4. Follow the rest of the installation instructions and finish the installation.
5. You should now be able to start Anaconda Navigator from the **Start** menu.

Now that we know how to download and install it on Linux, let's see how we can do it on macOS.

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:

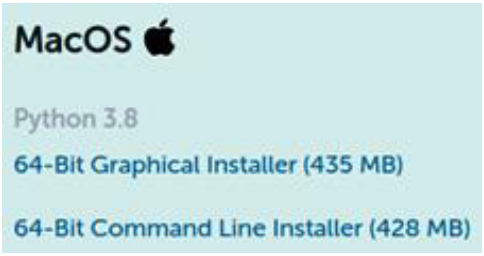

Figure 2.6 – Anaconda download link for macOS

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:

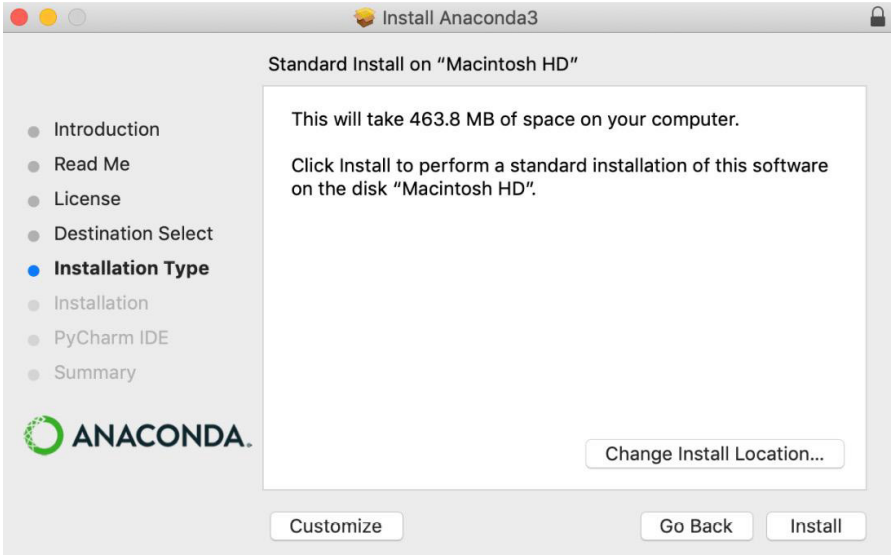

Figure 2.7 – Anaconda installer for macOS

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 the 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
```

The 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 transformers
```

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 the 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 and using Google Colab

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

```
!pip install transformers
```

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

We will see how to work with a BERT model and tokenizer.

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 transformers 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 Transformers 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:

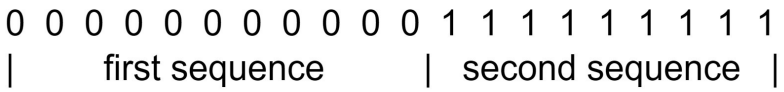

Figure 2.9 – Sequence separation for BERT

`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 an output for each one, the `pt` and `tf` keywords must be used in `return_tensors`. For example, you can use a tokenizer by simply running the following command:

```
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:

```
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:

```
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 transformers 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 Hugging Face that are ready to use. For example, the task of filling a mask can be seen in the following code snippet:

```
from transformers 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': 0.049627091735601425,  '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 using 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. Individuals and universities also provide many interesting models. Accessing and using them is also very easy. To start, you should visit the Transformer models directory available on their website (`https://huggingface.co/models`), as shown in the following screenshot:

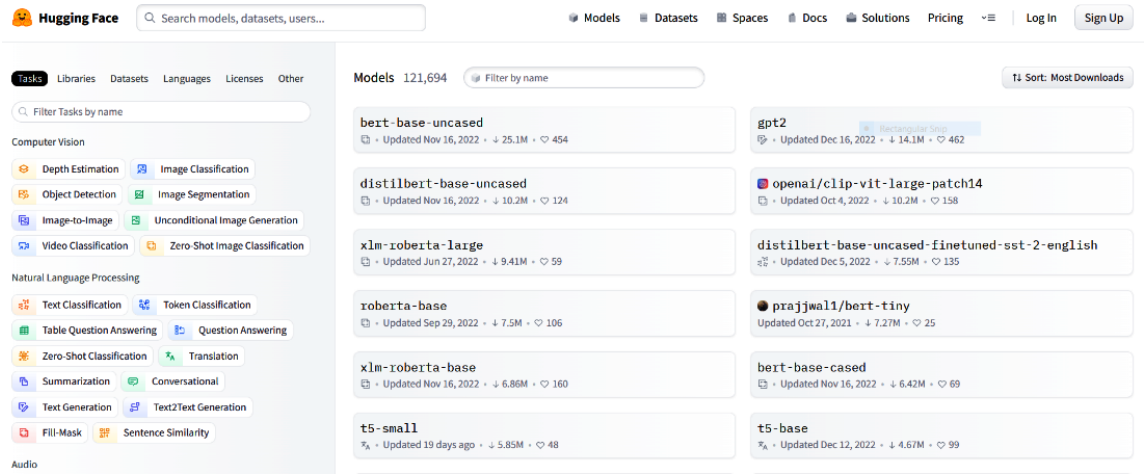

Figure 2.11 – An overview of Hugging Face models repository

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`):

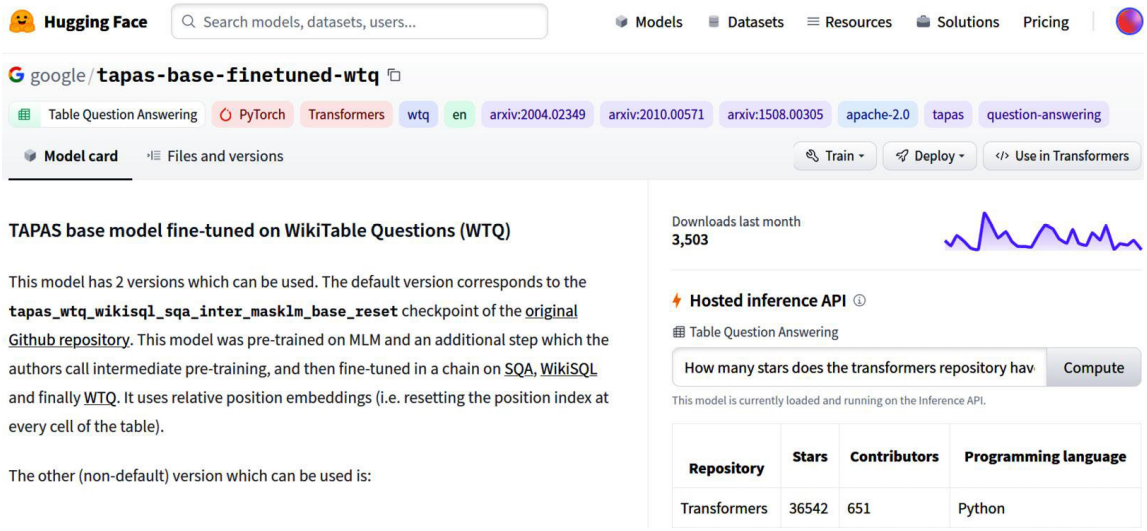

Figure 2.12 – TAPAS model page

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:

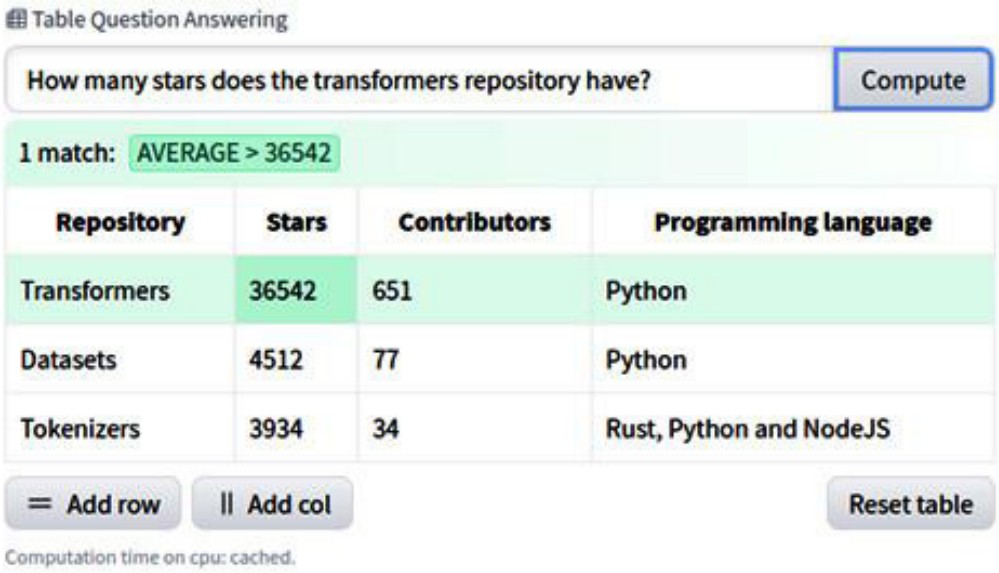

Figure 2.13 – Table QA using TAPAS

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 on 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:

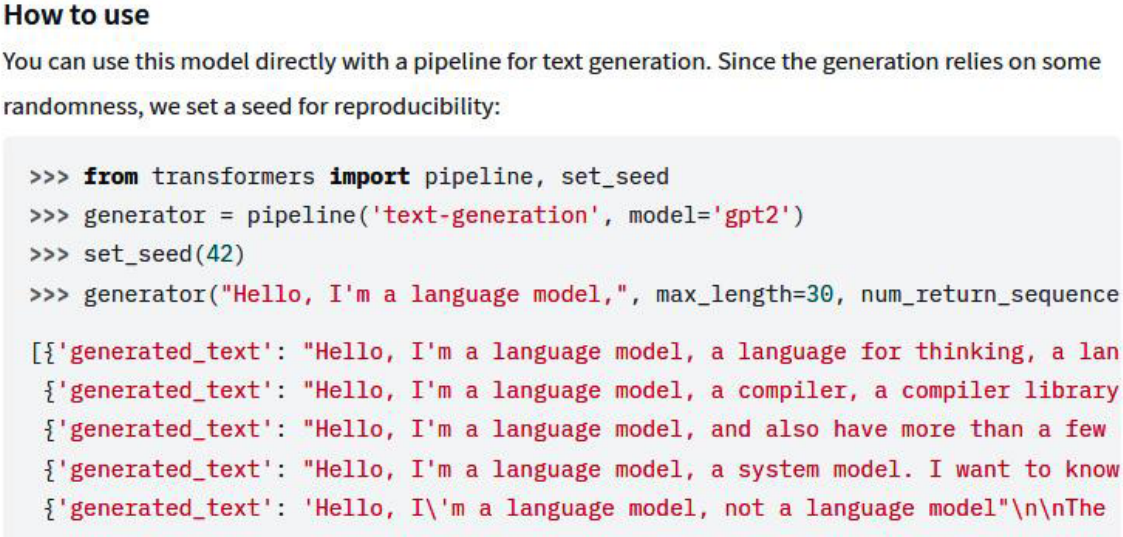

Figure 2.14 – Text-generation code example from the Hugging Face GPT-2 page

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 transformers 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)
```

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.'}
```

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 section, we will introduce the `datasets` library, which will be an essential utility in the upcoming experimental chapters.

Working with multimodal transformers

Multimodal transformers such as CLIP are very useful where more than one modality is involved. In order to provide a use-case for this kind of model, a very simple and straight forward example is given.

Zero-shot image classification can be very useful when only class names or phrases related to the classes are available. CLIP, which is a multimodal model, is able to represent images and texts in the same semantic space and can be very handy in this situation. To have a zero-shot image classifier with no prior knowledge about what classes can be or might be, imagine a case where class names are given with no examples for each of them. The only knowledge that is available at the moment is these names or maybe phrases and a set of images, which were not seen before.

1. The first thing that is needed to perform this experiment is to have a few images or download them from the internet:

   ```
   from PIL import Image
   import requests
   url = "http://images.cocodataset.org/test-
   stuff2017/000000000448.jpg"
   image = Image.open(requests.get(url, stream=True).raw)
   ```

2. Now that you have downloaded the image (a test-related image from the COCO dataset), you can easily see if you are using Jupyter Notebook or Colab by just running the following code:

   ```
   image
   ```

Figure 2.15 – A sample for testing CLIP (http://images.cocodataset.org/test-stuff2017/000000000448.jpg)

3. The next step is to just create a prompt to be added for all the classes and then create inputs for the text part:

```
prompt = "a photo of a "
class_names = ["fighting", "meeting"]
inputs = [prompt + class_name for class_name in class_names]
```

4. Up to this point, the inputs for text and image are ready, but we need to load the model:

```
from transformers import CLIPProcessor, CLIPModel
model = (CLIPModel.from_pretrained("openai/clip-vit-large-
patch14"))
processor = \
    (CLIPProcessor.from_pretrained("openai/clip-vit-large-
patch14"))
```

5. This processor is a wrapper that is very useful for tokenization. At the final step, you can hand the pre-processed and tokenized data to the model and finally get the output:

```
inputs = processor(text=inputs, images=image,
    return_tensors="pt", padding=True)
outputs = model(**inputs)
logits = outputs.logits_per_image
probs = logits_per_image.softmax(dim=1)
```

Logits have scores related to each combination (first prompt and image and second prompt and image) *10.9* and *18.5*. However, to get the final class probabilities, a final SoftMax is applied to it and will give the results, which are probabilities in our case (*0.99* for "meeting").

Now that we know the basics of multimodal transformers, let's see how we can use benchmarks and datasets in the next section.

Working with benchmarks and datasets

Thanks to the transformer architecture and transformer library, we are able to transfer what we have learned from a particular task to any other 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. We can apply it since current deep learning approaches allow us to solve many tasks at the same time, also known as **multitask learning** (**MTL**), or in order, also known as **sequential transfer learning** (**STL**). Benchmarking mechanisms test the extent to which these capabilities are possessed by the LM.

Before introducing the `datasets` library, it is worth talking about important benchmarks such as **General Language Understanding Evaluation** (**GLUE**), **Cross-lingual TRansfer Evaluation of Multilingual Encoders** (**XTREME**), and **Stanford Question Answering Dataset** (**SquAD**). Benchmarking is especially critical for evaluating transfer learnings within multitask and multilingual environments. In ML, we mostly tend to focus on a particular metric, which is a performance score on a certain task or dataset. With these benchmarks, we can measure the transfer learning capacity of the language models when we try to solve many tasks at the same time.

Let's start with the benchmarks.

Important benchmarks

In the following subsections, 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**

In order to use fewer pages, we only detail the tasks for the GLUE benchmark.