Chapter 2: A Hands-On Introduction to the Subject

So far, we have had an overall look at the evolution of **Natural Language Processing (NLP)** using **Deep Learning (DL)**-based methods. We have learned some basic information about Transformer and their respective architecture. In this chapter, we are going to have a deeper look into how a transformer model can be used. Tokenizers and models, such as **Bidirectional Encoder Representations from Transformer (BERT)**, will be described in more technical detail in this chapter with hands-on examples, including how to load a tokenizer/model and use community-provided pretrained models. But before using any specific model, we will understand the installation steps required to provide the necessary environment by using Anaconda. In the installation steps, installing libraries and programs on various operating systems such as Linux, Windows, and macOS will be covered. The installation of **PyTorch** and **TensorFlow**, in two versions of a **Central Processing Unit (CPU)** and a **Graphics Processing Unit (GPU)**, is also shown. A quick jump into a **Google Colaboratory (Google Colab)** installation of the Transformer library is provided. There is also a section dedicated to using models in the PyTorch and TensorFlow frameworks.

The HuggingFace models repository is also another important part of this chapter, in which finding different models and steps to use various pipelines are discussed—for example, models such as **Bidirectional and Auto-Regressive Transformer (BART)**, BERT, and **TAble PArSing (TAPAS)** are detailed, with a glance at **Generative Pre-trained Transformer 2 (GPT-2)** text generation. However, this is purely an overview, and this part of the chapter relates to getting the environment ready and using pretrained models. No model training is discussed here as this is given greater significance in upcoming chapters.

After everything is ready and we have understood how to use the `Transformer` library for inference by community-provided models, the `datasets` library is described. Here, we look at loading various datasets, benchmarks, and using metrics. Loading a specific dataset and getting data back from it is one of the main areas we look at here. Cross-lingual datasets and how to use local files with the `datasets`

library are also considered here. The `map` and `filter` functions are important functions of the `datasets` library in terms of model training and are also examined in this chapter.

This chapter is an essential part of the book because the `datasets` library is described in more detail here. It's also very important for you to understand how to use community-provided models and get the system ready for the rest of the book.

To sum all this up, we will cover the following topics in this chapter:

- Installing Transformer with Anaconda
- Working with language models and tokenizers
- Working with community-provided models
- Working with benchmarks and datasets
- Benchmarking for speed and memory

Technical requirements

You will need to install the libraries and software listed next. Although having the latest version is a plus, it is mandatory to install versions that are compatible with each other. For more information about the latest version installation for HuggingFace Transformer, take a look at their official web page at https://huggingface.co/Transformer/installation.html:

- Anaconda
- Transformer 4.0.0
- PyTorch 1.1.0
- TensorFlow 2.4.0
- Datasets 1.4.1

Finally, all the code shown in this chapter is available in this book's GitHub repository at https://github.com/PacktPublishing/Mastering-Transformer/tree/main/CH02.

Check out the following link to see the Code in Action video: https://bit.ly/372ek48

Installing Transformer with Anaconda

Anaconda is a distribution of the Python and R programming languages that makes package distribution and deployment easy for scientific computation. In this chapter, we will describe the installation of the `Transformer` library. However, it is also possible to install this library without the aid of Anaconda. The main motivation to use Anaconda is to explain the process more easily and moderate the packages used.

To start installing the related libraries, the installation of Anaconda is a mandatory step. Official guidelines provided by the Anaconda documentation offer simple steps to install it for common operating systems (macOS, Windows, and Linux).

Installation on Linux

Many distributions of Linux are available for users to enjoy, but among them, **Ubuntu** is one of the preferred ones. In this section, the steps to install Anaconda are covered for Linux. Proceed as follows:

1. Download the Anaconda installer for Linux from https://www.anaconda.com/products/individual#Downloads and go to the Linux section, as illustrated in the following screenshot:

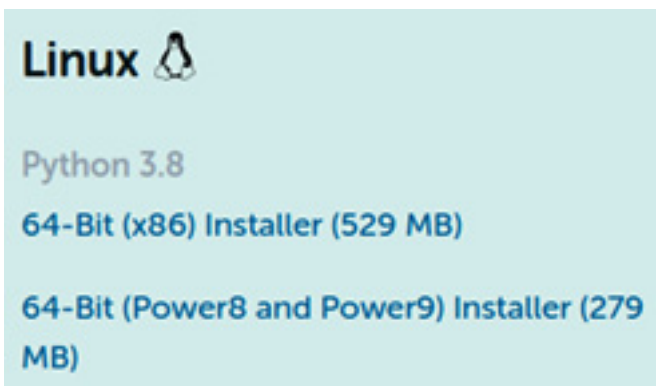

Figure 2.1 – Anaconda download link for Linux

2. Run a `bash` command to install it and complete the following steps:
3. Open the Terminal and run the following command:

```
bash Terminal./FilePath/For/Anaconda.sh
```

4. Press *Enter* to see the license agreement and press *Q* if you do not want to read it all, and then do the following:

5. Click **Yes** to agree.
6. Click **Yes** for the installer to always initialize the `conda` root environment.
7. After running a `python` command from the Terminal, you should see an Anaconda prompt after the Python version information.
8. You can access Anaconda Navigator by running an `anaconda-navigator` command from the Terminal. As a result, you will see the Anaconda **Graphical User Interface** (**GUI**) start loading the related modules, as shown in the following screenshot:

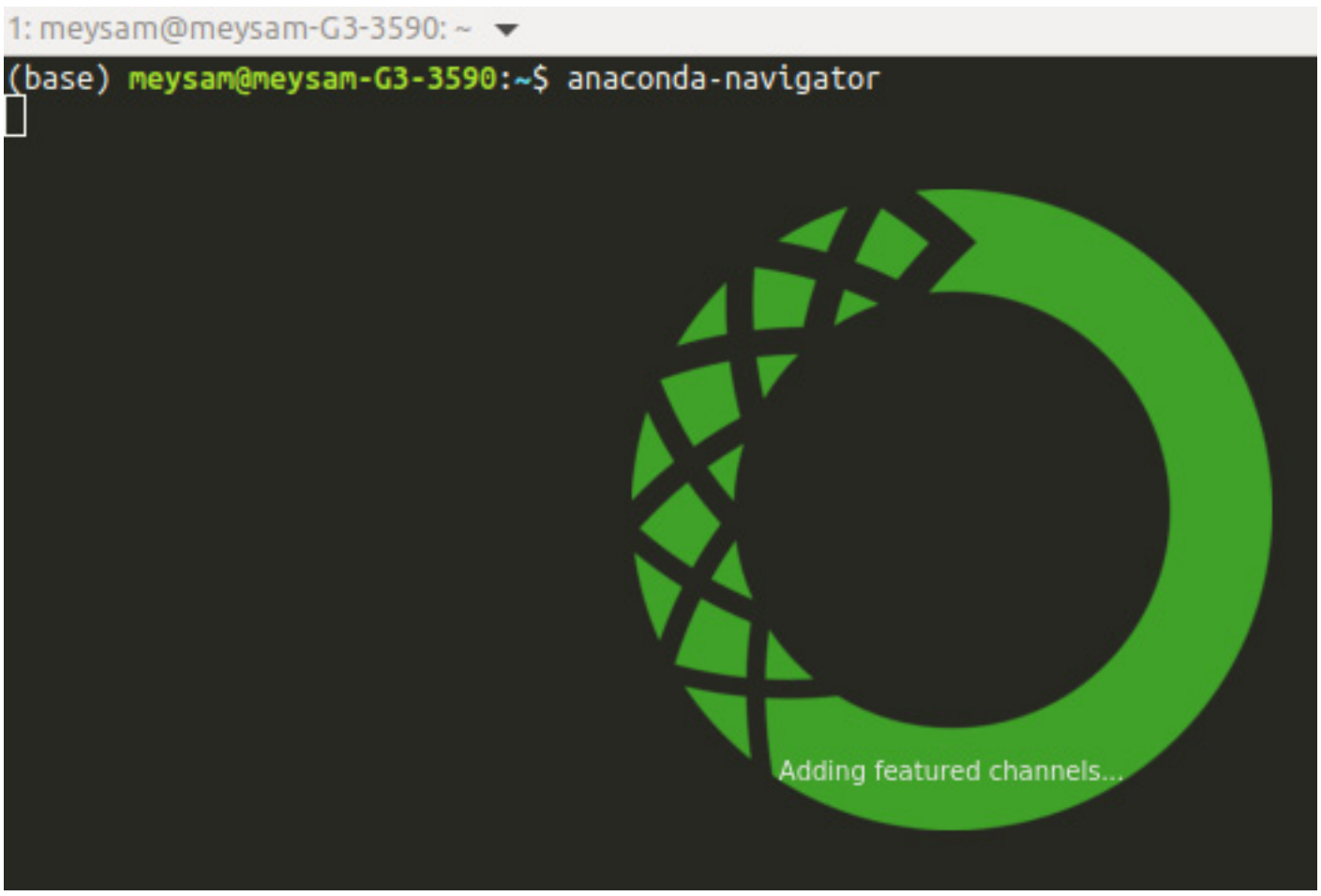

Figure 2.2 – Anaconda Navigator

Let's move on to the next section!

Installation on Windows

The following steps describe how you can install Anaconda on Windows operating systems:

1. Download the installer from https://www.anaconda.com/products/individual#Downloads and go to the Windows section, as illustrated in the following screenshot:

Figure 2.3 – Anaconda download link for Windows

2. Open the installer and follow the guide by clicking the **I Agree** button.
3. Select the location for installation, as illustrated in the following screenshot:

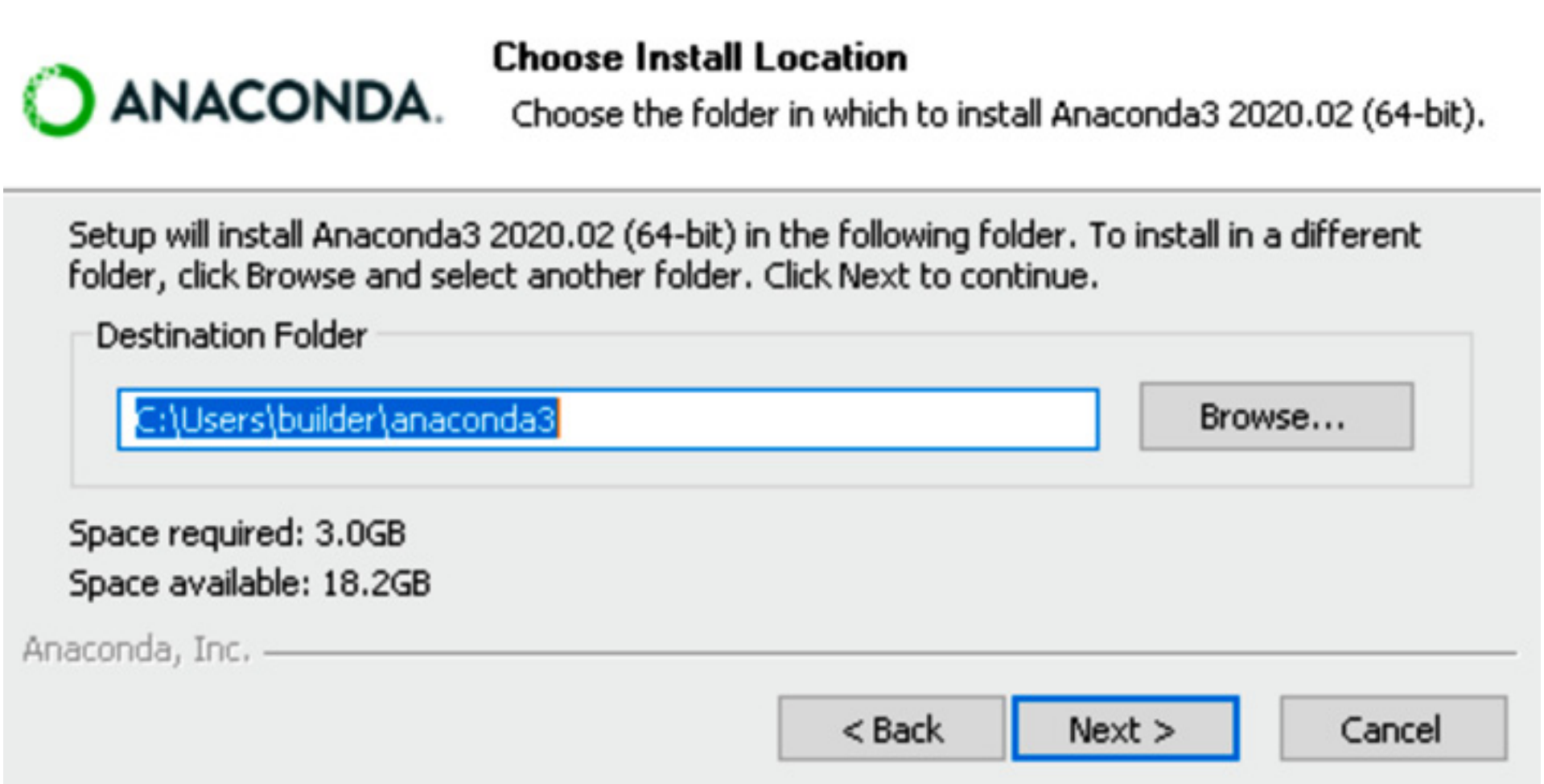

Figure 2.4 – Anaconda installer for Windows

4. Don't forget to check the **Add anaconda3 to my PATH environment variable** checkbox, as illustrated in the following screenshot. If you do not check this box, the Anaconda version of Python will not be added to the Windows environment variables, and you will not be able to directly run it with a `python` command from the Windows shell or the Windows command line:

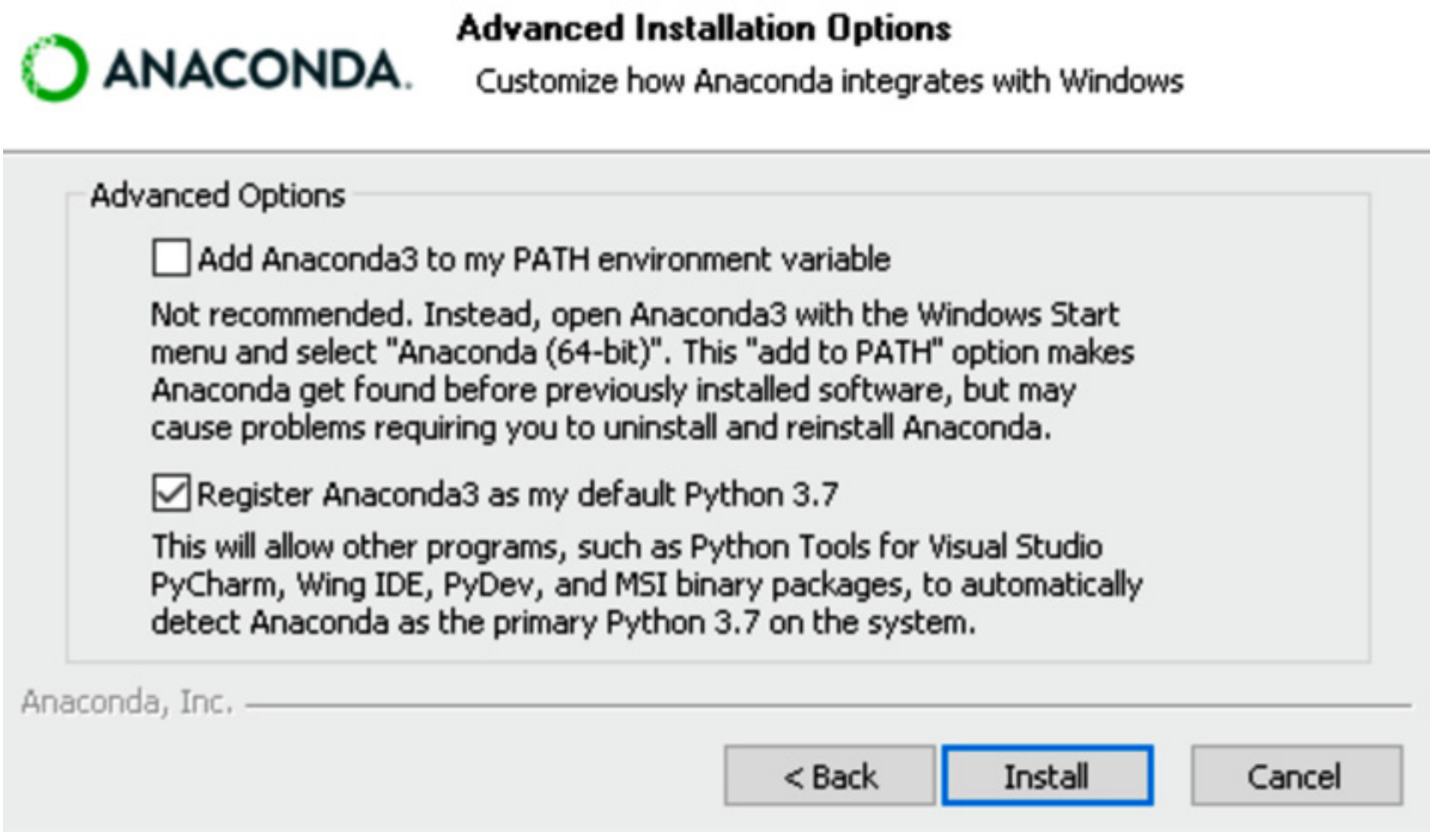

Figure 2.5 – Anaconda installer advanced options

5. Follow the rest of the installation instructions and finish the installation.

You should now be able to start Anaconda Navigator from the **Start** menu.

Installation on macOS

The following steps must be followed to install Anaconda on macOS:

1. Download the installer from https://www.anaconda.com/products/individual#Downloads and go to the macOS section, as illustrated in the following screenshot:

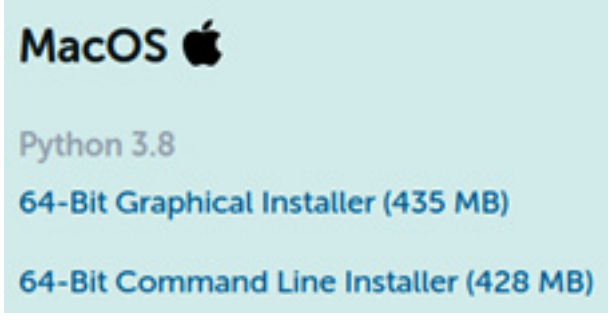

Figure 2.6 – Anaconda download link for macOS

2. Open the installer.
3. Follow the instructions and click the **Install** button to install macOS in a predefined location, as illustrated in the following screenshot. You can change the default directory, but this is not recommended:

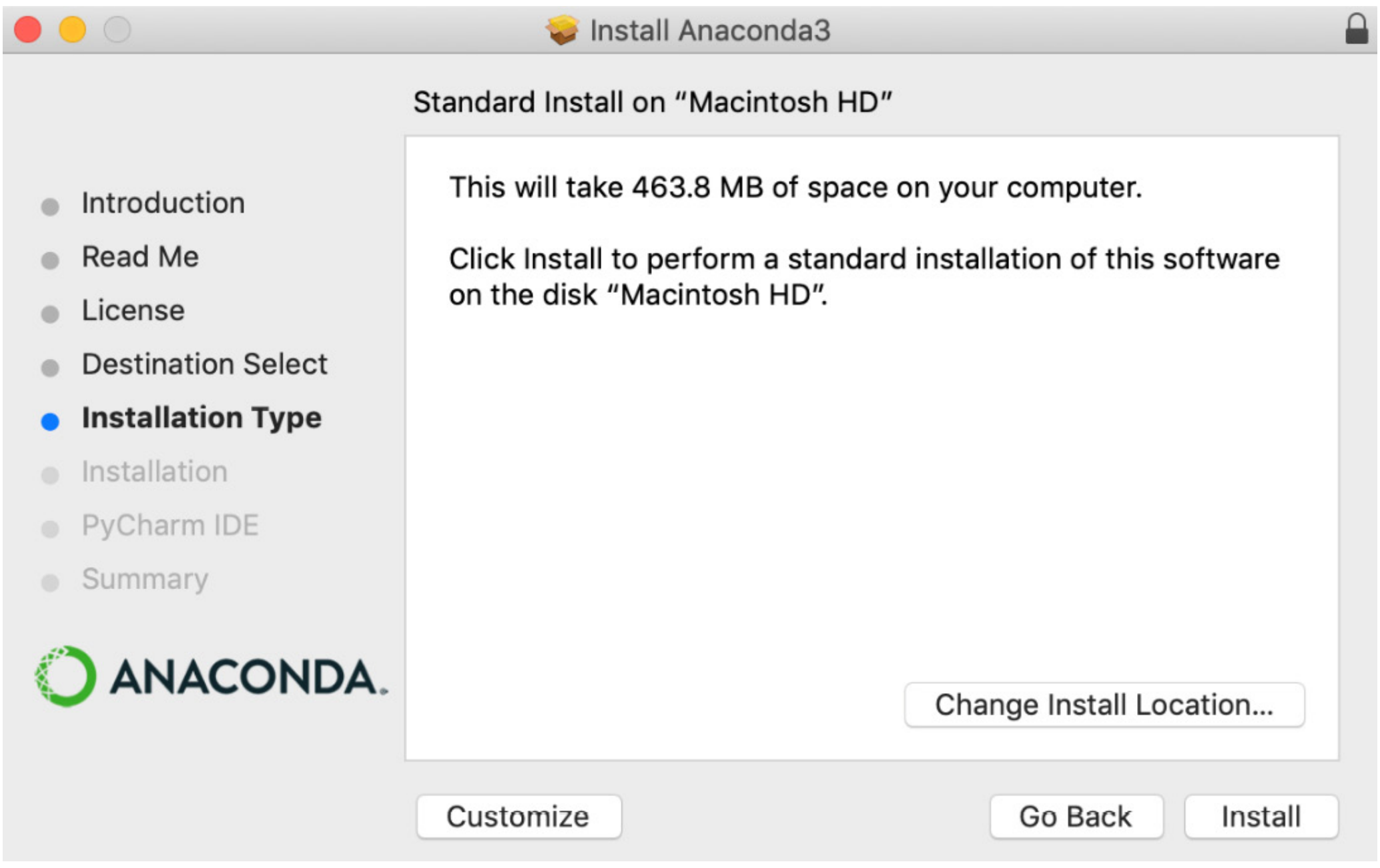

Figure 2.7 – Anaconda installer for macOS

Once you finish the installation, you should be able to access Anaconda Navigator.

Installing TensorFlow, PyTorch, and Transformer

The installation of TensorFlow and PyTorch as two major libraries that are used for DL can be made through `pip` or `conda` itself. `conda` provides a **Command-Line Interface** (**CLI**) for easier installation of these libraries.

For a clean installation and to avoid interrupting other environments, it is better to create a `conda` environment for the `huggingface` library. You can do this by running the following code:

```
conda create –n Transformer
```

This command will create an empty environment for installing other libraries. Once created, we will need to activate it, as follows:

```
conda activate Transformer
```

Installation of the `Transformer` library is easily done by running the following commands:

```
conda install –c conda-forge tensorflow
conda install –c conda-forge pytorch
conda install –c conda-forge Transformer
```

The `–c` argument in the `conda install` command lets Anaconda use additional channels to search for libraries.

Note that it is a requirement to have TensorFlow and PyTorch installed because the `Transformer` library uses both of these libraries. An additional note is the easy handling of CPU and GPU versions of TensorFlow by Conda. If you simply put `–gpu` after `tensorflow`, it will install the GPU version automatically. For installation of PyTorch through the `cuda` library (GPU version), you are required to have related libraries such as `cuda`, but `conda` handles this automatically and no further manual setup or installation is required. The following screenshot shows how `conda` automatically takes care of installing the PyTorch GPU version by installing the related `cudatoolkit` and `cudnn` libraries:

```
1: meysam@meysam-G3-3590: ~
(base) meysam@meysam-G3-3590:~$ conda install pytorch
Collecting package metadata (current_repodata.json): done
Solving environment: /
The environment is inconsistent, please check the package plan carefully
The following packages are causing the inconsistency:
  - defaults/linux-64::anaconda-navigator==1.9.12=py38_0
  - defaults/linux-64::ipykernel==5.3.4=py38h5ca1d4c_0
  - defaults/noarch::conda-verify==3.4.2=py_1
  - defaults/linux-64::notebook==6.1.4=py38_0
  - defaults/linux-64::nb_conda_kernels==2.2.3=py38_0
  - defaults/noarch::nbclient==0.5.1=py_0
  - conda-forge/linux-64::conda==4.9.2=py38h578d9bd_0
  - defaults/noarch::jupyter_client==6.1.7=py_0
  - defaults/linux-64::terminado==0.9.1=py38_0
  - defaults/linux-64::nbconvert==6.0.7=py38_0
  - defaults/linux-64::conda-build==3.18.11=py38_0
  - conda-forge/linux-64::widgetsnbextension==3.5.1=py38h32f6830_1
  - conda-forge/noarch::ipywidgets==7.5.1=pyh9f0ad1d_1
  - defaults/linux-64::_ipyw_jlab_nb_ext_conf==0.1.0=py38_0
  - defaults/linux-64::conda-package-handling==1.7.2=py38h03888b9_0
done

## Package Plan ##

  environment location: /home/meysam/anaconda3

  added / updated specs:
    - pytorch

The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    _openmp_mutex-4.5          |            1_gnu          22 KB
    _pytorch_select-0.1        |            cpu_0           3 KB
    anyio-2.2.0                |   py38h06a4308_1         125 KB
    babel-2.9.1                |     pyhd3eb1b0_0         5.5 MB
    ca-certificates-2021.7.5   |       h06a4308_1         113 KB
    certifi-2021.5.30          |   py38h06a4308_0         138 KB
```

Figure 2.8 – Conda installing PyTorch and related cuda libraries

Note that all of these installations can also be done without `conda`, but the reason behind using Anaconda is its ease of use. In terms of using environments or installing GPU versions of TensorFlow or PyTorch, Anaconda works like magic and is a good time saver.

Installing using Google Colab

Even if the utilization of Anaconda saves time and is useful, in most cases, not everyone has such a good and reasonable computation resource available. Google Colab is a good alternative in such cases. Installation of the `Transformer` library in Colab is carried out with the following command:

Copy Explain

```
!pip install Transformer
```

An exclamation mark before the statement makes the code run in a Colab shell, which is equivalent to running the code in the Terminal instead of running it using a Python interpreter. This will automatically install the `Transformer` library.

Working with language models and tokenizers

In this section, we will look at using the `Transformer` library with language models, along with their related **tokenizers**. In order to use any specified language model, we first need to import it. We will start with the BERT model provided by Google and use its pretrained version, as follows:

```
>>> from Transformer import BERTTokenizer
>>> tokenizer = \
BERTTokenizer.from_pretrained('BERT-base-uncased')
```

The first line of the preceding code snippet imports the BERT tokenizer, and the second line downloads a pretrained tokenizer for the BERT base version. Note that the uncased version is trained with uncased letters, so it does not matter whether the letters appear in upper- or lowercase. To test and see the output, you must run the following line of code:

```
>>> text = "Using Transformer is easy!"
>>> tokenizer(text)
```

This will be the output:

```
{'input_ids': [101, 2478, 19081, 2003, 3733, 999, 102], 'token_type_ids': [0, 0, 0, 0,
0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1]}
```

`input_ids` shows the token ID for each token, and `token_type_ids` shows the type of each token that separates the first and second sequence, as shown in the following screenshot:

```
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
| first sequence    | second sequence |
```

Figure 2.9 – Sequence separation for BERT

`attention_mask` is a mask of 0s and 1s that is used to show the start and end of a sequence for the transformer model in order to prevent unnecessary computations. Each tokenizer has its own way of adding special tokens to the original sequence. In the case of the BERT tokenizer, it adds a `[CLS]` token to the beginning and an `[SEP]` token to the end of the sequence, which can be seen by 101 and 102. These numbers come from the token IDs of the pretrained tokenizer.

A tokenizer can be used for both PyTorch- and TensorFlow-based `Transformer` models. In order to have output for each one, `pt` and `tf` keywords must be used in `return_tensors`. For example, you can use a tokenizer by simply running the following command:

Copy Explain

```
>>> encoded_input = tokenizer(text, return_tensors="pt")
```

`encoded_input` has the tokenized text to be used by the PyTorch model. In order to run the model—for example, the BERT base model—the following code can be used to download the model from the `huggingface` model repository:

Copy Explain

```
>>> from Transformer import BERTModel
>>> model = BERTModel.from_pretrained("BERT-base-uncased")
```

The output of the tokenizer can be passed to the downloaded model with the following line of code:

Copy Explain

```
>>> output = model(**encoded_input)
```

This will give you the output of the model in the form of embeddings and cross-attention outputs.

When loading and importing models, you can specify which version of a model you are trying to use. If you simply put `TF` before the name of a model, the `Transformer` library will load the TensorFlow version of it. The following code shows how to load and use the TensorFlow version of BERT base:

```
from Transformer import BERTTokenizer, TFBERTModel
tokenizer = \
BERTTokenizer.from_pretrained('BERT-base-uncased')
model = TFBERTModel.from_pretrained("BERT-base-uncased")
text = " Using Transformer is easy!"
encoded_input = tokenizer(text, return_tensors='tf')
output = model(**encoded_input)
```

For specific tasks such as filling masks using language models, there are pipelines designed by `huggingface` that are ready to use. For example, a task of filling a mask can be seen in the following code snippet:

```
>>> from Transformer import pipeline
>>> unmasker = \
pipeline('fill-mask', model='BERT-base-uncased')
>>> unmasker("The man worked as a [MASK].")
```

This code will produce the following output, which shows the scores and possible tokens to be placed in the `[MASK]` token:

```
[{'score': 0.09747539460659027,  'sequence': 'the man worked as a
carpenter.',  'token': 10533,  'token_str': 'carpenter'}, {'score':
0.052383217960596085,  'sequence': 'the man worked as a waiter.',  'token':
15610,  'token_str': 'waiter'}, {'score': 0.0496270917356014 25,  'sequence': 'the man
worked as a barber.',  'token': 13362,  'token_str': 'barber'}, {'score':
0.03788605332374573,  'sequence': 'the man worked as a mechanic.',  'token':
15893,  'token_str': 'mechanic'}, {'score': 0.03768084570765495,  'sequence': 'the man
worked as a salesman.',  'token': 18968,  'token_str': 'salesman'}]
```

To get a neat view with pandas, run the following code:

```
>>> pd.DataFrame(unmasker("The man worked as a [MASK]."))
```

The result can be seen in the following screenshot:

	score	sequence	token	token_str
0	0.097475	the man worked as a carpenter.	10533	carpenter
1	0.052383	the man worked as a waiter.	15610	waiter
2	0.049627	the man worked as a barber.	13362	barber
3	0.037886	the man worked as a mechanic.	15893	mechanic
4	0.037681	the man worked as a salesman.	18968	salesman

Figure 2.10 – Output of the BERT mask filling

So far, we have learned how to load and use a pretrained BERT model and have understood the basics of tokenizers, as well as the difference between PyTorch and TensorFlow versions of the models. In the next section, we will learn to work with community-provided models by loading different models, reading the related information provided by the model authors and using different pipelines such as text-generation or **Question Answering** (**QA**) pipelines.

Working with community-provided models

Hugging Face has tons of community models provided by collaborators from large **Artificial Intelligence** (**AI**) and **Information Technology** (**IT**) companies such as Google and Facebook. There are also many interesting models that individuals and universities provide. Accessing and using them is also very easy. To start, you should visit the Transformer models directory available at their website (https://huggingface.co/models), as shown in the following screenshot:

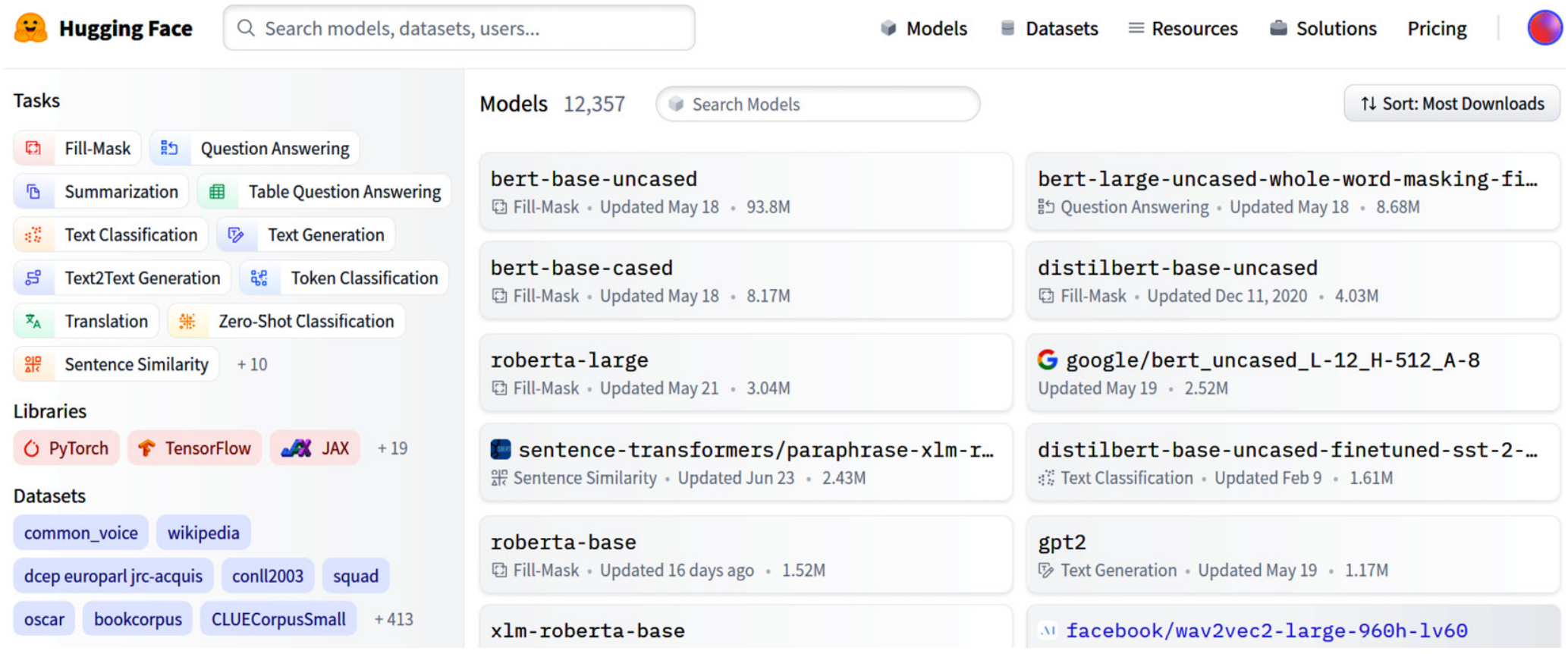

Figure 2.11 – Hugging Face models repository

Apart from these models, there are also many good and useful datasets available for NLP tasks. To start using some of these models, you can explore them by keyword searches, or just specify your major NLP task and pipeline.

For example, we are looking for a table QA model. After finding a model that we are interested in, a page such as the following one will be available from the Hugging Face website (https://huggingface.co/google/tapas-base-finetuned-wtq):

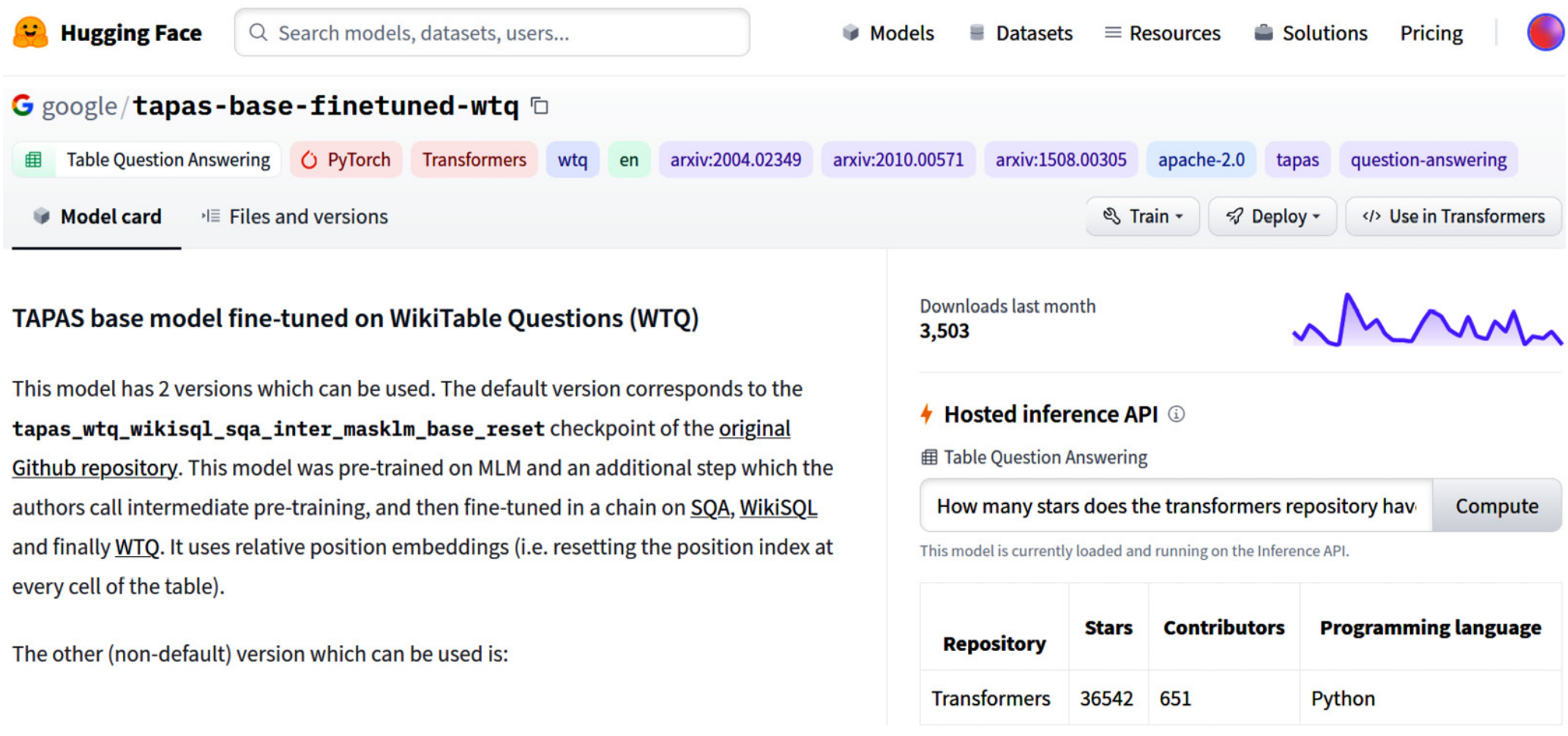

Figure 2.12 – TAPAS model page

On the right side, there is a panel where you can test this model. Note that this is a table QA model that can answer questions about a table provided to the model. If you ask a question, it will reply by highlighting the answer. The following screenshot shows how it gets input and provides an answer for a specific table:

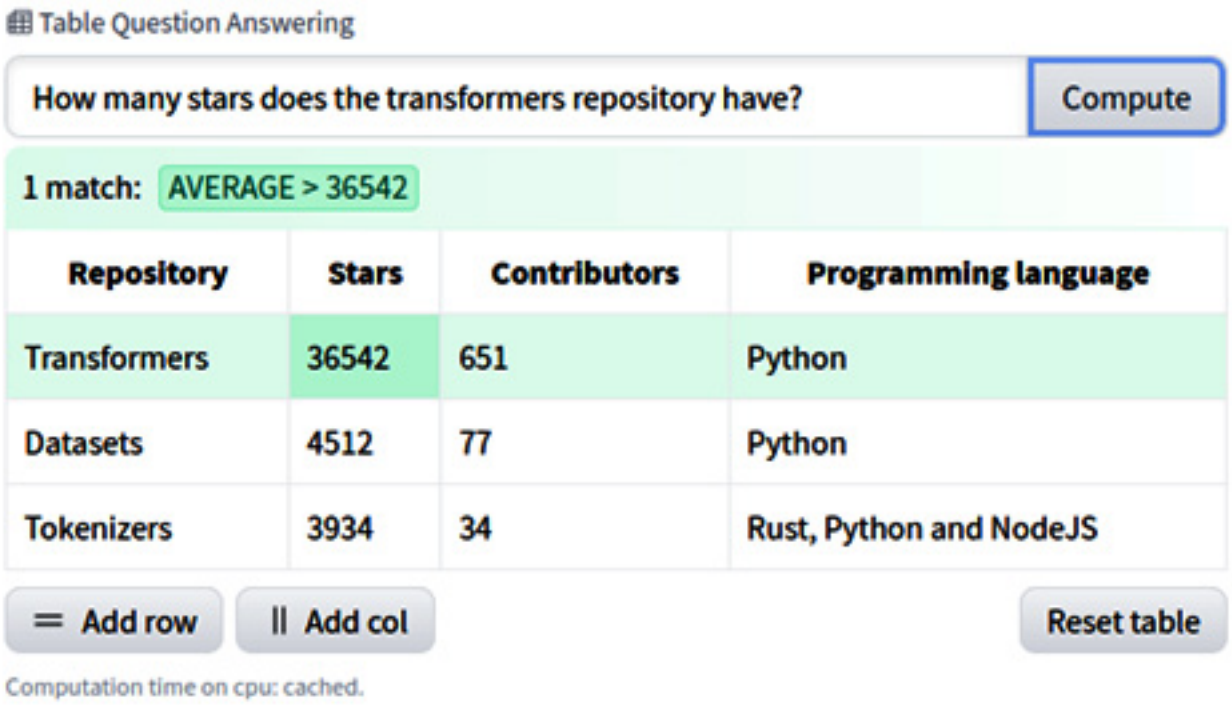

Figure 2.13 – Table QA using TAPAS

Each model has a page provided by the model authors that is also known as a **model card.** You can use the model by the examples provided in the model page. For example, you can visit the GPT-2 `huggingface` repository page and take a look at the example provided by the authors (https://huggingface.co/gpt2), as shown in the following screenshot:

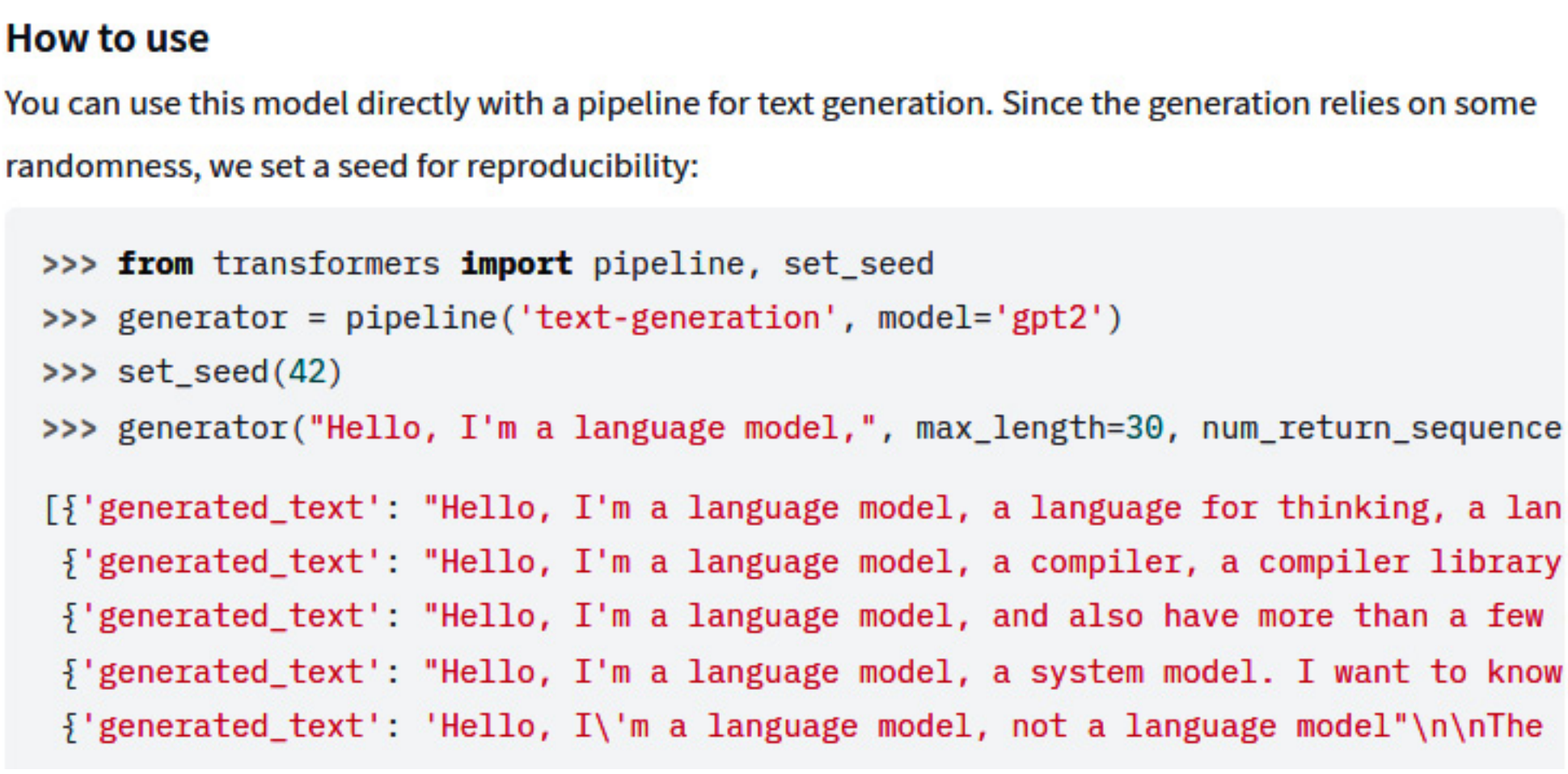

Figure 2.14 – Text-generation code example from the Hugging Face GPT-2 page

Using pipelines is recommended because all the dirty work is taken care of by the `Transformer` library. As another example, let's assume you need an out-of-the-box zero-shot classifier. The following code snippet shows how easy it is to implement and use such a pretrained model:

```
>>> from Transformer import pipeline
>>> classifier = pipeline("zero-shot-classification", model="facebook/bart-large-mnli")
>>> sequence_to_classify = "I am going to france."
>>> candidate_labels = ['travel', 'cooking', 'dancing']
>>> classifier(sequence_to_classify, candidate_labels)
```

Copy Explain

The preceding code will provide the following result:

```
{'labels': ['travel', 'dancing', 'cooking'], 'scores': [0.9866883158683777,
0.007197578903287649, 0.006114077754318714], 'sequence': 'I am going to france.'}
```

Copy Explain

We are done with the installation and the `hello-world` application part. So far, we have introduced the installation process, completed the environment settings, and experienced the first transformer pipeline. In the next part, we will introduce the `datasets` library, which will be our essential utility in the experimental chapters to come.

Working with benchmarks and datasets

Before introducing the `datasets` library, we'd better talk about important benchmarks such as **General Language Understanding Evalution (GLUE)**, **Cross-lingual TRansfer Evaluation of Multilingual Encoders (XTREME)**, and **Stanford Question Answering Dataset (SquAD)**. **Benchmarking** is especially critical for transferring learnings within multitask and multilingual environments. In NLP, we mostly focus on a particular metric that is a performance score on a certain task or dataset. Thanks to the `Transformer` library, we are able to transfer what we have learned from a particular task to a related task, which is called **Transfer Learning (TL)**. By transferring representations between related problems, we are able to train general-purpose models that share common linguistic knowledge across tasks, also known as **Multi-Task Learning (MTL)**. Another aspect of TL is to transfer knowledge across natural languages (multilingual models).

Important benchmarks

In this part, we will introduce the important benchmarks that are widely used by transformer-based architectures. These benchmarks exclusively contribute a lot to MTL and to multilingual and zero-shot learning, including many challenging tasks. We will look at the following benchmarks:

- **GLUE**
- **SuperGLUE**
- **XTREME**
- **XGLUE**
- **SQuAD**

For the sake of using fewer pages, we give details of the tasks for only the GLUE benchmark, so let's look at this benchmark first.

GLUE benchmark

Recent studies addressed the fact that multitask training approaches can achieve better results than single-task learning as a particular model for a task. In this direction, the **GLUE** benchmark has been introduced for MTL, which is a collection of tools and datasets for evaluating the performance of MTL models across a list of tasks. It offers a public leaderboard for monitoring submission performance on the benchmark, along with a single-number metric summarizing 11 tasks. This benchmark includes many sentence-understanding tasks that are based on existing tasks covering various datasets of differing size, text type, and difficulty levels. The tasks are categorized under three types, outlined as follows:

Single-sentence tasks

CoLA: The **Corpus of Linguistic Acceptability** dataset. This task consists of English acceptability judgments drawn from articles on linguistic theory.

SST-2: The **Stanford Sentiment Treebank** dataset. This task includes sentences from movie reviews and human annotations of their sentiment with `pos`/`neg` labels.

Similarity and paraphrase tasks

MRPC: The **Microsoft Research Paraphrase Corpus** dataset. This task looks at whether the sentences in a pair are semantically equivalent.

QQP: The **Quora Question Pairs** dataset. This task decides whether a pair of questions is semantically equivalent.

STS-B: The **Semantic Textual Similarity Benchmark** dataset. This task is a collection of sentence pairs drawn from news headlines, with a similarity score between 1 and 5.

Inference tasks

MNLI: The **Multi-Genre Natural Language Inference** corpus. This is a collection of sentence pairs with textual entailment. The task is to predict whether the text entails a hypothesis (entailment), contradicts the hypothesis (contradiction), or neither (neutral).

QNLI: Question Natural Language Inference dataset. This is a converted version of SquAD. The task is to check whether a sentence contains the answer to a question.

RTE: The **Recognizing Textual Entailment** dataset. This is a task of textual entailment challenges to combine data from various sources. This dataset is similar to the previous QNLI dataset, where the task is to check whether a first text entails a second one.

WNLI: The **Winograd Natural Language Inference** schema challenge. This is originally a pronoun resolution task linking a pronoun and a phrase in a sentence. GLUE converted the problem into sentence-pair classification, as detailed next.

SuperGLUE benchmark

Like Glue, **SuperGLUE** is a new benchmark styled with a new set of more difficult language-understanding tasks and offers a public leaderboard of around currently eight language tasks, drawing on existing data, associated with a single-number performance metric like that of GLUE. The motivation behind it is that as of writing this book, the current state-of-the-art GLUE Score (90.8) surpasses human performance (87.1). Thus, SuperGLUE provides a more challenging and diverse task toward general-purpose, language-understanding technologies.

You can access both GLUE and SuperGLUE benchmarks at gluebenchmark.com.

XTREME benchmark

In recent years, NLP researchers have increasingly focused on learning general-purpose representations rather than a single task that can be applied to many related tasks. Another way of building a general-purpose language model is by using multilingual tasks. It has been observed that recent multilingual models such as **Multilingual BERT (mBERT)** and XLM-R pretrained on massive amounts of multilingual corpora have performed better when transferring them to other languages. Thus, the

main advantage here is that cross-lingual generalization enables us to build successful NLP applications in resource-poor languages through zero-shot cross-lingual transfer.

In this direction, the **XTREME** benchmark has been designed. It currently includes around 40 different languages belonging to 12 language families and includes 9 different tasks that require reasoning for various levels of syntax or semantics. However, it is still challenging to scale up a model to cover over 7,000 world languages and there exists a trade-off between language coverage and model capability. Please check out the following link for more details on this: https://sites.research.google/xtreme.

XGLUE benchmark

XGLUE is another cross-lingual benchmark to evaluate and improve the performance of cross-lingual pretrained models for **Natural Language Understanding (NLU)** and **Natural Language Generation (NLG)**. It originally consisted of 11 tasks over 19 languages. The main difference from XTREME is that the training data is only available in English for each task. This forces the language models to learn only from the textual data in English and transfer this knowledge to other languages, which is called zero-shot cross-lingual transfer capability. The second difference is that it has tasks for NLU and NLG at the same time. Please check out the following link for more details on this: https://microsoft.github.io/XGLUE/.

SQuAD benchmark

SQuAD is a widely used QA dataset in the NLP field. It provides a set of QA pairs to benchmark the reading comprehension capabilities of NLP models. It consists of a list of questions, a reading passage, and an answer annotated by crowdworkers on a set of Wikipedia articles. The answer to the question is a span of text from the reading passage. The initial version, SQuAD1.1, doesn't have an unanswerable option where the datasets are collected, so each question has an answer to be found somewhere in the reading passage. The NLP model is forced to answer the question even if this appears impossible. SQuAD2.0 is an improved version, whereby the NLP models must not only answer questions when possible, but should also abstain from answering when it is impossible to answer. SQuAD2.0 contains 50,000 unanswerable questions written adversarially by crowdworkers to look similar to answerable ones. Additionally, it also has 100,000 questions taken from SQuAD1.1.

Accessing the datasets with an Application Programming Interface

The `datasets` library provides a very efficient utility to load, process, and share datasets with the community through the Hugging Face hub. As with TensorFlow datasets, it makes it easier to download, cache, and dynamically load the sets directly from the original dataset host upon request. The library also provides evaluation metrics along with the data. Indeed, the hub does not hold or distribute the datasets. Instead, it keeps all information about the dataset, including the owner, preprocessing script, description, and download link. We need to check whether we have permission to use the datasets under their corresponding license. To see other features, please check the `dataset_infos.json` and `DataSet-Name.py` files of the corresponding dataset under the GitHub repository, at https://github.com/huggingface/datasets/tree/master/datasets

Let's start by installing the `dataset` library, as follows:

Copy Explain

```
pip install datasets
```

The following code automatically loads the `cola` dataset using the Hugging Face hub. The `datasets.load_dataset()` function downloads the loading script from the actual path if the data is not cached already:

Copy Explain

```
from datasets import load_dataset
cola = load_dataset('glue', 'cola')
cola['train'][25:28]
```

Important note

Reusability of the datasets: As you rerun the code a couple of times, the `datasets` library starts caching your loading and manipulation request. It first stores the dataset and starts caching your operations on the dataset, such as splitting, selection, and sorting. You will see a warning message such as **reusing dataset xtreme (/home/savas/.cache/huggingface/dataset...)** or **loading cached sorted....**

In the preceding example, we downloaded the `cola` dataset from the GLUE benchmark and selected a few examples from the `train` split of it.

Currently, there are 661 NLP datasets and 21 metrics for diverse tasks, as the following code snippet shows:

```
from pprint import pprint
from datasets import list_datasets, list_metrics
all_d = list_datasets()
metrics = list_metrics()
print(f"{len(all_d)} datasets and {len(metrics)} metrics exist in the hub\n")
pprint(all_d[:20], compact=True)
pprint(metrics, compact=True)
```

This is the output:

```
661 datasets and 21 metrics exist in the hub.
['acronym_identification', 'ade_corpus_v2', 'adversarial_qa', 'aeslc',
'afrikaans_ner_corpus', 'ag_news', 'ai2_arc', 'air_dialogue', 'ajgt_twitter_ar',
'allegro_reviews', 'allocine', 'alt', 'amazon_polarity', 'amazon_reviews_multi',
'amazon_us_reviews', 'ambig_qa', 'amttl', 'anli', 'app_reviews', 'aqua_rat']
['accuracy', 'BERTscore', 'bleu', 'bleurt', 'comet', 'coval', 'f1', 'gleu', 'glue',
'indic_glue', 'meteor', 'precision', 'recall', 'rouge', 'sacrebleu', 'sari',
'seqeval', 'squad', 'squad_v2', 'wer', 'xnli']
```

A dataset might have several configurations. For instance, GLUE, as an aggregated benchmark, has many subsets, such as CoLA, SST-2, and MRPC, as we mentioned before. To access each GLUE benchmark dataset, we pass two arguments, where the first is `glue` and the second is a particular dataset of its example dataset (`cola` or `sst2`) that can be chosen. Likewise, the Wikipedia dataset has several configurations provided for several languages.

A dataset comes with the `DatasetDict` object, including several `Dataset` instances. When the split selection `(split='...')` is used, we get `Dataset` instances. For example, the `CoLA` dataset comes with `DatasetDict`, where we have three splits: *train*, *validation*, and *test*. While train and validation datasets include two labels (`1` for acceptable, `0` for unacceptable), the label value of test split is `-1`, which means no-label.

Let's see the structure of the `CoLA` dataset object, as follows:

Copy Explain

```
>>> cola = load_dataset('glue', 'cola')
>>> cola
DatasetDict({
train: Dataset({
features: ['sentence', 'label', 'idx'],
        num_rows: 8551 })
validation: Dataset({
features: ['sentence', 'label', 'idx'],
        num_rows: 1043 })
test: Dataset({
      features: ['sentence', 'label', 'idx'],
       num_rows: 1063  })
})
cola['train'][12]
{'idx': 12, 'label':1,'sentence':'Bill rolled out of the room.'}
>>> cola['validation'][68]
{'idx': 68, 'label': 0, 'sentence': 'Which report that John was incompetent did he
submit?'}
>>> cola['test'][20]
{'idx': 20, 'label': -1, 'sentence': 'Has John seen Mary?'}
```

The dataset object has some additional metadata information that might be helpful for us: `split`, `description`, `citation`, `homepage`, `license`, and `info`. Let's run the following code:

Copy Explain

```
>>> print("1#",cola["train"].description)
>>> print("2#",cola["train"].citation)
>>> print("3#",cola["train"].homepage)
1# GLUE, the General Language Understanding Evaluation
benchmark(https://gluebenchmark.com/) is a collection of resources for
training,evaluating, and analyzing natural language understanding systems.2#
@article{warstadt2018neural,  title={Neural Network Acceptability Judgments},  author=
{Warstadt, Alex and Singh, Amanpreet and Bowman, Samuel R},  journal={arXiv preprint
arXiv:1805.12471},  year={2018}}@inproceedings{wang2019glue,  title={{GLUE}: A Multi-
Task Benchmark and Analysis Platform for Natural Language Understanding},  author=
{Wang, Alex and Singh, Amanpreet and Michael, Julian and Hill, Felix and Levy, Omer
and Bowman, Samuel R.},  note={In the Proceedings of ICLR.},  year={2019}}3#
https://nyu-mll.github.io/CoLA/
```

The GLUE benchmark provides many datasets, as mentioned previously. Let's download the MRPC dataset, as follows:

Copy Explain

```
>>> mrpc = load_dataset('glue', 'mrpc')
```

Likewise, to access other GLUE tasks, we will change the second parameter, as follows:

```
>>> load_dataset('glue', 'XYZ')
```

In order to apply a sanity check of data availability, run the following piece of code:

```
>>> glue=['cola', 'sst2', 'mrpc', 'qqp', 'stsb', 'mnli',
       'mnli_mismatched', 'mnli_matched', 'qnli', 'rte',
       'wnli', 'ax']
>>> for g in glue:
        _=load_dataset('glue', g)
```

XTREME (working with a cross-lingual dataset) is another popular cross-lingual dataset that we already discussed. Let's pick the `MLQA` example from the XTREME set. MLQA is a subset of the XTREME benchmark, which is designed for assessing the performance of cross-lingual QA models. It includes about 5,000 extractive QA instances in the SQuAD format across seven languages, which are English, German, Arabic, Hindi, Vietnamese, Spanish, and Simplified Chinese.

For example, `MLQA.en.de` is an English-German QA example dataset and can be loaded as follows:

```
>>> en_de = load_dataset('xtreme', 'MLQA.en.de')
>>> en_de \
DatasetDict({
test: Dataset({features: ['id', 'title', 'context', 'question', 'answers'], num_rows:
4517
}) validation: Dataset({ features: ['id', 'title', 'context', 'question', 'answers'],
num_rows: 512})})
```

It could be more convenient to view it within a pandas DataFrame, as follows:

```
>>> import pandas as pd
>>> pd.DataFrame(en_de['test'][0:4])
```

Here is the output of the preceding code:

	answers	context	id	question	title
0	{'answer_start': [31], 'text': ['cell']}	An established or immortalized cell line has a...	037e8929e7e4d2f949ffbabd10f0f860499ff7c9	Woraus besteht die Linie?	Cell culture
1	{'answer_start': [232], 'text': ['1885']}	The 19th-century English physiologist Sydney R...	4b36724f3cbde7c287bde512ff09194cbba7f932	Wann hat Roux etwas von seiner Medullarplatte ...	Cell culture
2	{'answer_start': [131], 'text': ['TRIPS']}	After the Uruguay round, the GATT became the b...	13e58403df16d88b0e2c665953e89575704942d4	Was muss ratifiziert werden, wenn ein Land ger...	TRIPS Agreement

Figure 2.15 – English-German cross-lingual QA dataset

Data manipulation with the datasets library

Datasets come with many dictionaries of subsets, where the `split` parameter is used to decide which subset(s) or portion of the subset is to be loaded. If this is `none` by default, it will return a dataset dictionary of all subsets (`train`, `test`, `validation`, or any other combination). If the `split` parameter is specified, it will return a single dataset rather than a dictionary. For the following example, we retrieve a `train` split of the `cola` dataset only:

Copy Explain

```
>>> cola_train = load_dataset('glue', 'cola', split ='train')
```

We can get a mixture of `train` and `validation` subsets, as follows:

Copy Explain

```
>>> cola_sel = load_dataset('glue', 'cola', split = 'train[:300]+validation[-30:]')
```

The `split` expression means that the first 300 examples of `train` and the last 30 examples of `validation` are obtained as `cola_sel`.

We can apply different combinations, as shown in the following split examples:

The first 100 examples from `train` and `validation`, as shown here:

Copy Explain

```
split='train[:100]+validation[:100]'
```

50% of `train` and the last 30% of `validation`, as shown here:

```
split='train[:50%]+validation[-30%:]'
```

The first 20% of `train` and the examples in the slice [30:50] from `validation`, as shown here:

```
split='train[:20%]+validation[30:50]'
```

Sorting, indexing, and shuffling

The following execution calls the `sort()` function of the `cola_sel` object. We see the first 15 and the last 15 labels:

```
>>> cola_sel.sort('label')['label'][:15]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> cola_sel.sort('label')['label'][-15:]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

We are already familiar with Python slicing notation. Likewise, we can also access several rows using similar slice notation or with a list of indices, as follows:

```
>>> cola_sel[6,19,44]
{'idx': [6, 19, 44],
'label': [1, 1, 1],
 'sentence':['Fred watered the plants flat.',
  'The professor talked us into a stupor.',
  'The trolley rumbled through the tunnel.']}
```

We shuffle the dataset as follows:

```
>>> cola_sel.shuffle(seed=42)[2:5]
{'idx': [159, 1022, 46],
'label': [1, 0, 1],
'sentence': ['Mary gets depressed if she listens to the Grateful Dead.',
'It was believed to be illegal by them to do that.',
'The bullets whistled past the house.']}
```

Important note

Seed value: When shuffling, we need to pass a seed value to control the randomness and achieve a consistent output between the author and the reader.

Caching and reusability

Using cached files allows us to load large datasets by means of memory mapping (if datasets fit on the drive) by using a fast backend. Such smart caching helps in saving and reusing the results of operations executed on the drive. To see cache logs with regard to the dataset, run the following code:

Copy Explain

```
>>> cola_sel.cache_files
[{'filename': '/home/savas/.cache/huggingface...,'skip': 0,  'take': 300},
{'filename': '/home/savas/.cache/huggingface...','skip': 1013,  'take': 30}]
```

Dataset filter and map function

We might want to work with a specific selection of a dataset. For instance, we can retrieve sentences only, including the term `kick` in the `cola` dataset, as shown in the following execution. The `datasets.Dataset.filter()` function returns sentences including `kick` where an anonymous function and a `lambda` keyword are applied:

Copy Explain

```
>>> cola_sel = load_dataset('glue', 'cola', split='train[:100%]+validation[-30%:]')
>>> cola_sel.filter(lambda s: "kick" in s['sentence'])["sentence"][:3]
['Jill kicked the ball from home plate to third base.', 'Fred kicked the ball under
the porch.', 'Fred kicked the ball behind the tree.']
```

The following filtering is used to get positive (acceptable) examples from the set:

Copy Explain

```
>>> cola_sel.filter(lambda s: s['label']== 1 )["sentence"][:3]
["Our friends won't buy this analysis, let alone the next one we propose.",
"One more pseudo generalization and I'm giving up.",
"One more pseudo generalization or I'm giving up."]
```

In some cases, we might not know the integer code of a class label. Suppose we have many classes, and the code of the `culture` class is hard to remember out of 10 classes. Instead of giving integer code `1` in our preceding example, which is the code for `acceptable`, we can pass an `acceptable` label to the `str2int()` function, as follows:

```
>>> cola_sel.filter(lambda s: s['label']==
cola_sel.features['label'].str2int('acceptable'))["sentence"][:3]
```

This produces the same output as with the previous execution.

Processing data with the map function

The `datasets.Dataset.map()` function iterates over the dataset, applying a processing function to each example in the set, and modifies the content of the examples. The following execution shows a new `'len'` feature being added that denotes the length of a sentence:

```
>>> cola_new=cola_sel.map(lambda e:{'len': len(e['sentence'])})
>>> pd.DataFrame(cola_new[0:3])
```

This is the output of the preceding code snippet:

	idx	label	len	sentence
0	0	1	71	Our friends won't buy this analysis, let alone...
1	1	1	49	One more pseudo generalization and I'm giving up.
2	2	1	48	One more pseudo generalization or I'm giving up.

Figure 2.16 – Cola dataset an with additional column

As another example, the following piece of code cut the sentence after 20 characters. We do not create a new feature, but instead update the content of the sentence feature, as follows:

```
>>> cola_cut=cola_new.map(lambda e: {'sentence': e['sentence'][:20]+ '_'})
```

The output is shown here:

	idx	label	len	sentence
0	0	1	71	Our friends won't bu_
1	1	1	49	One more pseudo gene_
2	2	1	48	One more pseudo gene_

Figure 2.17 – Cola dataset with an update

Working with local files

To load a dataset from local files in a **Comma-Separated Values** (**CSV**), **Text** (**TXT**), or **JavaScript Object Notation** (**JSON**) format, we pass the file type (`csv`, `text`, or `json`) to the generic `load_dataset()` loading script, as shown in the following code snippet. Under the `../data/` folder, there are three CSV files (`a.csv`, `b.csv`, and `c.csv`), which are randomly selected toy examples from the SST-2 dataset. We can load a single file, as shown in the `data1` object, merge many files, as in the `data2` object, or make dataset splits, as in `data3`:

Copy Explain

```
from datasets import load_dataset
data1 = load_dataset('csv', data_files='../data/a.csv', delimiter="\t")
data2 = load_dataset('csv', data_files=['../data/a.csv','../data/b.csv',
'../data/c.csv'], delimiter="\t")
data3 = load_dataset('csv', data_files={'train':['../data/a.csv','../data/b.csv'],
'test':['../data/c.csv']}, delimiter="\t")
```

In order to get the files in other formats, we pass `json` or `text` instead of `csv`, as follows:

Copy Explain

```
>>> data_json = load_dataset('json', data_files='a.json')
>>> data_text = load_dataset('text', data_files='a.txt')
```

So far, we have discussed how to load, handle, and manipulate datasets that are either already hosted in the hub or are on our local drive. Now, we will study how to prepare datasets for transformer model training.

Preparing a dataset for model training

Let's start with the tokenization process. Each model has its own tokenization model that is trained before the actual language model. We will discuss this in detail in the next chapter. To use a tokenizer, we should have installed the `Transformer` library. The following example loads the tokenizer model from the pretrained `distilBERT-base-uncased` model. We use `map` and an anonymous function with `lambda` to apply a tokenizer to each split in `data3`. If `batched` is selected `True` in the `map` function, it provides a batch of examples to the `tokenizer` function. The `batch_size` value is `1000` by default, which is the number of examples per batch passed to the function. If not selected, the whole dataset is passed as a single batch. The code can be seen here:

```
from Transformer import DistilBERTTokenizer
tokenizer = \ DistilBERTTokenizer.from_pretrained('distilBERT-base-uncased')
encoded_data3 = data3.map(lambda e: tokenizer( e['sentence'], padding=True,
truncation=True, max_length=12), batched=True, batch_size=1000)
```

As shown in the following output, we see the difference between `data3` and `encoded_data3`, where two additional features—`attention_mask` and `input_ids`—are added to the datasets accordingly. We already introduced these two features in the previous part in this chapter. Put simply, `input_ids` are the indices corresponding to each token in the sentence. They are expected features needed by the `Trainer` class of Transformer, which we will discuss in the next fine-tuning chapters.

We mostly pass several sentences at once (called a **batch**) to the tokenizer and further pass the tokenized batch to the model. To do so, we pad each sentence to the maximum sentence length in the batch or a particular maximum length specified by the `max_length` parameter—`12` in this toy example. We also truncate longer sentences to fit that maximum length. The code can be seen in the following snippet:

```
>>> data3
DatasetDict({
train: Dataset({
   features: ['sentence','label'], num_rows: 199 })
test: Dataset({
   features: ['sentence','label'], num_rows: 100 })})
>>> encoded_data3
DatasetDict({
train: Dataset({
  features: ['attention_mask', 'input_ids', 'label',   'sentence'],
   num_rows: 199 })
test: Dataset({
features: ['attention_mask', 'input_ids', 'label', 'sentence'],
 num_rows: 100 })})
>>> pprint(encoded_data3['test'][12])
{'attention_mask': [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0], 'input_ids': [101, 2019,
5186, 16010, 2143, 1012, 102, 0, 0, 0, 0, 0], 'label': 0, 'sentence': 'an extremely
unpleasant film . '}
```

We are done with the `datasets` library. Up to this point, we have evaluated all aspects of datasets. We have covered GLUE-like benchmarking, where classification metrics are taken into consideration. In the next section, we will focus on how to

benchmark computational performance for speed and memory rather than classification.

Benchmarking for speed and memory

Just comparing the classification performance of large models on a specific task or a benchmark turns out to be no longer sufficient. We must now take care of the computational cost of a particular model for a given environment (**Random-Access Memory** (**RAM**), CPU, GPU) in terms of memory usage and speed. The computational cost of training and deploying to production for inference are two main values to be measured. Two classes of the `Transformer` library, `PyTorchBenchmark` and `TensorFlowBenchmark`, make it possible to benchmark models for both TensorFlow and PyTorch.

Before we start our experiment, we need to check our GPU capabilities with the following execution:

Copy Explain

```
>>> import torch
>>> print(f"The GPU total memory is {torch.cuda.get_device_properties(0).total_memory
/(1024**3)} GB")
The GPU total memory is 2.94921875 GB
```

The output is obtained from NVIDIA GeForce GTX 1050 (3 **Gigabytes** (**GB**)). We need more powerful resources for an advanced implementation. The `Transformer` library currently only supports single-device benchmarking. When we conduct benchmarking on a GPU, we are expected to indicate on which GPU device the Python code will run, which is done by setting the `CUDA_VISIBLE_DEVICES` environment variable. For example, `export CUDA_VISIBLE_DEVICES=0`. `0` indicates that the first `cuda` device will be used.

In the code example that follows, two grids are explored. We compare four randomly selected pretrained BERT models, as listed in the `models` array. The second parameter to be observed is `sequence_lengths`. We keep the batch size as `4`. If you have a better GPU capacity, you can extend the parameter search space with batch values in the range 4-64 and other parameters:

```
from Transformer import PyTorchBenchmark, PyTorchBenchmarkArguments
models= ["BERT-base-uncased","distilBERT-base-uncased","distilroBERTa-base",
"distilBERT-base-german-cased"]
batch_sizes=[4]
sequence_lengths=[32,64, 128, 256,512]
args = PyTorchBenchmarkArguments(models=models, batch_sizes=batch_sizes,
sequence_lengths=sequence_lengths, multi_process=False)
benchmark = PyTorchBenchmark(args)
```

Important note

Benchmarking for TensorFlow: The code examples are for PyTorch benchmarking in this part. For TensorFlow benchmarking, we simply use the `TensorFlowBenchmarkArguments` and `TensorFlowBenchmark` counterpart classes instead.

We are ready to conduct the benchmarking experiment by running the following code:

```
>>> results = benchmark.run()
```

This may take some time, depending on your CPU/GPU capacity and argument selection. If you face an out-of-memory problem for it, you should take the following actions to overcome this:

- **Restart your kernel or your operating system.**
- **Delete all unnecessary objects in the memory before starting.**
- **Set a lower batch size, such as 2, or even 1.**

The following output indicates the inference speed performance. Since our search space has four different models and five different sequence lengths, we see 20 rows in the results:

```
====================       INFERENCE - SPEED - RESULT       ====================
--------------------------------------------------------------------------------
          Model Name             Batch Size     Seq Length     Time in s
--------------------------------------------------------------------------------
      bert-base-uncased                4               32            0.021
      bert-base-uncased                4               64            0.031
      bert-base-uncased                4              128            0.057
      bert-base-uncased                4              256             0.12
      bert-base-uncased                4              512            0.269
   distilbert-base-uncased             4               32            0.007
   distilbert-base-uncased             4               64            0.011
   distilbert-base-uncased             4              128            0.021
   distilbert-base-uncased             4              256            0.044
   distilbert-base-uncased             4              512            0.095
      distilroberta-base               4               32            0.009
      distilroberta-base               4               64            0.014
      distilroberta-base               4              128            0.025
      distilroberta-base               4              256            0.053
      distilroberta-base               4              512            0.118
 distilbert-base-german-cased          4               32            0.007
 distilbert-base-german-cased          4               64            0.012
 distilbert-base-german-cased          4              128            0.021
 distilbert-base-german-cased          4              256            0.044
 distilbert-base-german-cased          4              512            0.095
--------------------------------------------------------------------------------
```

Figure 2.18 – Inference speed performance

Likewise, we see the inference memory usage for 20 different scenarios, as follows:

```
====================      INFERENCE - MEMORY - RESULT       ====================
--------------------------------------------------------------------------------
          Model Name             Batch Size     Seq Length    Memory in MB
--------------------------------------------------------------------------------
      bert-base-uncased                4               32            1453
      bert-base-uncased                4               64            1487
      bert-base-uncased                4              128            1547
      bert-base-uncased                4              256            1661
      bert-base-uncased                4              512            1901
   distilbert-base-uncased             4               32            1908
   distilbert-base-uncased             4               64            1900
   distilbert-base-uncased             4              128            1900
   distilbert-base-uncased             4              256            1900
   distilbert-base-uncased             4              512            1900
      distilroberta-base               4               32            1907
      distilroberta-base               4               64            1900
      distilroberta-base               4              128            1900
      distilroberta-base               4              256            2098
      distilroberta-base               4              512            2492
 distilbert-base-german-cased          4               32            2499
 distilbert-base-german-cased          4               64            2492
 distilbert-base-german-cased          4              128            2492
 distilbert-base-german-cased          4              256            2491
 distilbert-base-german-cased          4              512            2491
--------------------------------------------------------------------------------
```

Figure 2.19 – Inference memory usage

To observe the memory usage across the parameters, we will plot them by using the `results` object that stores the statistics. The following execution will plot the time inference performance across models and sequence lengths:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(8,8))
t=sequence_lengths
models_perf=[list(results.time_inference_result[m]['result'][batch_sizes[0]].values())
for m in models]
plt.xlabel('Seq Length')
plt.ylabel('Time in Second')
plt.title('Inference Speed Result')
plt.plot(t, models_perf[0], 'rs--', t, models_perf[1], 'g--.', t, models_perf[2], 'b--
^', t, models_perf[3], 'c--o')
plt.legend(models)
plt.show()
```

As shown in the following screenshot, two DistillBERT models showed close results and performed better than other two models. The `BERT-based-uncased` model performs poorly compared to the others, especially as the sequence length increases:

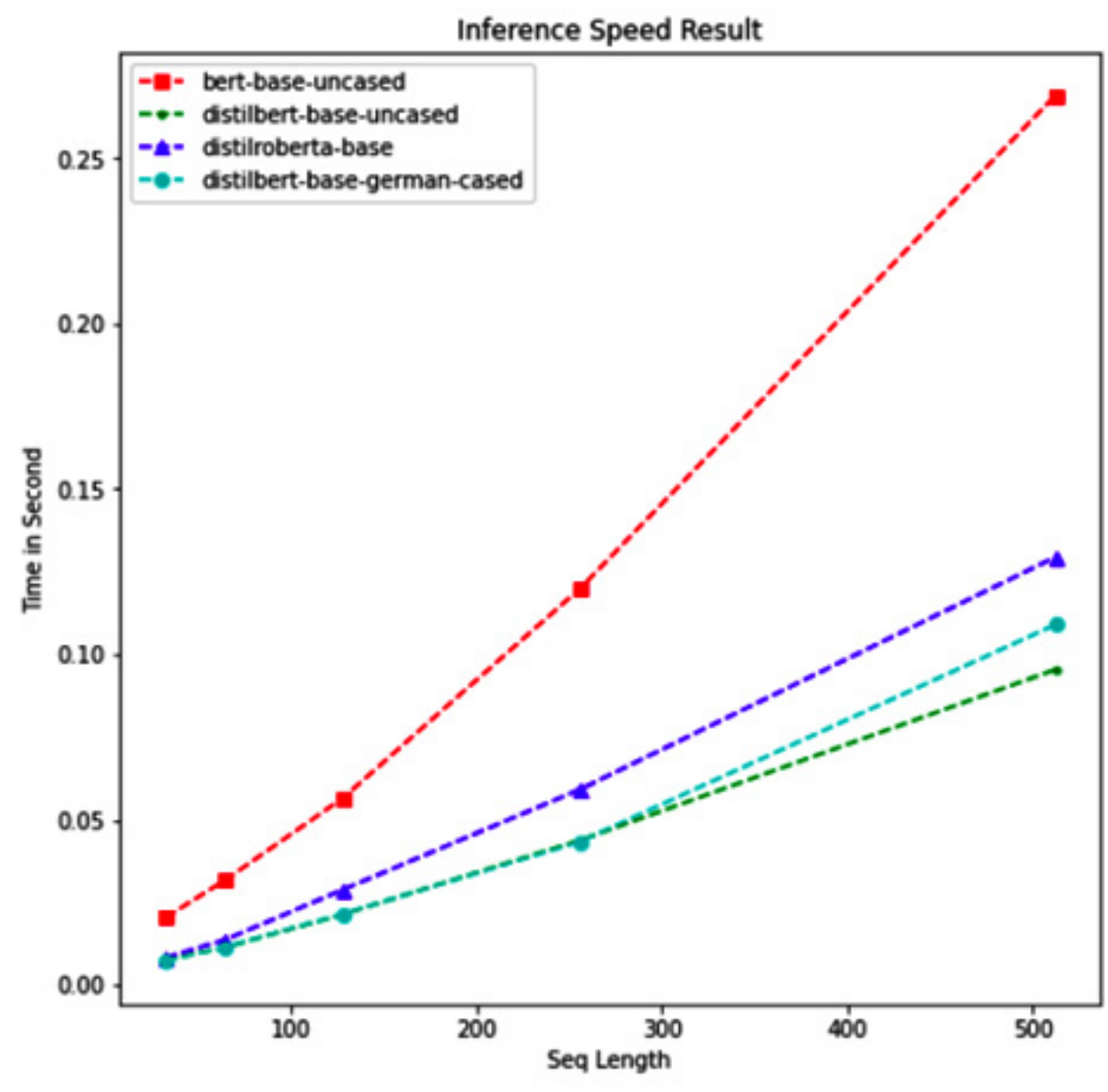

Figure 2.20 – Inference speed result

To plot the memory performance, you need to use the `memory_inference_result` result of the `results` object instead of `time_inference_result`, shown in the preceding code.

For more interesting benchmarking examples, please check out the following links:

https://huggingface.co/transformers/benchmarks.html

https://github.com/huggingface/transformers/tree/master/notebooks

Now that we are done with this section, we successfully completed this chapter. Congratulations on achieving the installation, running your first `hello-world` transformer program, working with the `datasets` library, and benchmarking!

Summary

In this chapter, we have covered a variety of introductory topics and also got our hands dirty with the `hello-world` transformer application. On the other hand, this chapter plays a crucial role in terms of applying what has been learned so far to the upcoming chapters. So, what has been learned so far? We took a first small step by setting the environment and system installation. In this context, the `anaconda` package manager helped us to install the necessary modules for the main operating systems. We also went through language models, community-provided models, and tokenization processes. Additionally, we introduced multitask (GLUE) and cross-lingual benchmarking (XTREME), which enables these language models to become stronger and more accurate. The `datasets` library was introduced, which facilitates efficient access to NLP datasets provided by the community. Finally, we learned how to evaluate the computational cost of a particular model in terms of memory usage and speed. Transformer frameworks make it possible to benchmark models for both TensorFlow and PyTorch.

The models that have been used in this section were already trained and shared with us by the community. Now, it is our turn to train a language model and disseminate it to the community.

In the next chapter, we will learn how to train a BERT language model as well as a tokenizer, and look at how to share them with the community.

Previous Chapter

Next Chapter