Chapter 9:Cross-Lingual and Multilingual Language Modeling

Up to this point, you have learned a lot about transformer-based architectures, from encoder-only models to decoder-only models, from efficient transformers to long-context transformers. You also learned about semantic text representation based on a Siamese network. However, we discussed all these models in terms of monolingual problems. We assumed that these models just understand a single language and are not capable of having a general understanding of text, regardless of the language itself. In fact, some of these models have multilingual variants; **Multilingual Bidirectional Encoder Representations from Transformers** (**mBERT**), **Multilingual Text-to-Text Transfer Transformer** (**mT5**), and **Multilingual Bidirectional and Auto-Regressive Transformer** (**mBART**), to name but a few. On the other hand, some models are specifically designed for multilingual purposes trained with cross-lingual objectives. For example, **Cross-lingual Language Model** (**XLM**) is such a method, and this will be described in detail in this chapter.

In this chapter, the concept of knowledge sharing between languages will be presented, and the impact of **Byte-Pair Encoding** (**BPE**) on the tokenization part is also another important subject to cover in order to achieve better input. Cross-lingual sentence similarity using the **Cross-Lingual Natural Language Inference** (**XNLI**) corpus will be detailed. Tasks such as cross-lingual classification and utilization of cross-lingual sentence representation for training on one language and testing on another one will be presented by concrete examples of real-life problems in **Natural Language Processing** (**NLP**), such as multilingual intent classification.

In short, you will learn the following topics in this chapter:

- Translation language modeling and cross-lingual knowledge sharing
- XLM and mBERT
- Cross-lingual similarity tasks
- Cross-lingual classification
- Cross-lingual zero-shot learning
- Fundamental limitations of multilingual models

Technical requirements

The code for this chapter is found in the repo at https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH09, which is in the GitHub repository for this book. We will be using Jupyter Notebook to run our coding exercises that require Python 3.6.0+, and the following packages will need to be installed:

```
tensorflow
pytorch
transformers >=4.00
datasets
sentence-transformers
umap-learn
openpyxl
```

Check out the following link to see the Code in Action video:

https://bit.ly/3zASz7M

Translation language modeling and cross-lingual knowledge sharing

So far, you have learned about **Masked Language Modeling** (**MLM**) as a cloze task. However, language modeling using neural networks is divided into three categories based on the approach itself and its practical usage, as follows:

MLM

Causal Language Modeling (CLM)

Translation Language Modeling (TLM)

It is also important to note that there are other pre-training approaches such as **Next Sentence Prediction (NSP)** and **Sentence Order Prediction (SOP)** too, but we just considered token-based language modeling. These three are the main approaches that are used in the literature. **MLM**, described and detailed in previous chapters, is a very close concept to a cloze task in language learning.

CLM is defined by predicting the next token, which is followed by some previous tokens. For example, if you see the following context, you can easily predict the next token:

<s> Transformers changed the natural language ...

As you see, only the last token is masked, and the previous tokens are given to the model to predict that last one. This token would be *processing* and if the context with this token is given to you again, you might end it with an *"</s>"* token. In order to have good training on this approach, it is required to not mask the first token, because the model would have just a sentence start token to make a sentence out of it. This sentence can be anything! Here's an example:

<s> ...

What would you predict out of this? It can be literally anything. To have better training and better results, it is required to give at least the first token, such as this:

<s> Transformers ...

And the model is required to predict the *change*; after giving it *Transformers changed ...* it is required to predict *the*, and so on. This approach is very similar to N-grams and **Long-Short-Term Memory (LSTM)**-based approaches because it is left-to-right modeling based on the probability $P(wn|wn-1, wn-2 ,\ldots,w0)$ where wn is the token to be predicted and the rest is the tokens before it. The token with the maximum probability is the predicted one.

These are the objectives used for monolingual models. So, what can be done for cross-lingual models? The answer is **TLM**, which is very similar to MLM, with a few changes. Instead of giving a sentence from a single language, a sentence pair is

given to a model in different languages, separated by a special token. The model is required to predict the masked tokens, which are randomly masked in any of these languages.

The following sentence pair is an example of such a task:

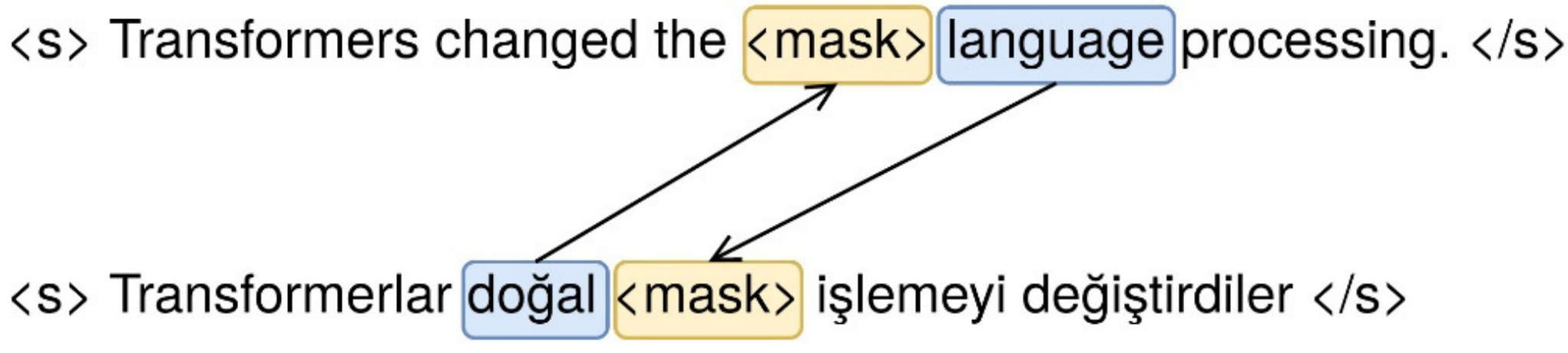

Figure 9.1 – Cross-lingual relation example between Turkish and English

Given these two masked sentences, the model is required to predict the missing tokens. In this task, on some occasions, the model has access to the tokens (for example, **doğal** and **language** respectively in the sentence pair in *Figure 9.1*) that are missing from one of the languages in the pair.

As another example, you can see the same pair from Persian and Turkish sentences. In the second sentence, the **değiştirdiler** token can be attended by the multiple tokens (one is masked) in the first sentence. In the following example, the word تغییر is missing but the meaning of **değiştirdiler** is تغییر دادند. :

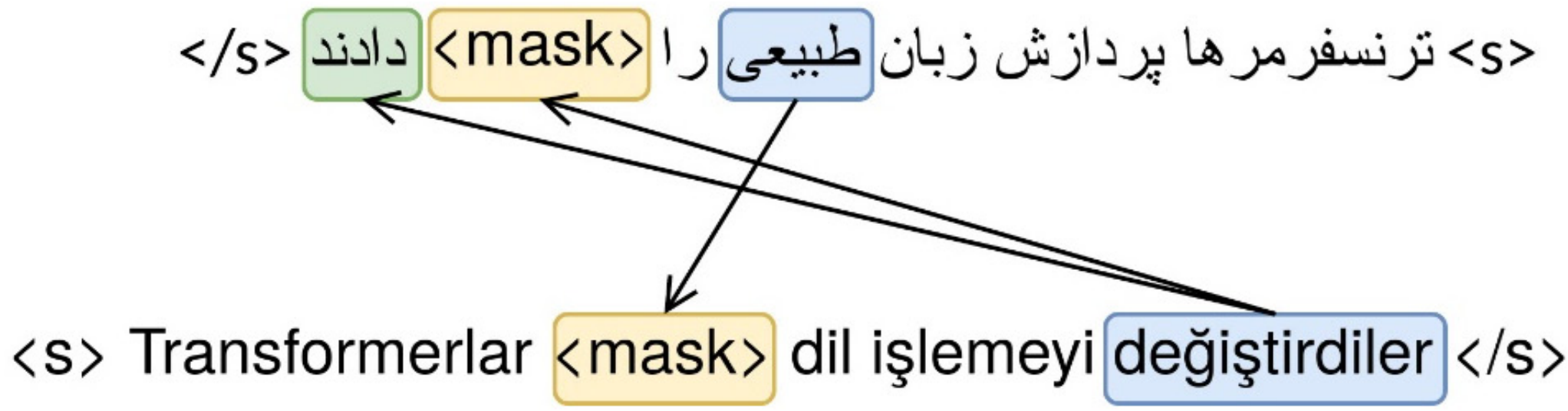

Figure 9.2 – Cross-lingual relation example between Persian and Turkish

Accordingly, a model can learn the mapping between these meanings. Just as with a translation model, our TLM must also learn these complexities between languages because **Machine Translation (MT)** is more than a token-to-token mapping.

XLM and mBERT

We have picked two models to explain in this section: mBERT and XLM. We selected these models because they correspond to the two best multilingual types as of writing this article. mBERT is a multilingual model trained on a different corpus of various languages using MLM modeling. It can operate separately for many languages. On the other hand, XLM is trained on different corpora using MLM, CLM, and TLM language modeling, and can solve cross-lingual tasks. For instance, it can measure the similarity of the sentences in two different languages by mapping them in a common vector space, which is not possible with mBERT.

mBERT

You are familiar with the BERT autoencoder model from *Chapter 3*, *Autoencoding Language Models*, and how to train it using MLM on a specified corpus. Imagine a case where a wide and huge corpus is provided not from a single language, but from 104 languages instead. Training on such a corpus would result in a multilingual version of BERT. However, training on such a wide variety of languages would increase the model size, and this is inevitable in the case of BERT. The vocabulary size would be increased and, accordingly, the size of the embedding layer would be larger because of more vocabulary.

Compared to a monolingual pre-trained BERT, this new version is capable of handling multiple languages inside a single model. However, the downside for this kind of modeling is that this model is not capable of mapping between languages. This means that the model, in the pre-training phase, does not learn anything about these mappings between semantic meanings of the tokens from different languages. In order to provide cross-lingual mapping and understanding for this model, it is necessary to train it on some of the cross-lingual supervised tasks, such as those available in the `XNLI` dataset.

Using this model is as easy as working with the models you have used in the previous chapters (see https://huggingface.co/bert-base-multilingual-uncased for more details). Here's the code you'll need to get started:

```
from transformers import pipeline
unmasker = pipeline('fill-mask', model='bert-base-
                    multilingual-uncased')
sentences = [
"Transformers changed the [MASK] language processing",
"Transformerlar [MASK] dil işlemeyi değiştirdiler",
"ترنسفرمرها [MASK] پردازش زبان را تغییر دادند"
]
for sentence in sentences:
    print(sentence)
    print(unmasker(sentence)[0]["sequence"])
    print("="*50)
```

The output will then be presented, as shown in the following code snippet:

```
Transformers changed the [MASK] language processing
transformers changed the english language processing
==================================================
Transformerlar [MASK] dil işlemeyi değiştirdiler
transformerlar bu dil islemeyi degistirdiler
==================================================
ترنسفرمرها [MASK] پردازش زبان را تغییر دادند
ترنسفرمرها پردازش زبانی را تغییر دادند
==================================================
```

As you can see, it can perform `fill-mask` for various languages.

XLM

Cross-lingual pre-training of language models, such as that shown with an XLM approach, is based on three different pre-training objectives. MLM, CLM, and TLM are used to pre-train the XLM model. The sequential order of this pre-training is performed using a shared BPE tokenizer between all languages. The reason that tokens are shared is that the shared tokens provide fewer tokens in the case of languages that have similar tokens or subwords, and on the other hand, these tokens can provide shared semantics in the pre-training process. For example, some tokens have remarkably similar writing and meaning across many languages, and accordingly, these tokens are shared by BPE for all. On the other hand, some tokens spelled the same in different languages can have different meanings—for example, *was* is shared in German and English contexts. Luckily, self-attention mechanisms help us to disambiguate the meaning of *was* using the surrounding context.

Another major improvement of this cross-lingual modeling is that it is also pre-trained on CLM, which makes it more reasonable for inferences where sentence prediction or completion is required. In other words, this model has an understanding of the languages and is capable of completing sentences, predicting missing tokens, and predicting missing tokens by using the other language source.

The following diagram shows the overall structure of cross-lingual modeling. You can read more at https://arxiv.org/pdf/1901.07291.pdf:

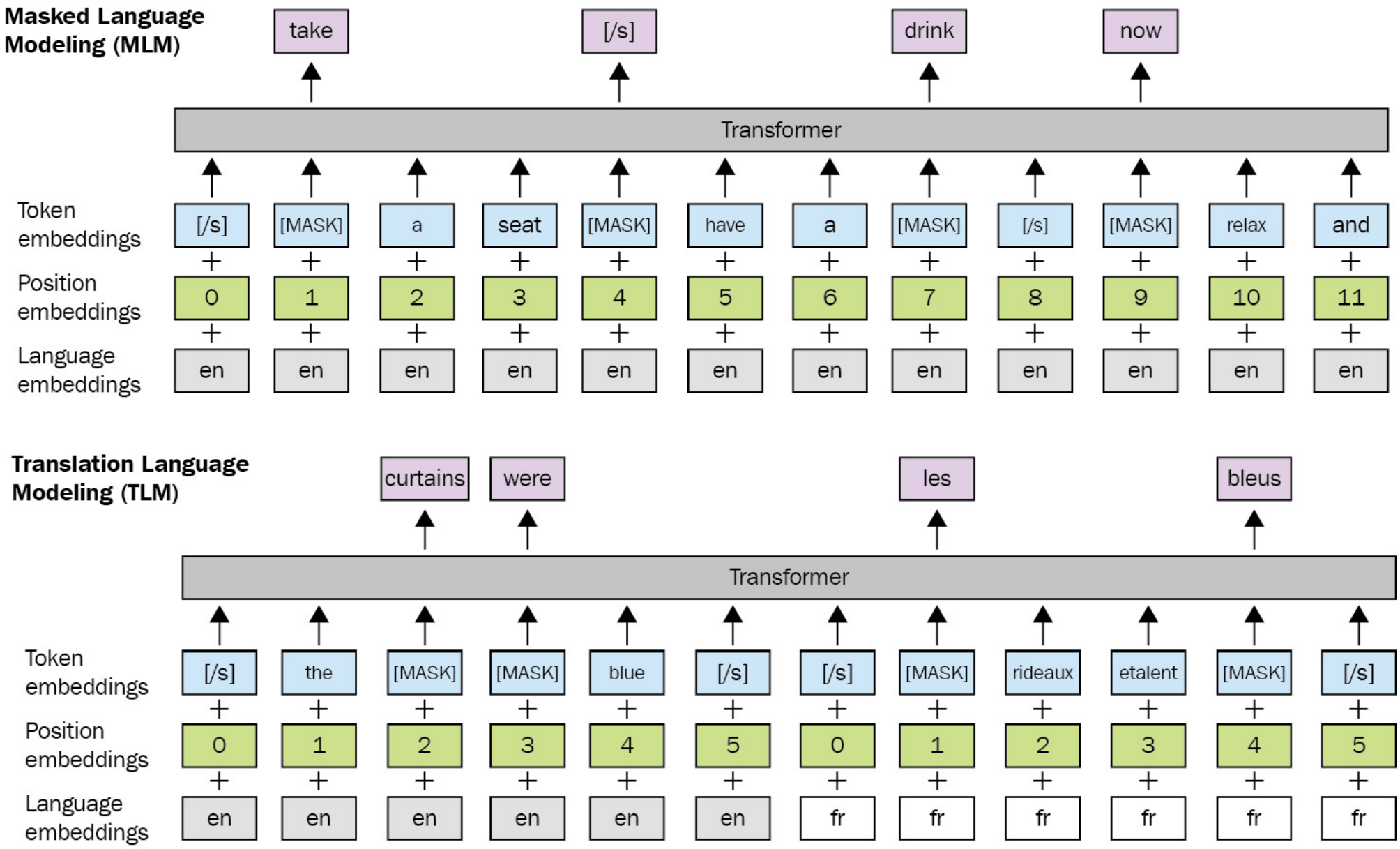

Figure 9.3 – MLM and TLM pre-training for cross-lingual modeling

A newer version of the XLM model is also released as **XLM-R**, which has minor changes in the training and corpus used. XLM-R is identical to the XLM model but is trained on more languages and a much bigger corpus. The **CommonCrawl** and **Wikipedia** corpus is aggregated, and the XLM-R is trained for MLM on it. However, the XNLI dataset is also used for TLM. The following diagram shows the amount of data used by XLM-R pre-training:

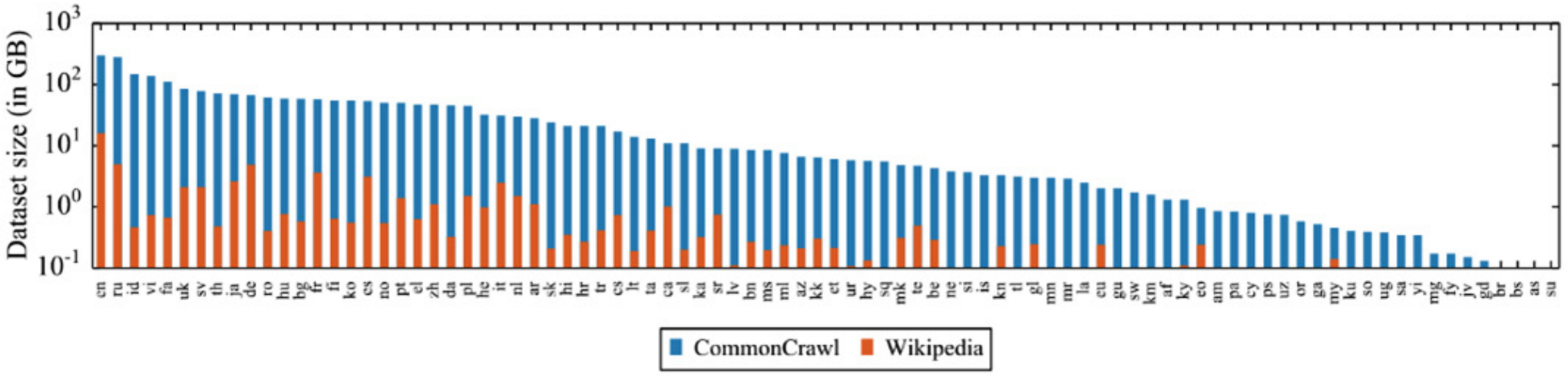

Figure 9.4 – Amount of data in gigabytes (GB) (log-scale)

There are many upsides and downsides when adding new languages for training data — for example, adding new languages may not always improve the overall model of **Natural Language Inference (NLI)**. The **XNLI dataset** is usually used for multilingual and cross-lingual NLI. From previous chapters, you have seen the **Multi-Genre NLI (MNLI)** dataset for English; the XNLI dataset is almost identical to it but has more languages, and it also has sentence pairs. However, training only on this task is not enough, and it will not cover TLM pre-training. For TLM pre-training, much broader datasets such as the parallel corpus of **OPUS** (short for **Open Source Parallel Corpus**) are used. This dataset contains subtitles from different languages, aligned and cleaned, with the translations provided by many software sources such as Ubuntu, and so on.

The following screenshot shows OPUS (https://opus.nlpl.eu/trac/) and its components for searching and getting information about the dataset:

Figure 9.5 – OPUS

Figure 9.5 – OPUS

The steps for using cross-lingual models are described here:

1. Simple changes to the previous code can show you how XLM-R performs mask filling. First, you must change the model, as follows:

```
unmasker = pipeline('fill-mask', model='xlm-roberta-base')
```

2. Afterward, you need to change the mask token from `[MASK]` to `<mask>`, which is a special token for XLM-R (or simply call `tokenizer.mask_token`). Here's the code to accomplish this:

```
sentences = [
"Transformers changed the <mask> language processing",
"Transformerlar <mask> dil işlemeyi değiştirdiler",
"ترنسفرمرها پردازش زبان <mask>" را تغییر دادند"
]
```

3. Then, you can run the same code, as follows:

```
for sentence in sentences:
  print(sentence)
  print(unmasker(sentence)[0]["sequence"])
  print("="*50)
```

4. The results will appear, like so:

```
Transformers changed the <mask> language processing
Transformers changed the human language processing
==================================================
Transformerlar <mask> dil işlemeyi değiştirdiler
Transformerlar, dil işlemeyi değiştirdiler
================================================== ترنسفرمرها پردازش زبان [MASK]  را تغییر
دادند
 ترنسفرمرها پردازش زبانی را تغییر دادند
==================================================
```

5. But as you see from the Turkish and Persian examples, the model still made mistakes; for example, in the Persian text, it just added ی, and in the Turkish version, it added ,. For the English sentence, it added human, which is not what was expected. The sentences are not wrong, but not what we expected. However, this time, we have a cross-lingual model that is trained using TLM; so, let's use it by concatenating two sentences and giving the model some extra hints. Here we go:

```
print(unmasker("Transformers changed the natural language processing. </s>
Transformerlar <mask> dil işlemeyi değiştirdiler.")[0]["sequence"])
```

6. The results will be shown, as follows:

```
Transformers changed the natural language processing. Transformerlar doğal dil
işlemeyi değiştirdiler.
```

7. That's it! The model has now made the right choice. Let's play with it a bit more and see how it performs, as follows:

```
print(unmasker("Earth is a great place to live in. </s> زمین جای خوبی برای <mask>  کردن است.")[0]["sequence"])
```

Here is the result:

```
Earth is a great place to live in. زمین جای خوبی برای زندگی کردن است.
```

Well done! So far, you have learned about multilingual and cross-lingual models such as mBERT and XLM. In the next section, you will learn how to use such models for multilingual text similarity. You will also see some use cases, such as multilingual plagiarism detection.

Cross-lingual similarity tasks

Cross-lingual models are capable of representing text in a unified form, where sentences are from different languages but those with close meaning are mapped to similar vectors in vector space. XLM-R, as was detailed in the previous section, is one of the successful models in this scope. Now, let's look at some applications on this.

Cross-lingual text similarity

In the following example, you will see how it is possible to use a cross-lingual language model pre-trained on the XNLI dataset to find similar texts from different languages. A use-case scenario is where a plagiarism detection system is required for this task. We will use sentences from the Azerbaijani language and see whether XLM-R finds similar sentences from English—if there are any. The sentences from both languages are identical. Here are the steps to take:

1. First, you need to load a model for this task, as follows:

```
from sentence_transformers import SentenceTransformer, util
model = SentenceTransformer("stsb-xlm-r-multilingual")
```

2. Afterward, we assume that we have sentences ready in the form of two separate lists, as illustrated in the following code snippet:

```
azeri_sentences = ['Pişik çöldə oturur',
              'Bir adam gitara çalır',
              'Mən makaron sevirəm',
              'Yeni film möhtəşəmdir',
              'Pişik bağda oynayır',
              'Bir qadın televizora baxır',
              'Yeni film çox möhtəşəmdir',
              'Pizzanı sevirsən?']
english_sentences = ['The cat sits outside',
             'A man is playing guitar',
             'I love pasta',
             'The new movie is awesome',
             'The cat plays in the garden',
             'A woman watches TV',
             'The new movie is so great',
             'Do you like pizza?']
```

3. And the next step is to represent these sentences in vector space by using the XLM-R model. You can do this by simply using the `encode` function of the model, as follows:

```
azeri_representation = model.encode(azeri_sentences)
english_representation = \
model.encode(english_sentences)
```

4. At the final step, we will search for semantically similar sentences of the first language on the other language's representations, as follows:

```
results = []
for azeri_sentence, query in zip(azeri_sentences, azeri_representation):
  id_, score = util.semantic_search(
          query,english_representation)[0][0].values()
  results.append({
      "azeri": azeri_sentence,
      "english": english_sentences[id_],
      "score": round(score, 4)
  })
```

5. In order to see a clear form of these results, you can use a pandas DataFrame, as follows:

```
import pandas as pd
pd.DataFrame(results)
```

And you will see the results with their matching score, as follows:

	azeri	english	score
0	Pişik çöldə oturur	The cat sits outside	0.5969
1	Bir adam gitara çalır	A man is playing guitar	0.9939
2	Mən makaron sevirəm	I love pasta	0.6878
3	Yeni film möhtəşəmdir	The new movie is so great	0.9757
4	Pişik bağda oynayır	A man is playing guitar	0.2695
5	Bir qadın televizora baxır	A woman watches TV	0.9946
6	Yeni film çox möhtəşəmdir	The new movie is so great	0.9797
7	Pizzanı sevirsən?	Do you like pizza?	0.9894

Figure 9.6 – Plagiarism detection results (XLM-R)

The model made mistakes in one case (row number *4*) if we accept the maximum scored sentence to be paraphrased or translated, but it is useful to have a threshold and accept values higher than it. We will show more comprehensive experimentation in the following sections.

On the other hand, there are alternative bi-encoders available too. Such approaches provide a pair encoding of two sentences and classify the result to train the model. In such cases, **Language-Agnostic BERT Sentence Embedding** (**LaBSE**) may be a good choice too, and it is available in the **sentence-transformers** library and in **TensorFlow Hub** too. LaBSE is a dual encoder based on Transformers, which is similar to Sentence-BERT, where two encoders that have the same parameters are combined with a loss function based on the dual similarity of two sentences.

Using the same example, you can change the model to LaBSE in a very simple way and rerun the previous code (*Step 1*), as follows:

```
model = SentenceTransformer("LaBSE")
```

The results are shown in the following screenshot:

	azeri	english	score
0	Pişik çöldə oturur	The cat sits outside	0.8686
1	Bir adam gitara çalır	A man is playing guitar	0.9143
2	Mən makaron sevirəm	I love pasta	0.8888
3	Yeni film möhtəşəmdir	The new movie is so great	0.9107
4	Pişik bağda oynayır	The cat sits outside	0.6761
5	Bir qadın televizora baxır	A woman watches TV	0.9359
6	Yeni film çox möhtəşəmdir	The new movie is so great	0.9258
7	Pizzanı sevirsən?	Do you like pizza?	0.9366

Figure 9.7 – Plagiarism detection results (LaBSE)

As you see, LaBSE performs better in this case, and the result in row number *4* is correct this time. LaBSE authors claim that it works very well in finding translations of sentences, but it is not so good at finding sentences that are not completely identical. For this purpose, it is a very useful tool for finding plagiarism in cases where a translation is used to steal intellectual material. However, there are many other factors that change the results too—for example, the resource size for the pre-trained model in each language and the nature of the language pairs is also important. For a reasonable comparison, we need a more comprehensive experiment, and we should consider many factors.

Visualizing cross-lingual textual similarity

Now, we will measure and visualize the degree of textual similarity between two sentences, one of which is a translation of the other. **Tatoeba** is a free collection of such sentences and translations, and it is part of the XTREME benchmark. The community aims to get high-quality sentence translation with the support of many participants. We'll now take the following steps:

1. We will get Russian and English sentences out of this collection. Make sure the following libraries are installed before you start working:

```
!pip install sentence_transformers datasets transformers umap-learn
```

2. Load the sentence pairs, as follows:

```
from datasets import load_dataset
import pandas as pd
data=load_dataset("xtreme","tatoeba.rus",
                  split="validation")
pd.DataFrame(data)[["source_sentence","target_sentence"]]
```

Let's look at the output, as follows:

	source_sentence	target_sentence
0	Я знаю много людей, у которых нет прав.\n	I know a lot of people who don't have driver's...
1	У меня много знакомых, которые не умеют играть...	I know a lot of people who don't know how to p...
2	Мой начальник отпустил меня сегодня пораньше.\n	My boss let me leave early today.\n
3	Я загорел на пляже.\n	I tanned myself on the beach.\n
4	Вы сегодня проверяли почту?\n	Have you checked your email today?\n
...	...	...
995	Что сказал врач?\n	What did the doctor say?\n
996	Я рад, что ты сегодня здесь.\n	I'm glad you're here today.\n
997	Фермеры пригнали в деревню пять волов, девять ...	The farmers had brought five oxen and nine cow...
998	Жужжание пчёл заставляет меня немного нервнича...	The buzzing of the bees makes me a little nerv...
999	С каждым годом они становились всё беднее.\n	From year to year they were growing poorer.\n

1000 rows × 2 columns

Figure 9.8 – Russian-English sentence pairs

3. First, we will take the first *K=30* sentence pairs for visualization, and later, we will run an experiment for the entire set. Now, we will encode them with sentence transformers that we already used for the previous example. Here is the execution of the code:

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer("stsb-xlm-r-multilingual")
K=30
q=data["source_sentence"][:K] + data["target_sentence"][:K]
emb=model.encode(q)
len(emb), len(emb[0])
Output: (60, 768)
```

4. We now have 60 vectors of length 768. We will reduce the dimensionality to 2 with **Uniform Manifold Approximation and Projection** (**UMAP**), which we have already encountered in previous chapters. We visualized sentences that are translations of each other, marking them with the same color and code. We also

drew a dashed line between them to make the link more obvious. The code is illustrated in the following snippet:

Copy Explain

```
import matplotlib.pyplot as plt
import numpy as np
import umap
import pylab
X= umap.UMAP(n_components=2, random_state=42).fit_transform(emb)
idx= np.arange(len(emb))
fig, ax = plt.subplots(figsize=(12, 12))
ax.set_facecolor('whitesmoke')
cm = pylab.get_cmap("prism")
colors = list(cm(1.0*i/K) for i in range(K))
for i in idx:
    if i<K:
        ax.annotate("RUS-"+str(i), # text
                    (X[i,0], X[i,1]), # coordinates
                    c=colors[i]) # color
    ax.plot((X[i,0],X[i+K,0]),(X[i,1],X[i+K,1]),"k:")
else:
    ax.annotate("EN-"+str(i%K),
                    (X[i,0], X[i,1]),
                    c=colors[i%K])
```

Here is the output of the preceding code:

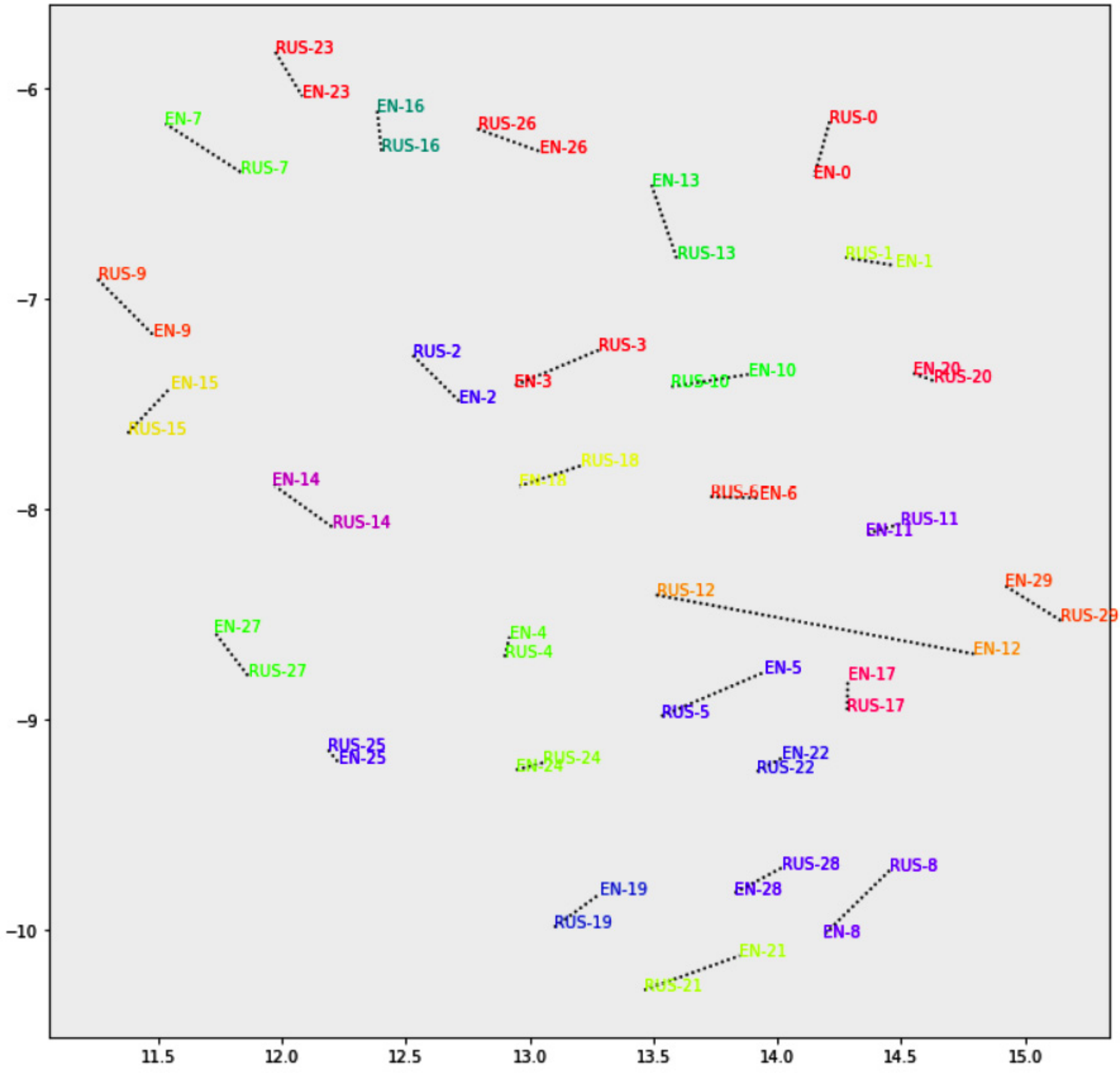

Figure 9.9 – Russian-English sentence similarity visualization

As we expected, most sentence pairs are located close to each other. Inevitably, some certain pairs (such as `id 12`) insist on not getting close.

5. For a comprehensive analysis, let's now measure the entire dataset. We encode all of the source and target sentences—1K pairs—as follows:

```
source_emb=model.encode(data["source_sentence"])
target_emb=model.encode(data["target_sentence"])
```

6. We calculate the cosine similarity between all pairs, save them in the `sims` variable, and plot a histogram, as follows:

```
from scipy import spatial
sims=[ 1 - spatial.distance.cosine(s,t) \
        for s,t in zip(source_emb, target_emb)]
plt.hist(sims, bins=100, range=(0.8,1))
plt.show()
```

Here is the output:

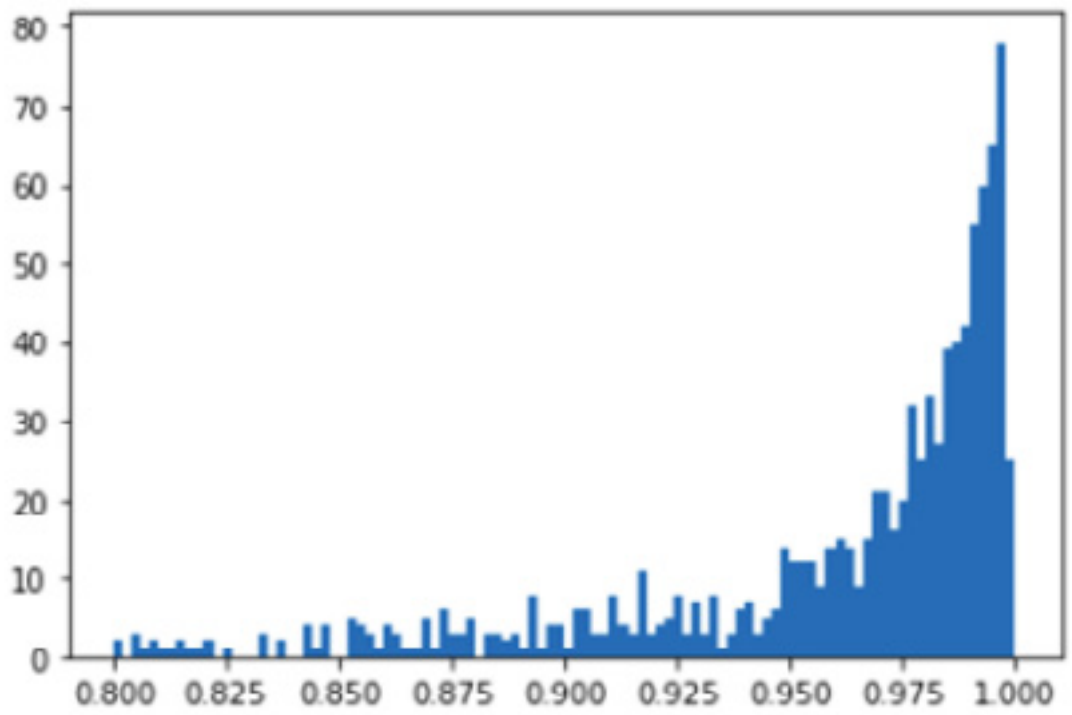

Figure 9.10 – Similarity histogram for the English and Russian sentence pairs

7. As can be seen, the scores are very close to 1. This is what we expect from a good cross-lingual model. The mean and standard deviation of all similarity measurements also support the cross-lingual model performance, as follows:

```
>>> np.mean(sims), np.std(sims)
(0.946, 0.082)
```

8. You can run the same code yourself for languages other than Russian. As you run it with **French** (`fra`), **Tamil** (`tam`), and so on, you will get the following resulting table. The table indicates that you will see in your experiment that the model works well in many languages but fails in others, such as **Afrikaans** or **Tamil**:

Language	Code	Mean	Std
French	fra	0.94	0.087
Afrikaans	afr	0.79	0.18
Arabic	ara	0.94	0.08
Korean	kor	0.92	0.11
Tamil	tam	0.77	0.19

Table 1 – Cross-lingual model performance for other languages

In this section, we applied cross-lingual models to measure similarity between different languages. In the next section, we'll make use of cross-lingual models in a supervised way.

Cross-lingual classification

So far, you have learned that cross-lingual models are capable of understanding different languages in semantic vector space where similar sentences, regardless of their language, are close in terms of vector distance. But how it is possible to use this capability in use cases where we have few samples available?

For example, you are trying to develop an intent classification for a chatbot in which there are few samples or no samples available for the second language; but for the first language—let's say English—you do have enough samples. In such cases, it is possible to freeze the cross-lingual model itself and just train a classifier for the task. A trained classifier can be tested on a second language instead of the language it is trained on.

In this section, you will learn how to train a cross-lingual model in English for text classification and test it in other languages. We have selected a very low-resource language known as **Khmer** (https://en.wikipedia.org/wiki/Khmer_language), which is spoken by 16 million people in Cambodia, Thailand, and Vietnam. It has few resources on the internet, and it is hard to find good datasets to train your model on

it. However, we have access to a good **Internet Movie Database** (**IMDb**) sentiment dataset of movie reviews for sentiment analysis. We will use that dataset to find out how our model performs on the language it is not trained on.

The following diagram nicely depicts the kind of flow we will follow. The model is trained with train data on the left, and this model is applied to the test sets on the right. Please notice that MT and sentence-encoder mappings play a significant role in the flow:

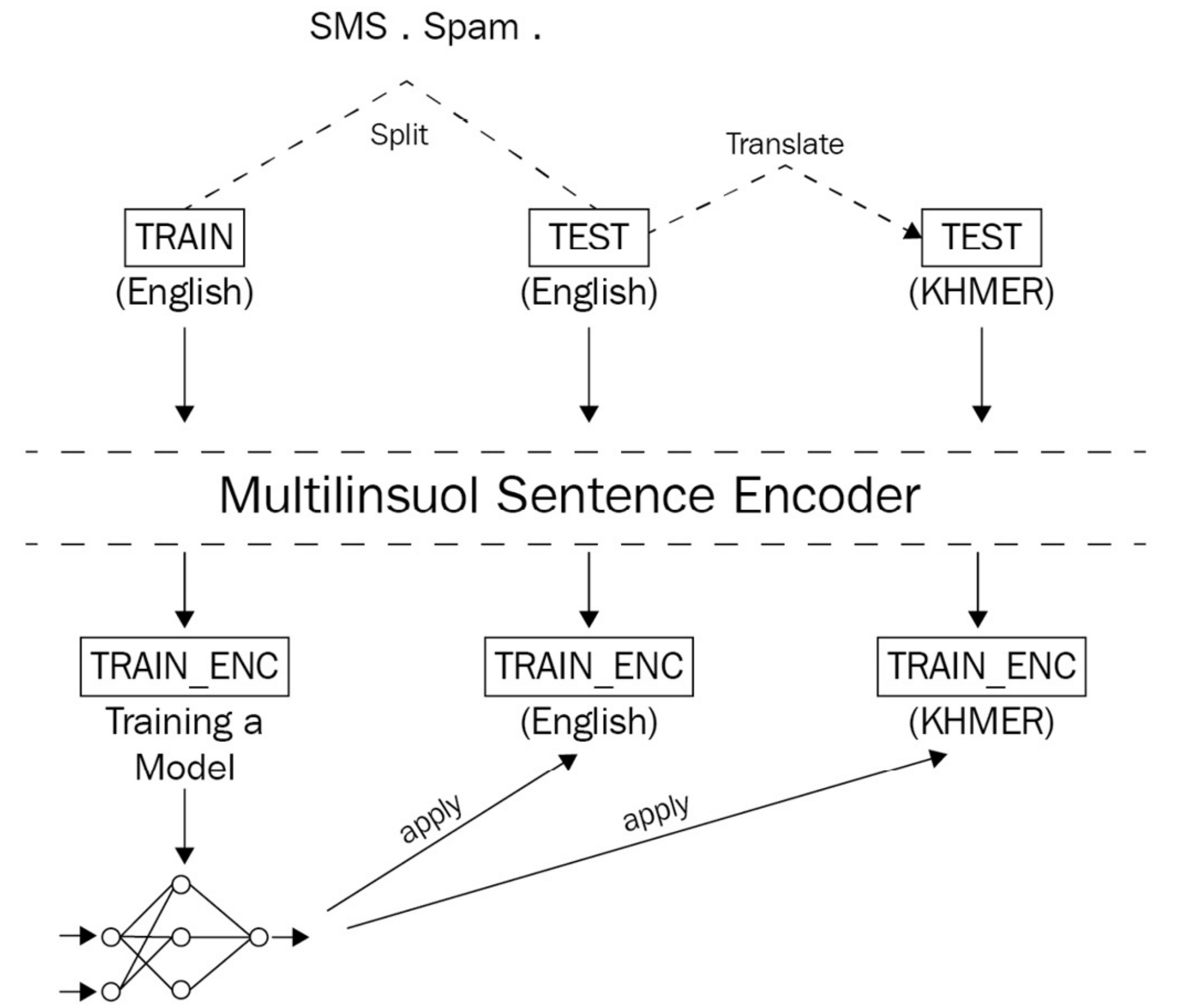

Figure 9.11 – Flow of cross-lingual classification

The required steps to load and train a model for cross-lingual testing are outlined here:

1. The first step is to load the dataset, as follows:

```
from datasets import load_dataset
sms_spam = load_dataset("imdb")
```

2. You need to shuffle the dataset to shuffle the samples before using them, as follows:

```
imdb = imdb.shuffle()
```

3. The next step is to make a good test split out of this dataset, which is in the Khmer language. In order to do so, you can use a translation service such as Google Translate. First, you should save this dataset in Excel format, as follows:

```
imdb_x = [x for x in imdb['train'][:1000]['text']]
labels = [x for x in imdb['train'][:1000]['label']]
import pandas as pd
pd.DataFrame(imdb_x,
             columns=["text"]).to_excel(
                                "imdb.xlsx",
                                 index=None)
```

4. Afterward, you can upload it to Google Translate and get the Khmer translation of this dataset (https://translate.google.com/?sl=en&tl=km&op=docs), as illustrated in the following screenshot:

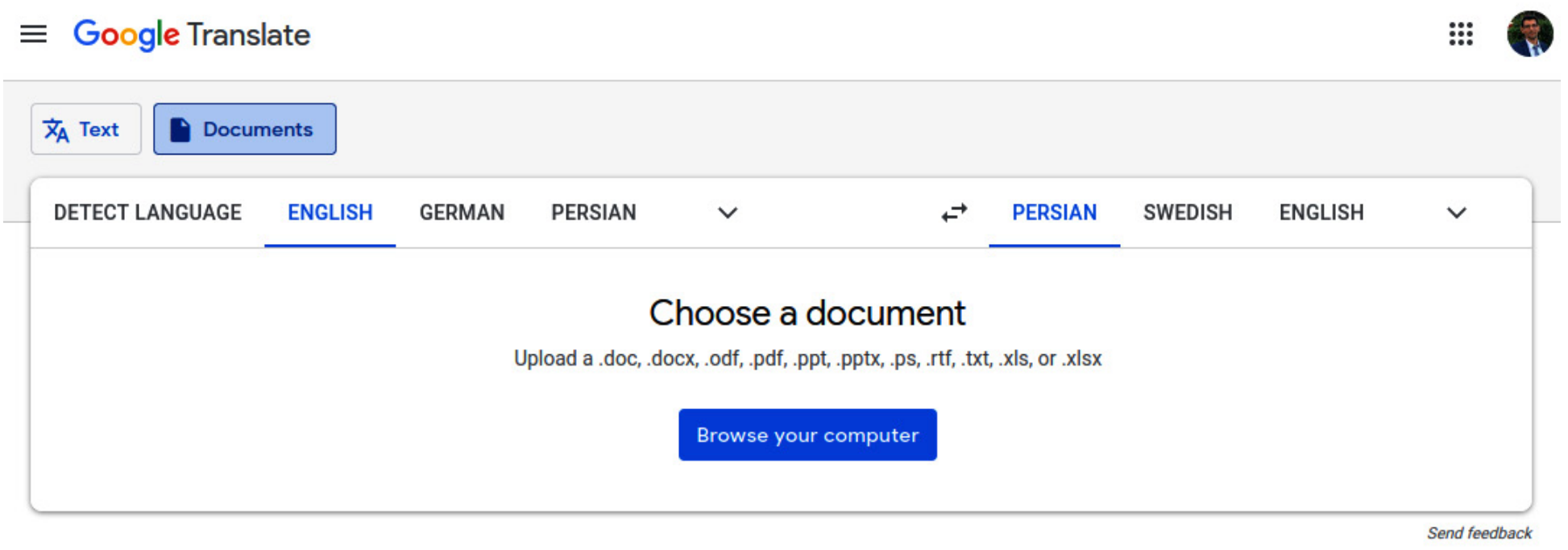

Figure 9.12 – Google document translator

5. After selecting and uploading the document, it will give you the translated version in Khmer, which you can copy and paste into an Excel file. It is also required to save it in Excel format again. The result would be an Excel document that is a translation of the original spam/ham English dataset. You can read it using pandas by running the following command:

```
pd.read_excel("KHMER.xlsx")
```

And the result will be seen, as follows:

	text
0	ទៅរហូតដល់ចំណុចចុងឆ្កួត .. អាចប្រើបានតែនៅបែបហ...
1	អ្នកឡូ ... ចូជី wif u oni ...
2	ធ្វើសារដោយឥតគិតថ្លៃក្នុងលេខ ២ ដោយឥតគិតថ្លៃដើម...
3	U dun និយាយអញ្ចឹងមុនគេ ... U c និយាយរួចទៅហើយ ...។
4	ទេខ្ញុំមិនគិតថាគាត់ទៅរកយើងទេគាត់រស់នៅទីនេះ
...	...
5569	នេះជាលើកទី ២ ហើយដែលយើងបានព្យាយាមទំនាក់ទំនង ២ ។...
5570	តើប៉ុន្មានទៅផ្ទះ esplanade fr?
5571	គួរឱ្យអាណិត, * គឺនៅក្នុងអារម្មណ៍សម្រាប់រឿងនោះ។...
5572	បុរសម្នាក់នេះបានធ្វើឱ្យចែចង់ខ្លះប៉ុន្តែខ្ញុំធ...
5573	Rofl ។ វាជាឈ្មោះពិត

5574 rows × 1 columns

Figure 9.13 – IMDB dataset in KHMER language.

6. However, it is required to get only text, so you should use the following code:

```
imdb_khmer = list(pd.read_excel("KHMER.xlsx").text)
```

7. Now that you have text for both languages and the labels, you can split the train and test validations, as follows:

```
from sklearn.model_selection import train_test_split
train_x, test_x, train_y, test_y, khmer_train, khmer_test =
train_test_split(imdb_x, labels, imdb_khmer, test_size = 0.2, random_state = 1)
```

8. The next step is to provide the representation of these sentences using the XLM-R cross-lingual model. First, you should load the model, as follows:

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer("stsb-xlm-r-multilingual")
```

9. And now, you can get the representations, like this:

```
encoded_train = model.encode(train_x)
encoded_test = model.encode(test_x)
encoded_khmer_test = model.encode(khmer_test)
```

10. But you should not forget to convert the labels to `numpy` format because TensorFlow and Keras only deal with `numpy` arrays when using the `fit` function of the Keras models. Here's how to do it:

```
import numpy as np
train_y = np.array(train_y)
test_y = np.array(test_y)
```

11. Now that everything is ready, let's make a very simple model for classifying the representations, as follows:

```
import tensorflow as tf
input_ = tf.keras.layers.Input((768,))
classification = tf.keras.layers.Dense(
                     1,
                    activation="sigmoid")(input_)
classification_model = \
          tf.keras.Model(input_, classification)
classification_model.compile(
         loss=tf.keras.losses.BinaryCrossentropy(),
         optimizer="Adam",
         metrics=["accuracy", "Precision", "Recall"])
```

12. You can fit your model using the following function:

```
classification_model.fit(
                    x = encoded_train,
                    y = train_y,
               validation_data=(encoded_test, test_y),
                    epochs = 10)
```

13. And the results for `20` epochs of training are shown, as follows:

```
Epoch 1/20
25/25 [==============================] - 4s 15ms/step - loss: 0.6512 - accuracy: 0.6025 - precision: 0.6229 - recall: 0.6110 - val_loss: 0.5918 - val_accuracy: 0.7100 - val_precision: 0.7257 - val_recall: 0.7523
Epoch 2/20
25/25 [==============================] - 0s 4ms/step - loss: 0.5478 - accuracy: 0.7362 - precision: 0.7321 - recall: 0.7828 - val_loss: 0.5429 - val_accuracy: 0.7100 - val_precision: 0.7684 - val_recall: 0.6697
Epoch 3/20
25/25 [==============================] - 0s 4ms/step - loss: 0.5097 - accuracy: 0.7638 - precision: 0.7738 - recall: 0.7757 - val_loss: 0.5219 - val_accuracy: 0.7150 - val_precision: 0.7653 - val_recall: 0.6881
Epoch 4/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4894 - accuracy: 0.7912 - precision: 0.7986 - recall: 0.8043 - val_loss: 0.5150 - val_accuracy: 0.7150 - val_precision: 0.7653 - val_recall: 0.6881
Epoch 5/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4749 - accuracy: 0.7912 - precision: 0.8000 - recall: 0.8019 - val_loss: 0.5065 - val_accuracy: 0.7350 - val_precision: 0.7745 - val_recall: 0.7248
Epoch 6/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4613 - accuracy: 0.7925 - precision: 0.8048 - recall: 0.7971 - val_loss: 0.5006 - val_accuracy: 0.7550 - val_precision: 0.7727 - val_recall: 0.7798
Epoch 7/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4543 - accuracy: 0.8062 - precision: 0.8028 - recall: 0.8353 - val_loss: 0.5122 - val_accuracy: 0.7150 - val_precision: 0.7708 - val_recall: 0.6789
Epoch 8/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4468 - accuracy: 0.8000 - precision: 0.8076 - recall: 0.8115 - val_loss: 0.4996 - val_accuracy: 0.7450 - val_precision: 0.7685 - val_recall: 0.7615
Epoch 9/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4429 - accuracy: 0.8037 - precision: 0.8032 - recall: 0.8282 - val_loss: 0.5032 - val_accuracy: 0.7350 - val_precision: 0.7692 - val_recall: 0.7339
Epoch 10/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4335 - accuracy: 0.8100 - precision: 0.8141 - recall: 0.8258 - val_loss: 0.5043 - val_accuracy: 0.7250 - val_precision: 0.7596 - val_recall: 0.7248
Epoch 11/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4300 - accuracy: 0.8087 - precision: 0.8107 - recall: 0.8282 - val_loss: 0.5091 - val_accuracy: 0.7250 - val_precision: 0.7647 - val_recall: 0.7156
Epoch 12/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4270 - accuracy: 0.8200 - precision: 0.8313 - recall: 0.8234 - val_loss: 0.5166 - val_accuracy: 0.7250 - val_precision: 0.7755 - val_recall: 0.6972
Epoch 13/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4216 - accuracy: 0.8175 - precision: 0.8212 - recall: 0.8329 - val_loss: 0.5107 - val_accuracy: 0.7200 - val_precision: 0.7624 - val_recall: 0.7064
Epoch 14/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4166 - accuracy: 0.8150 - precision: 0.8173 - recall: 0.8329 - val_loss: 0.5174 - val_accuracy: 0.7150 - val_precision: 0.7600 - val_recall: 0.6972
Epoch 15/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4124 - accuracy: 0.8150 - precision: 0.8219 - recall: 0.8258 - val_loss: 0.5085 - val_accuracy: 0.7300 - val_precision: 0.7570 - val_recall: 0.7431
Epoch 16/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4093 - accuracy: 0.8225 - precision: 0.8274 - recall: 0.8353 - val_loss: 0.5120 - val_accuracy: 0.7100 - val_precision: 0.7476 - val_recall: 0.7064
Epoch 17/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4053 - accuracy: 0.8175 - precision: 0.8182 - recall: 0.8377 - val_loss: 0.5116 - val_accuracy: 0.7150 - val_precision: 0.7500 - val_recall: 0.7156
Epoch 18/20
25/25 [==============================] - 0s 4ms/step - loss: 0.4057 - accuracy: 0.8150 - precision: 0.8265 - recall: 0.8186 - val_loss: 0.5129 - val_accuracy: 0.7350 - val_precision: 0.7593 - val_recall: 0.7523
Epoch 19/20
25/25 [==============================] - 0s 4ms/step - loss: 0.3981 - accuracy: 0.8313 - precision: 0.8365 - recall: 0.8425 - val_loss: 0.5174 - val_accuracy: 0.7250 - val_precision: 0.7596 - val_recall: 0.7248
Epoch 20/20
25/25 [==============================] - 0s 4ms/step - loss: 0.3962 - accuracy: 0.8375 - precision: 0.8416 - recall: 0.8496 - val_loss: 0.5226 - val_accuracy: 0.7150 - val_precision: 0.7600 - val_recall: 0.6972
<tensorflow.python.keras.callbacks.History at 0x7f8d141fdc10>
```

```
val_loss: 0.5226 - val_accuracy: 0.7150 - val_precision: 0.7600 - val_recall: 0.6972
```

Figure 9.14 – Training results on the English version of the IMDb dataset

14. As you have seen, we used an English test set to see the model performance across epochs, and it is reported as follows in the final epoch:

```
val_loss: 0.5226
val_accuracy: 0.7150
val_precision: 0.7600
val_recall: 0.6972
```

15. Now we have trained our model and tested it on English, let's test it on the Khmer test set, as our model never saw any of the samples either in English or in Khmer. Here's the code to accomplish this:

```
classification_model.evaluate(x = encoded_khmer_test,
                              y = test_y)
```

Here are the results:

```
loss: 0.5949
accuracy: 0.7250
precision: 0.7014
recall: 0.8623
```

So far, you have learned how it is possible to leverage the capabilities of cross-lingual models in low-resource languages. It makes a huge impact and difference when you can use such a capability in cases where there are very few samples or no samples to

train the model on. In the next section, you will learn how it is possible to use zero-shot learning where there are no samples available, even for high-resource languages such as English.

Cross-lingual zero-shot learning

In previous sections, you learned how to perform zero-shot text classification using monolingual models. Using XLM-R for multilingual and cross-lingual zero-shot classification is identical to the approach and code used previously, so we will use **mT5** here.

mT5, which is a massively multilingual pre-trained language model, is based on the encoder-decoder architecture of Transformers and is also identical to **T5**. T5 is pre-trained on English and mT5 is trained on 101 languages from **Multilingual Common Crawl (mC4)**.

The fine-tuned version of mT5 on the XNLI dataset is available from the HuggingFace repository (https://huggingface.co/alan-turing-institute/mt5-large-finetuned-mnli-xtreme-xnli).

The T5 model and its variant, mT5, is a completely text-to-text model, which means it will produce text for any task it is given, even if the task is classification or NLI. So, in the case of inferring this model, extra steps are required. We'll take the following steps:

1. The first step is to load the model and the tokenizer, as follows:

Copy Explain

```
from torch.nn.functional import softmax
from transformers import\
    MT5ForConditionalGeneration, MT5Tokenizer
model_name = "alan-turing-institute/mt5-large-finetuned-mnli-xtreme-xnli"
tokenizer = MT5Tokenizer.from_pretrained(model_name)
model = MT5ForConditionalGeneration\
    .from_pretrained(model_name)
```

2. In the next step, let's provide samples to be used in zero-shot classification—a sentence and labels, as follows:

```
sequence_to_classify = \
    "Wen werden Sie bei der nächsten Wahl wählen? "
candidate_labels = ["spor", "ekonomi", "politika"]
hypothesis_template = "Dieses Beispiel ist {}."
```

As you see, the sequence itself is in German (`"Who will you vote for in the next election?"`) but the labels are written in Turkish (`"spor"`, `"ekonomi"`, `"politika"`). The `hypothesis_template` says: `"this example is ..."` in German.

3. The next step is to set the label **identifiers** (`IDs`) of the entailment, `CONTRADICTS`, and `NEUTRAL`, which will be used later in inferring the generated results. Here's the code you'll need to do this:

```
ENTAILS_LABEL = "_0"
NEUTRAL_LABEL = "_1"
CONTRADICTS_LABEL = "_2"
label_inds = tokenizer.convert_tokens_to_ids([
                            ENTAILS_LABEL,
                            NEUTRAL_LABEL,
                            CONTRADICTS_LABEL])
```

4. As you'll recall, the T5 model uses prefixes to know the task that it is supposed to perform. The following function provides the XNLI prefix, along with the premise and hypothesis in the proper format:

```
def process_nli(premise, hypothesis):
    return f'xnli: premise: {premise} hypothesis: {hypothesis}'
```

5. In the next step, for each label, a sentence will be generated, as illustrated in the following code snippet:

```
pairs =[(sequence_to_classify,\
      hypothesis_template.format(label)) for label in
      candidate_labels]
seqs = [process_nli(premise=premise,
                    hypothesis=hypothesis)
                    for premise, hypothesis in pairs]
```

6. You can see the resulting sequences by printing them, as follows:

```
print(seqs)
['xnli: premise: Wen werden Sie bei der nächsten Wahl wählen?  hypothesis: Dieses
Beispiel ist spor.',
'xnli: premise: Wen werden Sie bei der nächsten Wahl wählen?  hypothesis: Dieses
Beispiel ist ekonomi.',
'xnli: premise: Wen werden Sie bei der nächsten Wahl wählen?  hypothesis: Dieses
Beispiel ist politika.']
```

These sequences simply say that the task is XNLI-coded by `xnli:`; the premise sentence is `"Who will you vote for in the next election?"` (in German) and the hypothesis is `"this example is politics"`, `"this example is a sport"`, or `"this example is economy"`.

7. In the next step, you can tokenize the sequences and give them to the model to generate the text according to it, as follows:

```
inputs = tokenizer.batch_encode_plus(seqs,
          return_tensors="pt", padding=True)
out = model.generate(**inputs, output_scores=True,
         return_dict_in_generate=True,num_beams=1)
```

8. The generated text actually gives scores for each token in the vocabulary, and what we are looking for is the entailment, contradiction, and neutral scores. You can get their score using their token IDs, as follows:

```
scores = out.scores[0]
scores = scores[:, label_inds]
```

9. You can see these scores by printing them, like this:

```
>>> print(scores)
tensor([[-0.9851,  2.2550, -0.0783],
        [-5.1690, -0.7202, -2.5855],
        [ 2.7442,  3.6727,  0.7169]])
```

10. The neutral score is not required for our purpose, and we only need contradiction compared to entailment. So, you can use the following code to get only these scores:

```
entailment_ind = 0
contradiction_ind = 2
entail_vs_contra_scores = scores[:, [entailment_ind, contradiction_ind]]
```

11. Now that you have these scores for each sequence of the samples, you can apply a `softmax` layer on it to get the probabilities, as follows:

```
entail_vs_contra_probas = softmax(entail_vs_contra_scores, dim=1)
```

12. To see these probabilities, you can use `print`, like this:

```
>>> print(entail_vs_contra_probas)
tensor([[0.2877, 0.7123],
        [0.0702, 0.9298],
        [0.8836, 0.1164]])
```

13. Now, you can compare the entailment probability of these three samples by selecting them and applying a `softmax` layer over them, as follows:

```
entail_scores = scores[:, entailment_ind]
entail_probas = softmax(entail_scores, dim=0)
```

14. And to see the values, use `print`, as follows:

```
>>> print(entail_probas)
tensor([2.3438e-02, 3.5716e-04, 9.7620e-01])
```

15. The result means the highest probability belongs to the third sequence. In order to see it in a better shape, use the following code:

```
>>> print(dict(zip(candidate_labels, entail_probas.tolist())))
{'ekonomi': 0.0003571564157027751,
'politika': 0.9762046933174133,
'spor': 0.02343809604644754}
```

The whole process can be summarized as follows: each label is given to the model with the premise, and the model generates scores for each token in the vocabulary. We use these scores to find out how much the entailment token scores over the contradiction.

Fundamental limitations of multilingual models

Although the multilingual and cross-lingual models are promising and will affect the direction of NLP work, they still have some limitations. Many recent works addressed these limitations. Currently, the mBERT model slightly underperforms in many tasks compared with its monolingual counterparts and may not be a potential substitute for a well-trained monolingual model, which is why monolingual models are still widely used.

Studies in the field indicate that multilingual models suffer from the so-called *curse of multilingualism* as they seek to appropriately represent all languages. Adding new languages to a multilingual model improves its performance, up to a certain point. However, it is also seen that adding it after this point degrades performance, which may be due to shared vocabulary. Compared to monolingual models, multilingual models are significantly more limited in terms of the parameter budget. They need to allocate their vocabulary to each one of more than 100 languages.

The existing performance differences between mono- and multilingual models can be attributed to the capability of the designated tokenizer. The study *How Good is Your Tokenizer? On the Monolingual Performance of Multilingual Language Models* (2021) by Rust et al. (https://arxiv.org/abs/2012.15613) showed that when a dedicated language-specific tokenizer rather than a general-purpose one (a shared multilingual tokenizer) is attached to a multilingual model, it improves the performance for that language.

Some other findings indicate that it is not currently possible to represent all the world's languages in a single model due to an imbalance in resource distribution of different languages. As a solution, low-resource languages can be oversampled, while high-resource languages can be undersampled. Another observation is that knowledge transfer between two languages can be more efficient if those languages

are close. If they are distant languages, this transfer may have little effect. This observation may explain why we got worse results for Afrikaans and Tamil languages in the previous cross-lingual sentence-pair experiment part.

However, there is a lot of work on this subject, and these limitations may be overcome at any time. As of writing this article, the team of XML-R recently proposed two new models—namely, XLM-R XL and XLM-R XXL—that outperform the original XLM-R model by 1.8% and 2.4% average accuracies respectively on XNLI.

Fine-tuning the performance of multilingual models

Now, let's check whether the fine-tuned performance of the multilingual models is actually worse than the monolingual models or not. As an example, let's recall the example of Turkish text classification with seven classes in *Chapter 5*, *Fine-Tuning Language Models for Text Classification*. In that experiment, we fine-tuned a Turkish-specific monolingual model and achieved a good result. We will repeat the same experiment, keeping everything as-is but replacing the Turkish monolingual model with the mBERT and XLM-R models, respectively. Here's how we'll do this:

1. Let's recall the codes in that example again. We had fine-tuned the `"dbmdz/bert-base-turkish-uncased"` model, as follows:

```
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained(
                  "dbmdz/bert-base-turkish-uncased")
from transformers import BertForSequenceClassification
model = \ BertForSequenceClassification.from_pretrained("dbmdz/bert-base-turkish-
uncased",num_labels=NUM_LABELS,
                     id2label=id2label,
                     label2id=label2id)
```

With the monolingual model, we got the following performance values:

[77/77 01:24]

	eval_loss	eval_Accuracy	eval_F1	eval_Precision	eval_Recall
train	0.091844	0.975510	0.97546	0.975942	0.975535
val	0.280120	0.924898	0.92381	0.924427	0.924510
test	0.280038	0.926531	0.92542	0.927410	0.925425

Figure 9.15 – Monolingual text classification performance (from Chapter 5, Fine-Tuning Language Models for Text Classification)

2. To fine-tune with mBERT, we need to only replace the preceding model instantiation lines. Now, we will use the `"bert-base-multilingual-uncased"` multilingual model. We instantiate it like this:

```
from transformers import \ BertForSequenceClassification, AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(
                    "bert-base-multilingual-uncased")
model = BertForSequenceClassification.from_pretrained(
                     "bert-base-multilingual-uncased",
                      num_labels=NUM_LABELS,
                      id2label=id2label,
                      label2id=label2id)
```

3. There is not much difference in coding. When we run the experiment keeping all other parameters and settings the same, we get the following performance values:

[77/77 01:26]

	eval_loss	eval_Accuracy	eval_F1	eval_Precision	eval_Recall
train	0.093405	0.978367	0.978373	0.978547	0.978291
val	0.325458	0.911837	0.911586	0.911678	0.911592
test	0.372160	0.904490	0.903152	0.902647	0.904335

Figure 9.16 – mBERT fine-tuned performance

Hmm! The multilingual model underperforms compared with its monolingual counterpart roughly by 2.2% on all metrics.

4. Let's fine-tune the `"xlm-roberta-base"` XLM-R model for the same problem. We'll execute the XLM-R model initialization code, as follows:

```
from transformers import AutoTokenizer, XLMRobertaForSequenceClassification
tokenizer = AutoTokenizer.from_pretrained(
                                "xlm-roberta-base")
model = XLMRobertaForSequenceClassification\
                .from_pretrained("xlm-roberta-base",
                num_labels=NUM_LABELS,
               id2label=id2label,label2id=label2id)
```

5. Again, we keep all other settings exactly the same. We get the following performance values with the XML-R model:

[77/77 01:26]

	eval_loss	eval_Accuracy	eval_F1	eval_Precision	eval_Recall
train	0.122369	0.968571	0.968665	0.968830	0.968862
val	0.339011	0.912653	0.912454	0.913331	0.912042
test	0.334882	0.915918	0.915662	0.918334	0.914893

Figure 9.17 – XLM-R fine-tuned performance

Not bad! The XLM model did give comparable results. The obtained results are quite close to the monolingual model, with a roughly 1.0% difference. Therefore, although monolingual results can be better than multilingual models in certain tasks, we can achieve promising results with multilingual models. Think of it this way: we may not want to train a whole monolingual model for a 1% performance that lasts 10 days and more. Such small performance differences may be negligible for us.

Summary

In this chapter, you learned about multilingual and cross-lingual language model pre-training and the difference between monolingual and multilingual pre-training. CLM and TLM were also covered, and you gained knowledge about them. You learned how it is possible to use cross-lingual models on various use cases, such as semantic search, plagiarism, and zero-shot text classification. You also learned how it is possible to train on a dataset from a language and test on a completely different language using cross-lingual models. Fine-tuning the performance of multilingual models was evaluated, and we concluded that some multilingual models can be a substitute for monolingual models, remarkably keeping performance loss to a minimum.

In the next chapter, you will learn how to deploy transformer models for real problems and train them for production at an industrial scale.

References

Conneau, A., Lample, G., Rinott, R., Williams, A., Bowman, S. R., Schwenk, H. and Stoyanov, V. (2018). XNLI: Evaluating cross-lingual sentence representations. arXiv preprint arXiv:1809.05053.

Xue, L., Constant, N., Roberts, A., Kale, M., Al-Rfou, R., Siddhant, A. and Raffel, C. (2020). mT5: A massively multilingual pre-trained text-to-text transformer. arXiv preprint arXiv:2010.11934.

Lample, G. and Conneau, A. (2019). Cross-lingual language model pretraining. arXiv preprint arXiv:1901.07291.

Conneau, A., Khandelwal, K., Goyal, N., Chaudhary, V., Wenzek, G., Guzmán, F. and Stoyanov, V. (2019). Unsupervised cross-lingual representation learning at scale. arXiv preprint arXiv:1911.02116.

Feng, F., Yang, Y., Cer, D., Arivazhagan, N. and Wang, W. (2020). Language-agnostic bert sentence embedding. arXiv preprint arXiv:2007.01852.

Rust, P., Pfeiffer, J., Vulić, I., Ruder, S. and Gurevych, I. (2020). How Good is Your Tokenizer? On the Monolingual Performance of Multilingual Language Models. arXiv preprint arXiv:2012.15613.

Goyal, N., Du, J., Ott, M., Anantharaman, G. and Conneau, A. (2021). Larger-Scale Transformers for Multilingual Masked Language Modeling. arXiv preprint arXiv:2105.00572.

Previous Chapter

Next Chapter