Search models, datasets, users...

0. **Setup**

1. **Transformer models**

   Transformers, why are they so damn cool?

   Introduction

   Natural Language Processing

   Transformers, what can they do?

   How do Transformers work?

   Encoder models

   Decoder models

   Sequence-to-sequence models

   Bias and limitations

   Summary

   End-of-chapter quiz

2. **Using 🤗 Transformers**

3. **Fine-tuning a pretrained model**

4. **Sharing models and tokenizers**

Pytorch TensorFlow

Open in Colab

Transformers, what can they do?

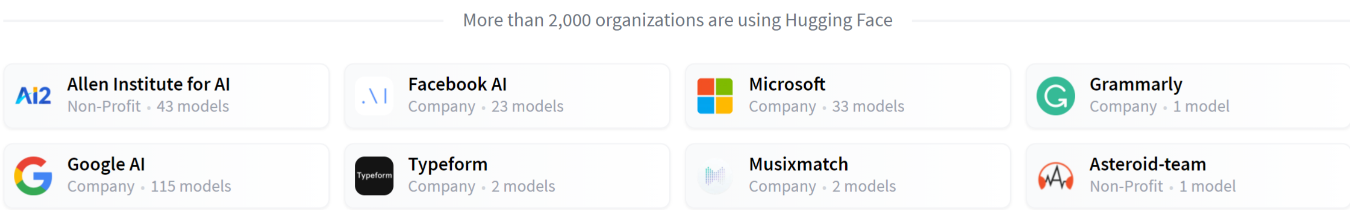
In this section, we will look at what Transformer models can do and use our first tool from the 🤗 Transformers library: the `pipeline`.

👀 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:



The 🤗 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`**. 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, 0.043427448719739
```

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 underst
                    'data flow and data interchange when handling use
                    'will be working with one or more of the most com
                    'data flows — data flows of various types, as see
                    '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 manipul
                    'move your mental and physical capabilities to yo
 {'generated_text': 'In this course, we will teach you how to become 
                    'practice realtime, and with a hands on experienc
                    '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
```

```
[{'sequence': 'This course will teach you all about mathematical mode
  'score': 0.19619831442832947,
  'token': 30412,
  'token_str': ' mathematical'},
 {'sequence': 'This course will teach you all about computational moc
  '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'
 {'entity_group': 'ORG', 'score': 0.97960, 'word': 'Hugging Face', 's
 {'entity_group': 'LOC', 'score': 0.99321, 'word': 'Brooklyn', 'start
]
```

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 Brookly
)
```

```
{'score': 0.6385916471481323, 'start': 33, 'end': 45, 'answer': 'Hugg
```

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 ha
    graduates in traditional engineering disciplines such as mechanic
    electrical, chemical, and aeronautical engineering declined, but 
    the premier American universities engineering curricula now conce
    and encourage largely the study of engineering science. As a resu
    are declining offerings in engineering subjects dealing with infr
    the environment, and related issues, and greater concentration or
    technology subjects, largely supporting increasingly complex scie
    developments. While the latter is important, it should not be at 
    of more traditional engineering.

    Rapidly developing economies such as China and India, as well as 
    industrial countries in Europe and Asia, continue to encourage an
    the teaching of engineering. Both China and India, respectively, 
    six and eight times as many traditional engineers as does the Uni
    Other industrial countries at minimum maintain their output, whil
    suffers an increasingly serious decline in the number of engineer
    and a lack of well-educated engineers.
""")
```

```
[{'summary_text': ' America has changed dramatically during recent ye
                  'number of engineering graduates in the U.S. has de
                  'traditional engineering disciplines such as mechan
                  ', electrical, chemical, and aeronautical engineeri
```

```
                     'developing economies such as China and India, as w
                     'industrial countries in Europe and Asia, continue
                     '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-e
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`** and how to customize its behavior.

Next Section >