text': 'Randomized study of dose or schedule modification of granulocyte colony-stimulating factor in platinum-based chemotherapy for elderly patients with lung cancer ...'}
```

In this example, we selected a random example from the first 10,000 examples in the buffer. Once an example is accessed, its spot in the buffer is filled with the next example in the corpus (i.e., the 10,001st example in the case above). You can also select elements from a streamed dataset using the `IterableDataset.take()` and `IterableDataset.skip()` functions, which act in a similar way to `Dataset.select()`. For example, to select the first 5 examples in the PubMed Abstracts dataset we can do the following:

```
dataset_head = pubmed_dataset_streamed.take(5)
```

```
list(dataset_head)
```

```
[{'meta': {'pmid': 11409574, 'language': 'eng'},  
  'text': 'Epidemiology of hypoxaemia in children with acute lower respiratory  
infection ...'},  
 {'meta': {'pmid': 11409575, 'language': 'eng'},  
  'text': 'Clinical signs of hypoxaemia in children with acute lower respiratory  
infection: indicators of oxygen therapy ...'},  
 {'meta': {'pmid': 11409576, 'language': 'eng'},  
  'text': "Hypoxaemia in children with severe pneumonia in Papua New Guinea ..."},  
 {'meta': {'pmid': 11409577, 'language': 'eng'},  
  'text': 'Oxygen concentrators and cylinders ...'},  
 {'meta': {'pmid': 11409578, 'language': 'eng'},  
  'text': 'Oxygen supply in rural africa: a personal experience ...'}]
```

Similarly, you can use the `IterableDataset.skip()` function to create training and validation splits from a shuffled dataset as follows:

```
# Skip the first 1,000 examples and include the rest in the training set  
train_dataset = shuffled_dataset.skip(1000)
```



```
# Take the first 1,000 examples for the validation set
```

```
validation_dataset = shuffled_dataset.take(1000)
```

Let's round out our exploration of dataset streaming with a common application: combining multiple datasets together to create a single corpus. 🤖 Datasets provides an `interleave_datasets()` function that converts a list of `IterableDataset` objects into a single `IterableDataset`, where the elements of the new dataset are obtained by alternating among the source examples. This function is especially useful when you're trying to combine large datasets, so as an example let's stream the FreeLaw subset of the Pile, which is a 51 GB dataset of legal opinions from US courts:

```
law_dataset_streamed = load_dataset(  
  
    "json",  
  
    data_files="https://the-eye.eu/public/AI/pile_preliminary_components/FreeLaw_Opinions.jsonl.zst",  
  
    split="train",  
  
    streaming=True,  
  
)  
  
next(iter(law_dataset_streamed))
```

```
{'meta': {'case_ID': '110921.json',  
  
    'case_jurisdiction': 'scotus.tar.gz',  
  
    'date_created': '2010-04-28T17:12:49Z'},  
  
    'text': '\n461 U.S. 238 (1983)\nOLIM ET AL.\nv.\nWAKINEKONA\nNo. 81-1581.\nSupreme Court of United States.\nArgued January 19, 1983.\nDecided April 26, 1983.\nCERTIORARI TO THE UNITED STATES COURT OF APPEALS FOR THE NINTH CIRCUIT\n*239
```

```
Michael A. Lilly, First Deputy Attorney General of Hawaii, argued the cause for
petitioners. With him on the brief was James H. Dannenberg, Deputy Attorney
General...']}]
```

This dataset is large enough to stress the RAM of most laptops, yet we've been able to load and access it without breaking a sweat! Let's now combine the examples from the FreeLaw and PubMed Abstracts datasets with the `interleave_datasets()` function:

```
from itertools import islice
```

```
from datasets import interleave_datasets
```

```
combined_dataset = interleave_datasets([pubmed_dataset_streamed,
law_dataset_streamed])
```

```
list(islice(combined_dataset, 2))
```

```
[{'meta': {'pmid': 11409574, 'language': 'eng'},
```

```
  'text': 'Epidemiology of hypoxaemia in children with acute lower respiratory
infection ...'},
```

```
 {'meta': {'case_ID': '110921.json',
```

```
  'case_jurisdiction': 'scotus.tar.gz',
```

```
  'date_created': '2010-04-28T17:12:49Z'},
```

```
  'text': '\n461 U.S. 238 (1983)\nOLIM ET AL.\nv.\nWAKINEKONA\nNo. 81-1581.\nSupreme
Court of United States.\nArgued January 19, 1983.\nDecided April 26,
1983.\nCERTIORARI TO THE UNITED STATES COURT OF APPEALS FOR THE NINTH CIRCUIT\n*239
Michael A. Lilly, First Deputy Attorney General of Hawaii, argued the cause for
petitioners. With him on the brief was James H. Dannenberg, Deputy Attorney
General...']}]
```

Here we've used the `islice()` function from Python's `itertools` module to select the first two examples from the combined dataset, and we can see that they match the first examples from each of the two source datasets.

Finally, if you want to stream the Pile in its 825 GB entirety, you can grab all the prepared files as follows:

```
base_url = "https://the-eye.eu/public/AI/pile/"

data_files = {

    "train": [base_url + "train/" + f"{idx:02d}.jsonl.zst" for idx in range(30)],

    "validation": base_url + "val.jsonl.zst",

    "test": base_url + "test.jsonl.zst",


}

pile_dataset = load_dataset("json", data_files=data_files, streaming=True)

next(iter(pile_dataset["train"]))
```

```
{'meta': {'pile_set_name': 'Pile-CC'},

 'text': 'It is done, and submitted. You can play "Survival of the Tastiest" on Android, and on the web...'}
}
```

 **Try it out!** Use one of the large Common Crawl corpora like [mc4](#) or [oscar](#) to create a streaming multilingual dataset that represents the spoken proportions of languages in a country of your choice. For example, the four national languages in Switzerland are German, French, Italian, and Romansh, so you could try creating a Swiss corpus by sampling the Oscar subsets according to their spoken proportion.

You now have all the tools you need to load and process datasets of all shapes and sizes — but unless you're exceptionally lucky, there will come a point in your NLP journey where you'll have to actually create a dataset to solve the problem at hand. That's the topic of the next section!

Creating your own dataset

Sometimes the dataset that you need to build an NLP application doesn't exist, so you'll need to create it yourself. In this section we'll show you how to create a corpus of [GitHub issues](#), which are commonly used to track bugs or features in GitHub repositories. This corpus could be used for various purposes, including:

- Exploring how long it takes to close open issues or pull requests
- Training a *multilabel classifier* that can tag issues with metadata based on the issue's description (e.g., "bug," "enhancement," or "question")
- Creating a semantic search engine to find which issues match a user's query

Here we'll focus on creating the corpus, and in the next section we'll tackle the semantic search application. To keep things meta, we'll use the GitHub issues associated with a popular open source project: 🤖 Datasets! Let's take a look at how to get the data and explore the information contained in these issues.

Getting the data

You can find all the issues in 🤖 Datasets by navigating to the repository's [Issues tab](#). As shown in the following screenshot, at the time of writing there were 331 open issues and 668 closed ones.

The screenshot shows the GitHub interface for the `huggingface/datasets` repository. At the top, there are navigation buttons for Watch (216), Unstar (8.7k), and Fork (1.1k). Below that, a horizontal menu contains links for Code, Issues (281), Pull requests (55), Discussions, Actions, Projects (1), Wiki, Security, and Insights. A notification banner reads: "Label issues and pull requests for new contributors. Now, GitHub will help potential first-time contributors discover issues labeled with good first issue." Below the notification, there are filters for "is:issue is:open", "Labels 19", and "Milestones 3", along with a "New issue" button. The main content area shows a list of issues with columns for status, author, label, projects, milestones, assignee, and sort. Three issues are visible:

Status	Issue Title	Author	Label	Projects	Milestones	Assignee	Sort
Open	How to sample every file in a list of files making up a split in a dataset when loading?	brijow					
Open	ConnectionError: Couldn't reach https://raw.githubusercontent.com	jinec	bug				4
Open	document config.HF_DATASETS_OFFLINE and precedence	stas00	enhancement				

If you click on one of these issues you'll find it contains a title, a description, and a set of labels that characterize the issue. An example is shown in the screenshot below.

Enable streaming for Wikipedia corpora #2808

Title

Edit

New issue

Open

lewntun opened this issue on 16 Aug · 0 comments



lewntun commented on 16 Aug

Description

Member

Is your feature request related to a problem? Please describe.

Several of the [Wikipedia corpora](#) on the Hub involve quite large files that would be a good candidate for streaming. Currently it is not possible to stream these corpora:

```
from datasets import load_dataset

# Throws ValueError: Builder wikipedia is not streamable.
wiki_dataset_streamed = load_dataset("wikipedia", "20200501.en", split="train", streaming=True)
```

Given that these corpora are derived from Wikipedia dumps in XML format which are then processed with Apache Beam, I am not sure whether streaming is possible in principle. The goal of this issue is to discuss whether this feature even makes sense :)

Describe the solution you'd like

It would be nice to be able to stream Wikipedia corpora from the Hub with something like

```
from datasets import load_dataset

wiki_dataset_streamed = load_dataset("wikipedia", "20200501.en", split="train", streaming=True)
```



1

Assignees

No one—assign yourself

Labels

enhancement

Projects

None yet

Milestone

No milestone

Linked pull requests

Successfully merging a pull request may close this issue.

None yet

Notifications

Customize

Unsubscribe

You're receiving notifications because you authored the thread.

To download all the repository's issues, we'll use the [GitHub REST API](#) to poll the [Issues endpoint](#). This endpoint returns a list of JSON objects, with each object containing a large number of fields that include the title and description as well as metadata about the status of the issue and so on.

A convenient way to download the issues is via the `requests` library, which is the standard way for making HTTP requests in Python. You can install the library by running:

```
!pip install requests
```

Once the library is installed, you can make GET requests to the `Issues` endpoint by invoking the `requests.get()` function. For example, you can run the following command to retrieve the first issue on the first page:

```
import requests
```

```
url = "https://api.github.com/repos/huggingface/datasets/issues?page=1&per_page=1"

response = requests.get(url)
```

The `response` object contains a lot of useful information about the request, including the HTTP status code:

```
response.status_code
```

```
200
```

where a `200` status means the request was successful (you can find a list of possible HTTP status codes [here](#)). What we are really interested in, though, is the *payload*, which can be accessed in various formats like bytes, strings, or JSON. Since we know our issues are in JSON format, let's inspect the payload as follows:

```
response.json()
```

```
[{'url': 'https://api.github.com/repos/huggingface/datasets/issues/2792',
  'repository_url': 'https://api.github.com/repos/huggingface/datasets',
  'labels_url':
'https://api.github.com/repos/huggingface/datasets/issues/2792/labels{/name}',
  'comments_url':
'https://api.github.com/repos/huggingface/datasets/issues/2792/comments'}
```


```
'events_url':  
'https://api.github.com/repos/huggingface/datasets/issues/2792/events',  
  
'html_url': 'https://github.com/huggingface/datasets/pull/2792',  
  
'id': 968650274,  
  
'node_id': 'MDExO1B1bGxSZXF1ZXN0NzEwNzUyMjc0',  
  
'number': 2792,  
  
'title': 'Update GooAQ',  
  
'user': {'login': 'bhavitvyamalik',  
  
'id': 19718818,  
  
'node_id': 'MDQ6VXNlcjE5NzE4ODE4',  
  
'avatar_url': 'https://avatars.githubusercontent.com/u/19718818?v=4',  
  
'gravatar_id': '',  
  
'url': 'https://api.github.com/users/bhavitvyamalik',  
  
'html_url': 'https://github.com/bhavitvyamalik',  
  
'followers_url': 'https://api.github.com/users/bhavitvyamalik/followers',  
  
'following_url':  
'https://api.github.com/users/bhavitvyamalik/following{/other_user}',  
  
'gists_url': 'https://api.github.com/users/bhavitvyamalik/gists{/gist_id}',  
  
'starred_url':  
'https://api.github.com/users/bhavitvyamalik/starred{/owner}/{/repo}',  
  
'subscriptions_url': 'https://api.github.com/users/bhavitvyamalik/subscriptions',  
  
'organizations_url': 'https://api.github.com/users/bhavitvyamalik/orgs',  
  
'repos_url': 'https://api.github.com/users/bhavitvyamalik/repos',
```



```
'events_url': 'https://api.github.com/users/bhavitvyamalik/events{/privacy}',
'received_events_url':
'https://api.github.com/users/bhavitvyamalik/received_events',
'type': 'User',
'site_admin': False},
'labels': [],
'state': 'open',
'locked': False,
'assignee': None,
'assignees': [],
'milestone': None,
'comments': 1,
'created_at': '2021-08-12T11:40:18Z',
'updated_at': '2021-08-12T12:31:17Z',
'closed_at': None,
'author_association': 'CONTRIBUTOR',
'active_lock_reason': None,
'pull_request': {'url':
'https://api.github.com/repos/huggingface/datasets/pulls/2792',
'html_url': 'https://github.com/huggingface/datasets/pull/2792',
'diff_url': 'https://github.com/huggingface/datasets/pull/2792.diff',
'patch_url': 'https://github.com/huggingface/datasets/pull/2792.patch'},
```


```
'body': '[GooAQ](https://github.com/allenai/gooaq) dataset was recently updated after splits were added for the same. This PR contains new updated GooAQ with train/val/test splits and updated README as well.',  
  
'performed_via_github_app': None}]
```

Whoa, that's a lot of information! We can see useful fields like `title`, `body`, and `number` that describe the issue, as well as information about the GitHub user who opened the issue.

 **Try it out!** Click on a few of the URLs in the JSON payload above to get a feel for what type of information each GitHub issue is linked to.

As described in the GitHub [documentation](#), unauthenticated requests are limited to 60 requests per hour. Although you can increase the `per_page` query parameter to reduce the number of requests you make, you will still hit the rate limit on any repository that has more than a few thousand issues. So instead, you should follow GitHub's [instructions](#) on creating a `personal access token` so that you can boost the rate limit to 5,000 requests per hour. Once you have your token, you can include it as part of the request header:

```
GITHUB_TOKEN = xxx # Copy your GitHub token here  
  
headers = {"Authorization": f"token {GITHUB_TOKEN}"}
```

 Do not share a notebook with your `GITHUB_TOKEN` pasted in it. We recommend you delete the last cell once you have executed it to avoid leaking this information accidentally. Even better, store the token in a `.env` file and use the [python-dotenv library](#) to load it automatically for you as an environment variable.

Now that we have our access token, let's create a function that can download all the issues from a GitHub repository:

```
import time  
  
import math  
  
from pathlib import Path
```

```
import pandas as pd

from tqdm.notebook import tqdm

def fetch_issues(
    owner="huggingface",
    repo="datasets",
    num_issues=10_000,
    rate_limit=5_000,
    issues_path=Path("."),
):
    if not issues_path.is_dir():
        issues_path.mkdir(exist_ok=True)

    batch = []

    all_issues = []

    per_page = 100 # Number of issues to return per page

    num_pages = math.ceil(num_issues / per_page)

    base_url = "https://api.github.com/repos"

    for page in tqdm(range(num_pages)):
```

```

# Query with state=all to get both open and closed issues

query = f"issues?page={page}&per_page={per_page}&state=all"

issues = requests.get(f"{base_url}/{owner}/{repo}/{query}", headers=headers)

batch.extend(issues.json())

if len(batch) > rate_limit and len(all_issues) < num_issues:

    all_issues.extend(batch)

    batch = [] # Flush batch for next time period

    print(f"Reached GitHub rate limit. Sleeping for one hour ...")

    time.sleep(60 * 60 + 1)

all_issues.extend(batch)

df = pd.DataFrame.from_records(all_issues)

df.to_json(f"{issues_path}/{repo}-issues.jsonl", orient="records", lines=True)


print(

    f"Downloaded all the issues for {repo}! Dataset stored at  

    {issues_path}/{repo}-issues.jsonl"

)

```

Now when we call `fetch_issues()` it will download all the issues in batches to avoid exceeding GitHub's limit on the number of requests per hour; the result will be stored in a `repository_name-issues.jsonl` file, where each line is a JSON object that represents an issue. Let's use this function to grab all the issues from  Datasets:

```
# Depending on your internet connection, this can take several minutes to run...
```



```
fetch_issues()
```

Once the issues are downloaded we can load them locally using our newfound skills from [section 2](#):

```
issues_dataset = load_dataset("json", data_files="datasets-issues.jsonl",  
split="train")
```

```
issues_dataset
```

```
Dataset({  
  
  features: ['url', 'repository_url', 'labels_url', 'comments_url', 'events_url',  
'html_url', 'id', 'node_id', 'number', 'title', 'user', 'labels', 'state', 'locked',  
'assignee', 'assignees', 'milestone', 'comments', 'created_at', 'updated_at',  
'closed_at', 'author_association', 'active_lock_reason', 'pull_request', 'body',  
'timeline_url', 'performed_via_github_app'],  
  
  num_rows: 3019  
  
})
```

Great, we've created our first dataset from scratch! But why are there several thousand issues when the [Issues tab](#) of the  Datasets repository only shows around 1,000 issues in total ? As described in the GitHub [documentation](#), that's because we've downloaded all the pull requests as well:

GitHub's REST API v3 considers every pull request an issue, but not every issue is a pull request. For this reason, "Issues" endpoints may return both issues and pull requests in the response. You can identify pull requests by the `pull_request` key. Be aware that the `id` of a pull request returned from "Issues" endpoints will be an issue id.

Since the contents of issues and pull requests are quite different, let's do some minor preprocessing to enable us to distinguish between them.

Cleaning up the data

The above snippet from GitHub's documentation tells us that the `pull_request` column can be used to differentiate between issues and pull requests. Let's look at a random sample to see what the difference is. As we did in [section 3](#), we'll chain `Dataset.shuffle()` and `Dataset.select()` to create a random sample and then zip the `html_url` and `pull_request` columns so we can compare the various URLs:

```
sample = issues_dataset.shuffle(seed=666).select(range(3))

# Print out the URL and pull request entries

for url, pr in zip(sample["html_url"], sample["pull_request"]):

    print(f">> URL: {url}")

    print(f">> Pull request: {pr}\n")
```

```
>> URL: https://github.com/huggingface/datasets/pull/850

>> Pull request: {'url':
'https://api.github.com/repos/huggingface/datasets/pulls/850', 'html_url':
'https://github.com/huggingface/datasets/pull/850', 'diff_url':
'https://github.com/huggingface/datasets/pull/850.diff', 'patch_url':
'https://github.com/huggingface/datasets/pull/850.patch'}

>> URL: https://github.com/huggingface/datasets/issues/2773


>> Pull request: None
```

```
>> URL: https://github.com/huggingface/datasets/pull/783
```

```
>> Pull request: {'url':  
'https://api.github.com/repos/huggingface/datasets/pulls/783', 'html_url':  
'https://github.com/huggingface/datasets/pull/783', 'diff_url':  
'https://github.com/huggingface/datasets/pull/783.diff', 'patch_url':  
'https://github.com/huggingface/datasets/pull/783.patch'}
```

Here we can see that each pull request is associated with various URLs, while ordinary issues have a None entry. We can use this distinction to create a new `is_pull_request` column that checks whether the `pull_request` field is None or not:

```
issues_dataset = issues_dataset.map(  
  
    lambda x: {"is_pull_request": False if x["pull_request"] is None else True}  
  
)
```

 **Try it out!** Calculate the average time it takes to close issues in 🤗 Datasets. You may find the `Dataset.filter()` function useful to filter out the pull requests and open issues, and you can use the `Dataset.set_format()` function to convert the dataset to a `DataFrame` so you can easily manipulate the `created_at` and `closed_at` timestamps. For bonus points, calculate the average time it takes to close pull requests.

Although we could proceed to further clean up the dataset by dropping or renaming some columns, it is generally a good practice to keep the dataset as “raw” as possible at this stage so that it can be easily used in multiple applications.

Before we push our dataset to the Hugging Face Hub, let’s deal with one thing that’s missing from it: the comments associated with each issue and pull request. We’ll add them next with — you guessed it — the GitHub REST API!

Augmenting the dataset

As shown in the following screenshot, the comments associated with an issue or pull request provide a rich source of information, especially if we’re interested in building a search engine to answer user queries about the library.



yjernite commented 14 days ago

Member



Adding a Dataset

- **Name:** CommonVoice mid-2021 release
- **Description:** more data in CommonVoice: Languages that have increased the most by percentage are Thai (almost 20x growth, from 12 hours to 250 hours), Luganda (almost 9x growth, from 8 to 80), Esperanto (7x growth, from 100 to 840), and Tamil (almost 8x, from 24 to 220).
- **Paper:** <https://discourse.mozilla.org/t/common-voice-2021-mid-year-dataset-release/83812>
- **Data:** <https://commonvoice.mozilla.org/en/datasets>
- **Motivation:** More data and more varied. I think we just need to add configs in the existing dataset script.

Instructions to add a new dataset can be found [here](#).



1



yjernite added the **dataset request** label 14 days ago



yjernite commented 14 days ago

Member

Author



cc @patrickvonplaten?



patrickvonplaten commented 5 days ago

Member



Does anybody know if there is a bundled link, which would allow direct data download instead of manual?

Something similar to: <https://voice-prod-bundler-ee1969a6ce8178826482b88e843c335139bd3fb4.s3.amazonaws.com/cv-corpus-6.1-2020-12-11/ab.tar.gz> ? cc @patil-suraj



patrickvonplaten linked a pull request that will close this issue 5 days ago

[WIP][Common Voice 7] Add common voice 7.0 #2771

Open

4 tasks

The GitHub REST API provides a [Comments endpoint](#) that returns all the comments associated with an issue number. Let's test the endpoint to see what it returns:

```
issue_number = 2792
```

```
url =  
f"https://api.github.com/repos/huggingface/datasets/issues/{issue_number}/comments"
```

```
response = requests.get(url, headers=headers)
```



```
response.json()
```

```
[{'url':  
'https://api.github.com/repos/huggingface/datasets/issues/comments/897594128',  
  
  'html_url': 'https://github.com/huggingface/datasets/pull/2792#issuecomment-  
897594128',  
  
  'issue_url': 'https://api.github.com/repos/huggingface/datasets/issues/2792',  
  
  'id': 897594128,  
  
  'node_id': 'IC_kwD0Dunzps41gDMQ',  
  
  'user': {'login': 'bhavitvyamalik',  
  
  'id': 19718818,  
  
  'node_id': 'MDQ6VXNlcjE5NzE4ODE4',  
  
  'avatar_url': 'https://avatars.githubusercontent.com/u/19718818?v=4',  
  
  'gravatar_id': '',  
  
  'url': 'https://api.github.com/users/bhavitvyamalik',  
  
  'html_url': 'https://github.com/bhavitvyamalik',  
  
  'followers_url': 'https://api.github.com/users/bhavitvyamalik/followers',  
  
  'following_url':  
'https://api.github.com/users/bhavitvyamalik/following{/other_user}',  
  
  'gists_url': 'https://api.github.com/users/bhavitvyamalik/gists{/gist_id}',  
  
  'starred_url':  
'https://api.github.com/users/bhavitvyamalik/starred{/owner}/{/repo}',  
  
  'subscriptions_url': 'https://api.github.com/users/bhavitvyamalik/subscriptions',
```

```

'organizations_url': 'https://api.github.com/users/bhavitvyamalik/orgs',

'repos_url': 'https://api.github.com/users/bhavitvyamalik/repos',

'events_url': 'https://api.github.com/users/bhavitvyamalik/events{/privacy}',

'received_events_url':
'https://api.github.com/users/bhavitvyamalik/received_events',

'type': 'User',

'site_admin': False},

'created_at': '2021-08-12T12:21:52Z',

'updated_at': '2021-08-12T12:31:17Z',

'author_association': 'CONTRIBUTOR',

'body': "@albertvillanova my tests are failing here:\r\n```\r\n\r\ndataset_name =
'gooaq'\r\n\r\n    def test_load_dataset(self, dataset_name):\r\n        configs =
self.dataset_tester.load_all_configs(dataset_name, is_local=True)[:1]\r\n>
self.dataset_tester.check_load_dataset(dataset_name, configs, is_local=True,
use_local_dummy_data=True)\r\n\r\n\r\ntests/test_dataset_common.py:234: \r\n_ _ _ _ _
_ _ _ _ _
\r\ntests/test_dataset_common.py:187: in check_load_dataset\r\n
self.parent.assertTrue(len(dataset[split]) > 0)\r\nE   AssertionError: False is not
true\r\n```\r\n\r\nWhen I try loading dataset on local machine it works fine. Any
suggestions on how can I avoid this error?",

'performed_via_github_app': None}]

```

We can see that the comment is stored in the `body` field, so let's write a simple function that returns all the comments associated with an issue by picking out the `body` contents for each element in `response.json()`:

```

def get_comments(issue_number):

    url =
f"https://api.github.com/repos/huggingface/datasets/issues/{issue_number}/comments"

```

```
response = requests.get(url, headers=headers)
```

```
return [r["body"] for r in response.json()]
```

```
# Test our function works as expected
```

```
get_comments(2792)
```

```
["@albertvillanova my tests are failing here:\r\n```\r\n\r\ndataset_name =  
'gooaq'\r\n\r\n    def test_load_dataset(self, dataset_name):\r\n        configs =  
self.dataset_tester.load_all_configs(dataset_name, is_local=True)[:1]\r\n>  
self.dataset_tester.check_load_dataset(dataset_name, configs, is_local=True,  
use_local_dummy_data=True)\r\n\r\n\r\nntests/test_dataset_common.py:234: \r\n_ _ _ _ _  
_ _ _ _ _  
\r\n\r\nntests/test_dataset_common.py:187: in check_load_dataset\r\n\r\nself.parent.assertTrue(len(dataset[split]) > 0)\r\n\r\nE   AssertionError: False is not  
true\r\n\r\n```\r\n\r\nWhen I try loading dataset on local machine it works fine. Any  
suggestions on how can I avoid this error?"]
```


This looks good, so let's use `Dataset.map()` to add a new `comments` column to each issue in our dataset:

```
# Depending on your internet connection, this can take a few minutes...
```

```
issues_with_comments_dataset = issues_dataset.map(  
  
    lambda x: {"comments": get_comments(x["number"])}  
  
)
```

The final step is to push our dataset to the Hub. Let's take a look at how we can do that.

Uploading the dataset to the Hugging Face Hub

Now that we have our augmented dataset, it's time to push it to the Hub so we can share it with the community! Uploading a dataset is very simple: just like models and tokenizers from  Transformers, we can use a `push_to_hub()` method to push a dataset. To do that we need an authentication token, which can be obtained by first logging into the Hugging Face Hub with the `notebook_login()` function:

```
from huggingface_hub import notebook_login
```

```
notebook_login()
```

This will create a widget where you can enter your username and password, and an API token will be saved in `~/.huggingface/token`. If you're running the code in a terminal, you can log in via the CLI instead:

```
huggingface-cli login
```

Once we've done this, we can upload our dataset by running:

```
issues_with_comments_dataset.push_to_hub("github-issues")
```

From here, anyone can download the dataset by simply providing `load_dataset()` with the repository ID as the `path` argument:

```
remote_dataset = load_dataset("lewtun/github-issues", split="train")
```

```
remote_dataset
```

```
Dataset({  
  
    features: ['url', 'repository_url', 'labels_url', 'comments_url', 'events_url',  
'html_url', 'id', 'node_id', 'number', 'title', 'user', 'labels', 'state', 'locked',  
'assignee', 'assignees', 'milestone', 'comments', 'created_at', 'updated_at',  
'closed_at', 'author_association', 'active_lock_reason', 'pull_request', 'body',  
'performed_via_github_app', 'is_pull_request'],  
  
    num_rows: 2855  
  
})
```

Cool, we've pushed our dataset to the Hub and it's available for others to use! There's just one important thing left to do: adding a `dataset card` that explains how the corpus was created and provides other useful information for the community.

💡 You can also upload a dataset to the Hugging Face Hub directly from the terminal by using `huggingface-cli` and a bit of Git magic. See the [📖 Datasets guide](#) for details on how to do this.

Creating a dataset card

Well-documented datasets are more likely to be useful to others (including your future self!), as they provide the context to enable users to decide whether the dataset is relevant to their task and to evaluate any potential biases in or risks associated with using the dataset.

On the Hugging Face Hub, this information is stored in each dataset repository's `README.md` file. There are two main steps you should take before creating this file:

1. Use the [datasets-tagging application](#) to create metadata tags in YAML format. These tags are used for a variety of search features on the Hugging Face Hub and ensure your dataset can be easily found by members of the community. Since we have created a custom dataset here, you'll need to clone the `datasets-tagging` repository and run the application locally. Here's what the interface looks like:

Dataset name

Pick a nice descriptive name for the dataset

Hugging Face GitHub Issues

Supported tasks

Task category

What categories of task does the dataset support?

text-classification: pr... ✕

text-retrieval: informa... ✕

Specific *text-classification* tasks

What specific tasks does the dataset support?

multi-class-classifica... ✕

multi-label-classificat... ✕

Specific *text-retrieval* tasks

What specific tasks does the dataset support?

document-retrieval ✕

Finalized tag set

✅ This is a valid tagset! 😊

```
annotations_creators:
- no-annotation
language_creators:
- found
languages:
- en
licenses:
- unknown
multilinguality:
- monolingual
pretty_name: Hugging Face GitHub Issues
size_categories:
- unknown
source_datasets:
- original
task_categories:
- text-classification
- text-retrieval
task_ids:
- multi-class-classification
- multi-label-classification
- document-retrieval
```

2. Read the [📖 Datasets guide](#) on creating informative dataset cards and use it as a template.

You can create the `README.md` file directly on the Hub, and you can find a template dataset card in the `lewtun/github-issues` dataset repository. A screenshot of the filled-out dataset card is shown below.

Dataset Structure

Data Instances

Data Fields

Data Splits

Dataset Creation

Curation Rationale


Source Data


Annotations


Personal and Sensitive I...



Dataset Card for GitHub Issues

Dataset Summary

GitHub Issues is a dataset consisting of GitHub issues and pull requests associated with the  Datasets [repository](#). It is intended for educational purposes and can be used for semantic search or multilabel text classification. The contents of each GitHub issue are in English and concern the domain of datasets for NLP, computer vision, and beyond.

 **Try it out!** Use the `dataset-tagging` application and [📖 Datasets guide](#) to complete the `README.md` file for your GitHub issues dataset.

That's it! We've seen in this section that creating a good dataset can be quite involved, but fortunately uploading it and sharing it with the community is not. In the next section we'll use our new dataset to create a semantic search engine with  Datasets that can match questions to the most relevant issues and comments.

 **Try it out!** Go through the steps we took in this section to create a dataset of GitHub issues for your favorite open source library (pick something other than  Datasets, of course!). For bonus points, fine-tune a multilabel classifier to predict the tags present in the `labels` field.

Datasets, check!



Well, that was quite a tour through the 🤗 Datasets library — congratulations on making it this far! With the knowledge that you've gained from this chapter, you should be able to:

- Load datasets from anywhere, be it the Hugging Face Hub, your laptop, or a remote server at your company.
- Wrangle your data using a mix of the `Dataset.map()` and `Dataset.filter()` functions.
- Quickly switch between data formats like Pandas and NumPy using `Dataset.set_format()`.
- Create your very own dataset and push it to the Hugging Face Hub.
- Embed your documents using a Transformer model and build a semantic search engine using FAISS.

Completed Code Examples

Note: Some of the output is cut off in the following examples to fit the document.
All code blocks and text blocks in the notebook can be seen.

Some examples require a Hugging Face account and a Google Colab Account. The free versions of the accounts can be used, but be aware you may have to reset your runtime if you exceed your processing runtime or memory limitations. To avoid this, close each notebook before proceeding to the next example.