Chapter 7: Text Representation

So far, we have addressed classification and generation problems with the `transformers` library. Text representation is another crucial task in modern **Natural Language Processing** (**NLP**), especially for unsupervised tasks such as clustering, semantic search, and topic modeling. Representing sentences by using various models such as **Universal Sentence Encoder** (**USE**) and Siamese BERT (Sentence-BERT) with additional libraries such as sentence transformers will be explained here. Zero-shot learning using BART will also be explained, and you will learn how to utilize it. Few-shot learning methodologies and unsupervised use cases such as semantic text clustering and topic modeling will also be described. Finally, one-shot learning use cases such as semantic search will be covered.

The following topics will be covered in this chapter:

- Introduction to sentence embeddings
- Benchmarking sentence similarity models
- Using BART for zero-shot learning
- Semantic similarity experiment with FLAIR
- Text clustering with Sentence-BERT
- Semantic search with Sentence-BERT

Technical requirements

We will be using a Jupyter notebook to run our coding exercises. For this, you will need Python 3.6+ and the following packages:

- `sklearn`
- `transformers >=4.00`

```
datasets
sentence-transformers
tensorflow-hub
flair
umap-learn
bertopic
```

All the notebooks for the coding exercises in this chapter will be available at the following GitHub link:

https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH07

Check out the following link to see the Code in Action video: https://bit.ly/2VcMCyI

Introduction to sentence embeddings

Pre-trained BERT models do not produce efficient and independent sentence embeddings as they always need to be fine-tuned in an end-to-end supervised setting. This is because we can think of a pre-trained BERT model as an indivisible whole and semantics is spread across all layers, not just the final layer. Without fine-tuning, it may be ineffective to use its internal representations independently. It is also hard to handle unsupervised tasks such as clustering, topic modeling, information retrieval, or semantic search. Because we have to evaluate many sentence pairs during clustering tasks, for instance, this causes massive computational overhead.

Luckily, many modifications have been made to the original BERT model, such as **Sentence-BERT (SBERT)**, to derive semantically meaningful and independent sentence embeddings. We will talk about these approaches in a moment. In the NLP literature, many neural sentence embedding methods have been proposed for mapping a single sentence to a common feature space (vector space model) wherein a cosine function (or dot product) is usually used to measure similarity and the Euclidean distance to measure dissimilarity.

The following are some applications that can be efficiently solved with sentence embeddings:

- Sentence-pair tasks
- Information retrieval
- Question answering
- Duplicate question detection
- Paraphrase detection
- Document clustering
- Topic modeling

The simplest but most efficient kind of neural sentence embedding is the average-pooling operation, which is performed on the embeddings of words in a sentence. To get a better representation of this, some early neural methods learned sentence embeddings in an unsupervised fashion, such as Doc2Vec, Skip-Thought, FastSent, and Sent2Vec. Doc2Vec utilized a token-level distributional theory and an objective function to predict adjacent words, similar to Word2Vec. The approach injects an additional memory token (called **Paragraph-ID**) into each sentence, which is reminiscent of CLS or SEP tokens in the `transformers` library. This additional token acts as a piece of memory that represents the context or document embeddings. SkipThought and FastSent are considered sentence-level approaches, where the objective function is used to predict adjacent sentences. These models extract the sentence's meaning to obtain necessary information from the adjacent sentences and their context.

Some other methods, such as InferSent, leveraged supervised learning and multi-task transfer learning to learn generic sentence embeddings. InferSent trained various supervised tasks to get more efficient embedding. RNN-based supervised models such as GRU or LSTM utilize the last hidden state (or stacked entire hidden states) to obtain sentence embeddings in a supervised setting. We touched on the RNN approach in *Chapter 1*, *From Bag-of-Words to the Transformers.*

Cross-encoder versus bi-encoder

So far, we have discussed how to train Transformer-based language models and fine-tune them in semi-supervised and supervised settings, respectively. As we learned in the previous chapters, we got successful results thanks to the transformer architectures. Once a task-specific thin linear layer has been put on top of a pre-trained model, all the weights of the network (not only the last task-specific thin layer) are fine-tuned with task-specific labeled data. We also experienced how the BERT architecture has been fine-tuned for two different groups of tasks (single-sentence or sentence-pair) without any architectural modifications being required. The only difference is that for the sentence-pair tasks, the sentences are concatenated and marked with a SEP token. Thus, self-attention is applied to all the tokens of the concatenated sentences. This is a big advantage of the BERT model, where both input sentences can get the necessary information from each other at every layer. In the end, they are encoded simultaneously. This is called cross-encoding.

However, there are two disadvantages regarding the cross-encoders that were addressed by the SBERT authors and *Humeau et al., 2019*, as follows:

> The cross-encoder setup is not convenient for many sentence-pair tasks due to too many possible combinations needing to be processed. For instance, to get the two closest sentences from a list of 1,000 sentences, the cross-encoder model (BERT) requires around 500,000 (*n * (n-1) /2*) inference computation. Therefore, it would be very slow compared to alternative solutions such as SBERT or USE. This is because these alternatives produce independent sentence embeddings wherein the similarity function (cosine similarity) or dissimilarity function (Euclidean or Manhattan) can easily be applied. Note that these dis/similarity functions can be performed efficiently on modern architectures. Moreover, with the help of an optimized index structure, we can reduce computational complexity from many hours to a few minutes when comparing or clustering many documents.
>
> Due to its supervised characteristics, the BERT model can't derive independent meaningful sentence embeddings. It is hard to leverage a pre-trained BERT model as is for unsupervised tasks such as clustering, semantic search, or topic modeling. The BERT model produces a fixed-size vector for each token in a document. In an unsupervised setting, the document-level representation may be obtained by averaging or pooling token vectors, plus SEP and CLS tokens. Later, we will see that such a representation of BERT produces below-average sentence embeddings, and that its performance scores are usually worse than word embedding pooling techniques such as Word2Vec, FastText, or GloVe.

Alternatively, bi-encoders (such as SBERT) independently map a sentence pair to a semantic vector space, as shown in the following diagram. Since the representations are separate, bi-encoders can cache the encoded input representation for each input, resulting in fast inference time. One of the successful bi-encoder modifications of BERT is SBERT. Based on the Siamese and Triplet network structures, SBERT fine-tunes the BERT model to produce semantically meaningful and independent embeddings of the sentences.

The following diagram shows the bi-encoder architecture:

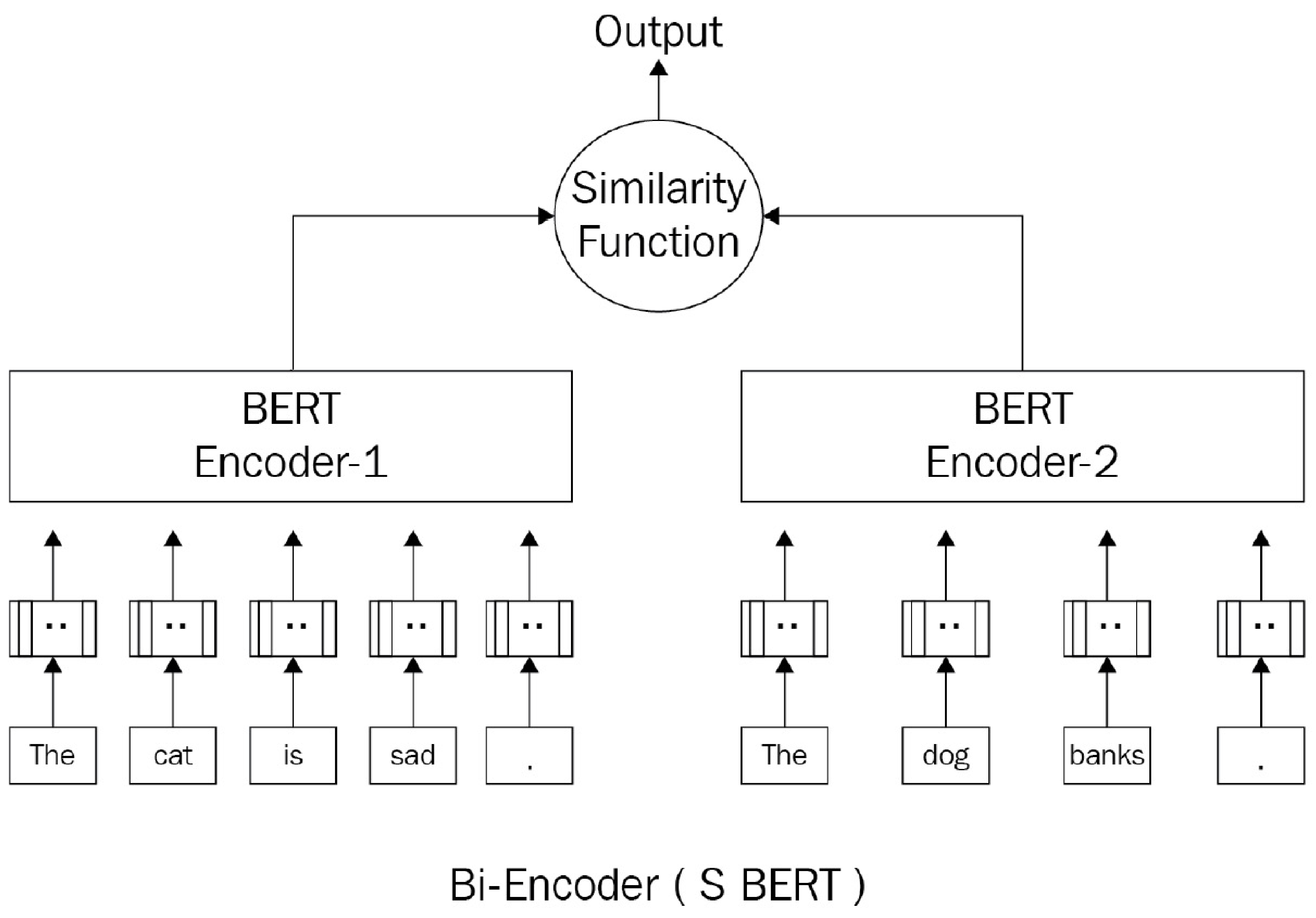


Figure 7.1 – Bi-encoder architecture

You can find hundreds of pre-trained SBERT models that have been trained with different objectives at https://public.ukp.informatik.tu-darmstadt.de/reimers/sentence-transformers/v0.2/.

We will use some of them in the next section.

Benchmarking sentence similarity models

There are many *semantic textual similarity* models available, but it is highly recommended that you benchmark and understand their capabilities and differences using metrics. *Papers With Code* provides a list of these datasets at https://paperswithcode.com/task/semantic-textual-similarity.

Also, there are many model outputs in each dataset that are ranked by their results. These results have been taken from the aforementioned article.

GLUE provides most of these datasets and tests, but it is not only for semantic textual similarity. **GLUE**, which stands for **General Language Understanding Evaluation**, is a general benchmark for evaluating a model with different NLP characteristics. More details about the GLUE dataset and its usage was provided in Chapter 2, *A Hands-On Introduction to the Subject*. Let's take a look at it before we move on:

1. To load the metrics and MRPC dataset from the GLUE benchmark, you can use the following code:

   Copy Explain

   ```
   from datasets import load_metric, load_dataset
   metric = load_metric('glue', 'mrpc')
   mrpc = load_dataset('glue', 'mrpc')
   ```

   The samples in this dataset are labeled `1` and `0`, which indicates whether they are similar or dissimilar, respectively. You can use any model, regardless of the architecture, to produce values for two given sentences. In other words, the model should classify the two sentences as zeros and ones.

2. Let's assume the model produces values and that these values are stored in an array called `predictions`. You can easily use this metric with the predictions to see the F1 and accuracy values:

   Copy Explain

   ```
   labels = [i['label'] for i in dataset['test']]
   metric.compute(predictions=predictions, references=labels)
   ```

3. Some semantic textual similarity datasets such as **Semantic Textual Similarity Benchmark (STSB)** have different metrics. For example, this benchmark uses Spearman and Pearson correlations because the outputs and predictions are between 0 and 5 and are float numbers instead of being 0s and 1s, which is a regression problem. The following code shows an example of this benchmark:

   Copy Explain

   ```
   metric = load_metric('glue', 'stsb')
   metric.compute(predictions=[1,2,3],references=[5,2,2])
   ```

   The predictions and references are the same as the ones from the **Microsoft Research Paraphrase Corpus (MRPC)**; the predictions are the model outputs, while the references are the dataset labels.

4. To get a comparative result between two models, we will use a distilled version of Roberta and test these two on the STSB. To start, you must load both models. The following code shows how to install the required libraries before loading and using models:

```
pip install tensorflow-hub
pip install sentence-transformers
```

5. As we mentioned previously, the next step is to load the dataset and metric:

```
from datasets import load_metric, load_dataset
stsb_metric = load_metric('glue', 'stsb')
stsb = load_dataset('glue', 'stsb')
```

6. Afterward, we must load both models:

```
import tensorflow_hub as hub
use_model = hub.load(
   "https://tfhub.dev/google/universal-sentence-encoder/4")
from sentence_transformers import SentenceTransformer
distilroberta = SentenceTransformer(
                      'stsb-distilroberta-base-v2')
```

7. Both of these models provide embeddings for a given sentence. To compare the similarity between two sentences, we will use cosine similarity. The following function takes sentences as a batch and provides cosine similarity for each pair by utilizing USE:

```
import tensorflow as tf
import math
def use_sts_benchmark(batch):
  sts_encode1 = \
  tf.nn.l2_normalize(use_model(tf.constant(batch['sentence1'])), axis=1)
  sts_encode2 = \
  tf.nn.l2_normalize(use_model(tf.constant(batch['sentence2'])),   axis=1)
  cosine_similarities = \
              tf.reduce_sum(tf.multiply(sts_encode1,sts_encode2),axis=1)
  clip_cosine_similarities = \
          tf.clip_by_value(cosine_similarities,-1.0, 1.0)
  scores = 1.0 - \
           tf.acos(clip_cosine_similarities) / math.pi
return scores
```

8. With small modifications, the same function can be used for RoBERTa too. These small modifications are only for replacing the embedding function, which is different for TensorFlow Hub models and transformers. The following is the modified function:

```
def roberta_sts_benchmark(batch):
  sts_encode1 = \
  tf.nn.l2_normalize(distilroberta.encode(batch['sentence1']), axis=1)
  sts_encode2 = \
    tf.nn.l2_normalize(distilroberta.encode(batch['sentence2']), axis=1)
  cosine_similarities = \
          tf.reduce_sum(tf.multiply(sts_encode1, sts_encode2),  axis=1)
  clip_cosine_similarities = tf.clip_by_value(cosine_similarities, -1.0, 1.0)
  scores = 1.0  - tf.acos(clip_cosine_similarities) / math.pi
return scores
```

9. Applying these functions to the dataset will result in similarity scores for each of the models:

```
use_results = use_sts_benchmark(stsb['validation'])
distilroberta_results = roberta_sts_benchmark(
                                      stsb['validation'])
```

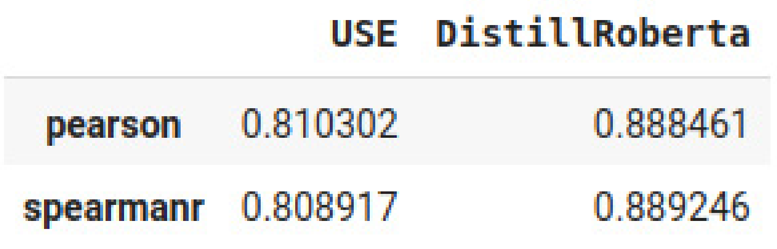
10. Using metrics on both results produces the Spearman and Pearson correlation values:

```
results = {
      "USE":stsb_metric.compute(
                predictions=use_results,
                references=references),
      "DistillRoberta":stsb_metric.compute(
                predictions=distilroberta_results,
                references=references)
}
```

11. You can simply use pandas to see the results as a table in a comparative fashion:

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

The output is as follows:



	USE	DistillRoberta
pearson	0.810302	0.888461
spearmanr	0.808917	0.889246

Figure 7.2 – STSB validation results on DistilRoberta and USE

In this section, you learned about the important benchmarks of semantic textual similarity. Regardless of the model, you learned how to use any of these metrics to quantify model performance. In the next section, you will learn about the few-shot learning models.

Using BART for zero-shot learning

In the field of machine learning, zero-shot learning is referred to as models that can perform a task without explicitly being trained on it. In the case of NLP, it's assumed that there's a model that can predict the probability of some text being assigned to classes that are given to the model. However, the interesting part about this type of learning is that the model is not trained on these classes.

With the rise of many advanced language models that can perform transfer learning, zero-shot learning came to life. In the case of NLP, this kind of learning is performed by NLP models at test time, where the model sees samples belonging to new classes where no samples of them were seen before.

This kind of learning is usually used for classification tasks, where both the classes and the text are represented and the semantic similarity of both is compared. The represented form of these two is an embedding vector, while the similarity metric (such as cosine similarity or a pre-trained classifier such as a dense layer) outputs the probability of the sentence/text being classified as the class.

There are many methods and schemes we can use to train such models, but one of the earliest methods used crawled pages from the internet containing keyword tags in the meta part. For more information, read the following article and blog post at https://amitness.com/2020/05/zero-shot-text-classification/.

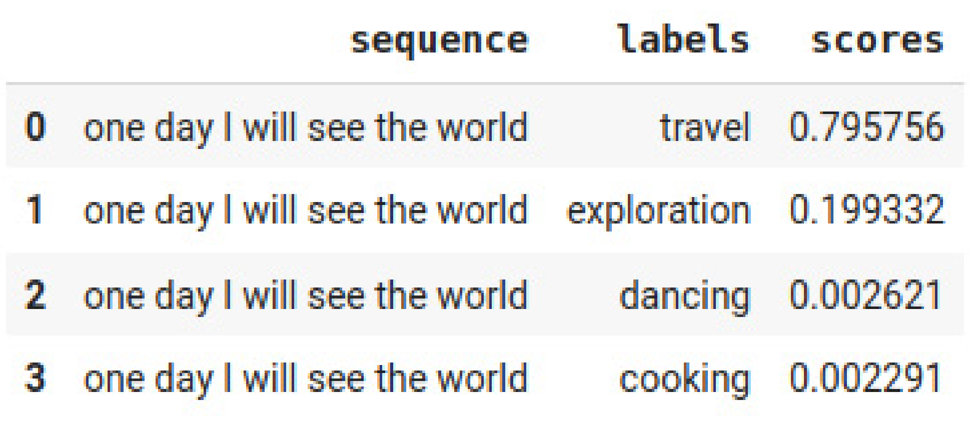
Instead of using such huge data, there are language models such as BART that use the **Multi-Genre Natural Language Inference (MNLI)** dataset to fine-tune and detect the relationship between two different sentences. Also, the HuggingFace model repository contains many models that have been implemented for zero-shot learning. They also provide a zero-shot learning pipeline for ease of use.

For example, BART from **Facebook AI Research (FAIR)** is being used in the following code to perform zero-shot text classification:

Copy Explain

```
from transformers import pipeline
import pandas as pd
classifier = pipeline("zero-shot-classification",
                      model="facebook/bart-large-mnli")
sequence_to_classify = "one day I will see the world"
candidate_labels = ['travel',
                    'cooking',
                    'dancing',
                    'exploration']
result = classifier(sequence_to_classify, candidate_labels)
pd.DataFrame(result)
```

The results are as follows:



	sequence	labels	scores
0	one day I will see the world	travel	0.795756
1	one day I will see the world	exploration	0.199332
2	one day I will see the world	dancing	0.002621
3	one day I will see the world	cooking	0.002291

Figure 7.3 – Results of zero-shot learning using BART

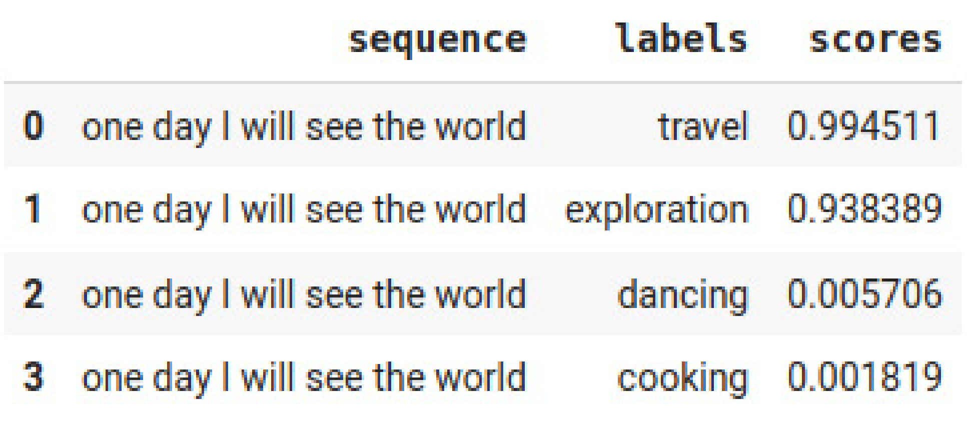
As you can see, the travel and exploration labels have the highest probability, but the most probable one is travel.

However, sometimes, one sample can belong to more than one class (multilabel). HuggingFace provides a parameter called `multi_label` for this. The following example uses this parameter:

Copy Explain

```
result = classifier(sequence_to_classify,
                      candidate_labels,
                      multi_label=True)
Pd.DataFrame(result)
```

Due to this, it is changed to the following:



	sequence	labels	scores
0	one day I will see the world	travel	0.994511
1	one day I will see the world	exploration	0.938389
2	one day I will see the world	dancing	0.005706
3	one day I will see the world	cooking	0.001819

Figure 7.4 – Results of zero-shot learning using BART (multi_label = True)

You can test the results even further and see how the model performs if very similar labels to the travel one are used. For example, you can see how it performs if `moving` and `going` are added to the label list.

There are other models that also leverage the semantic similarity between labels and the context to perform zero-shot classification. In the case of few-shot learning, some samples are given to the model, but these samples are not enough to train a model alone. Models can use these samples to perform tasks such as semantic text clustering, which will be explained shortly.

Now that you've learned how to use BART for zero-shot learning, you should learn how it works. BART is fine-tuned on **Natural Language Inference** (**NLI**) datasets such as MNLI. These datasets contain sentence pairs and three classes for each pair; that is, *Neutral*, *Entailment*, and *Contradiction*. Models that have been trained on these datasets can capture the semantics of two sentences and classify them by assigning a label in one-hot format. If you take out Neutral labels and only use Entailment and Contradiction as your output labels, if two sentences can come after each other, then it means these two are closely related to each other. In other words, you can change the first sentence to the label (`travel`, for example) and the second sentence to the content (`one day I will see the world`, for example). According to this, if these two can come after each other, this means that the label and the content are semantically related. The following code example shows how to directly use the BART model without the zero-shot classification pipeline according to the preceding descriptions:

Copy Explain

```
from transformers \
      import AutoModelForSequenceClassification,\
      AutoTokenizer
nli_model = AutoModelForSequenceClassification\
                .from_pretrained(
                    "facebook/bart-large-mnli")
tokenizer = AutoTokenizer\
                .from_pretrained(
           "facebook/bart-large-mnli")
premise = "one day I will see the world"
label = "travel"
hypothesis = f'This example is {label}.'
x = tokenizer.encode(
    premise,
    hypothesis,
    return_tensors='pt',
    truncation_strategy='only_first')
logits = nli_model(x)[0]
entail_contradiction_logits = logits[:,[0,2]]
probs = entail_contradiction_logits.softmax(dim=1)
prob_label_is_true = probs[:,1]
print(prob_label_is_true)
```

The result is as follows:

Copy Explain

```
tensor([0.9945], grad_fn=<SelectBackward>)
```

You can also call the first sentence the hypothesis and the sentence containing the label the premise. According to the result, the premise can entail the hypothesis. which means that the hypothesis is labeled as the premise.

So far, you've learned how to use zero-shot learning by utilizing NLI fine-tuned models. Next, you will learn how to perform few-/one-shot learning using semantic text clustering and semantic search.

Semantic similarity experiment with FLAIR

In this experiment, we will qualitatively evaluate the sentence representation models thanks to the `flair` library, which really simplifies obtaining the document embeddings for us.

We will perform experiments while taking on the following approaches:

- Document average pool embeddings
- RNN-based embeddings
- BERT embeddings
- SBERT embeddings

We need to install these libraries before we can start the experiments:

Copy Explain

```
!pip install sentence-transformers
!pip install dataset
!pip install flair
```

For qualitative evaluation, we define a list of similar sentence pairs and a list of dissimilar sentence pairs (five pairs for each). What we expect from the embeddings models is that they should measure a high score and a low score, respectively.

The sentence pairs are extracted from the SBS Benchmark dataset, which we are already familiar with from the sentence-pair regression part of Chapter 6, *Fine-Tuning Language Models for Token Classification*. For similar pairs, two sentences are completely equivalent, and they share the same meaning.

The pairs with a similarity score of around 5 in the STSB dataset are randomly taken, as follows:

Copy Explain

```
import pandas as pd
similar=[
 ("A black dog walking beside a pool.",
 "A black dog is walking along the side of a pool."),
 ("A blonde woman looks for medical supplies for work in a  suitcase. ",
 " The blond woman is searching for medical supplies in a  suitcase."),
 ("A doubly decker red bus driving down the road.",
 "A red double decker bus driving down a street."),
 ("There is a black dog jumping into a swimming pool.",
 "A black dog is leaping into a swimming pool."),
 ("The man used a sword to slice a plastic bottle.",
 "A man sliced a plastic bottle with a sword.")]
pd.DataFrame(similar, columns=["sen1", "sen2"])
```

The output is as follows:

	sen1	sen2
0	A black dog walking beside a pool.	A black dog is walking along the side of a pool.
1	A blonde woman looks for medical supplies for ...	The blond woman is searching for medical supp...
2	A doubly decker red bus driving down the road.	A red double decker bus driving down a street.
3	There is a black dog jumping into a swimming p...	A black dog is leaping into a swimming pool.
4	The man used a sword to slice a plastic bottle.\t	A man sliced a plastic bottle with a sword.

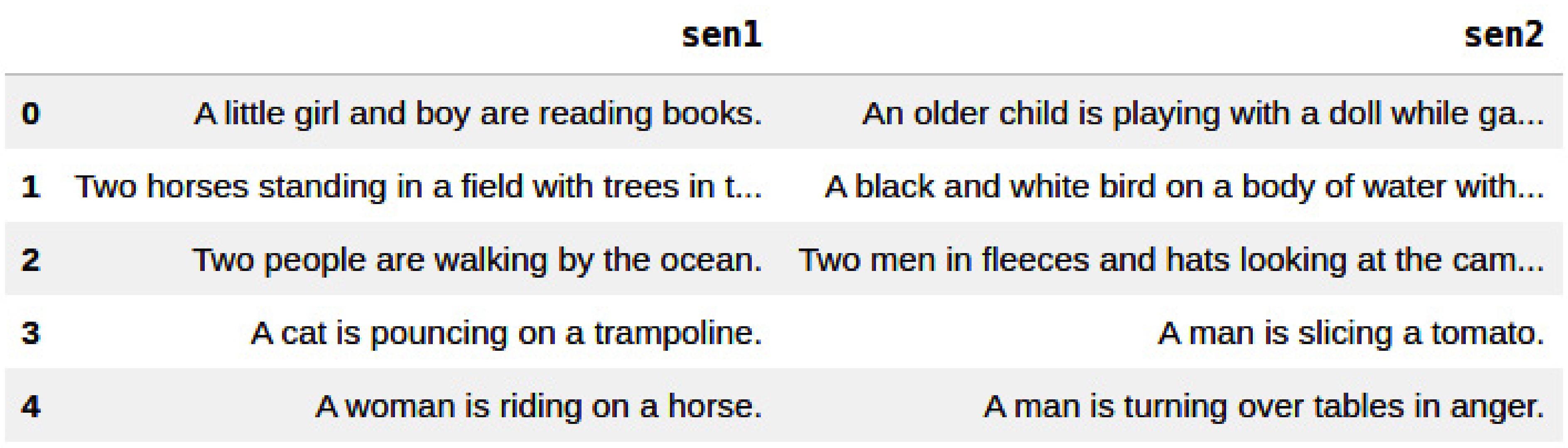
Figure 7.5 – Similar pair list

Here is the list of dissimilar sentences whose similarity scores are around 0, taken from the STS-B dataset:

Copy Explain

```
import pandas as pd
dissimilar= [
("A little girl and boy are reading books. ",
 "An older child is playing with a doll while gazing out the window."),
 ("Two horses standing in a field with trees in the background.",
 "A black and white bird on a body of water with grass in the background."),
 ("Two people are walking by the ocean.",
 "Two men in fleeces and hats looking at the camera."),
 ("A cat is pouncing on a trampoline.",
 "A man is slicing a tomato."),
("A woman is riding on a horse.",
 "A man is turning over tables in anger.")]
pd.DataFrame(dissimilar, columns=["sen1", "sen2"])
```

The output is as follows:



	sen1	sen2
0	A little girl and boy are reading books.	An older child is playing with a doll while ga...
1	Two horses standing in a field with trees in t...	A black and white bird on a body of water with...
2	Two people are walking by the ocean.	Two men in fleeces and hats looking at the cam...
3	A cat is pouncing on a trampoline.	A man is slicing a tomato.
4	A woman is riding on a horse.	A man is turning over tables in anger.

Figure 7.6 – Dissimilar pair list

Now, let's prepare the necessary functions to evaluate the embeddings models. The following `sim()` function computes the cosine similarity between two sentences; that is, `s1`, `s2`:

Copy Explain

```
import torch, numpy as np
def sim(s1,s2):
    s1=s1.embedding.unsqueeze(0)
    s2=s2.embedding.unsqueeze(0)
    sim=torch.cosine_similarity(s1,s2).item()
    return np.round(sim,2)
```

The document embeddings models that were used in this experiment are all pre-trained models. We will pass the document embeddings model object and sentence pair list (similar or dissimilar) to the following `evaluate()` function, where, once the model encodes the sentence embeddings, it will compute the similarity score for each pair in the list, along with the list average. The definition of the function is as follows:

Copy Explain

```
from flair.data import Sentence
def evaluate(embeddings, myPairList):
    scores=[]
    for s1, s2 in myPairList:
        s1,s2=Sentence(s1), Sentence(s2)
        embeddings.embed(s1)
        embeddings.embed(s2)
        score=sim(s1,s2)
        scores.append(score)
    return scores, np.round(np.mean(scores),2)
```

Now, it is time to evaluate sentence embedding models. We will start with the average pooling method!

Average word embeddings

Average word embeddings (or **document pooling**) apply the mean pooling operation to all the words in a sentence, where the average of all the word embeddings is considered to be sentence embedding. The following execution instantiates a document pool embedding based on GloVe vectors. Note that although we will use only GloVe vectors here, the flair API allows us to use multiple word embeddings. Here is the code definition:

```
from flair.data import Sentence
from flair.embeddings\
     import WordEmbeddings, DocumentPoolEmbeddings
glove_embedding = WordEmbeddings('glove')
glove_pool_embeddings = DocumentPoolEmbeddings(
                                      [glove_embedding]
                                      )
```

Let's evaluate the GloVe pool model on similar pairs, as follows:

```
>>> evaluate(glove_pool_embeddings, similar)
([0.97, 0.99, 0.97, 0.99, 0.98], 0.98)
```

The results seem to be good since those resulting values are very high, which is what we expect. However, the model produces high scores such as 0.94 on average for the dissimilar list as well. Our expectation would be less than 0.4. We'll talk about why we got this later in this chapter. Here is the execution:

```
>>> evaluate(glove_pool_embeddings, dissimilar)
([0.94, 0.97, 0.94, 0.92, 0.93], 0.94)
```

Next, let's evaluate some RNN embeddings on the same problem.

RNN-based document embeddings

Let's instantiate a GRU model based on GloVe embeddings, where the default model of `DocumentRNNEmbeddings` is a GRU:

```
from flair.embeddings \
      import WordEmbeddings, DocumentRNNEmbeddings
gru_embeddings = DocumentRNNEmbeddings([glove_embedding])
```

Run the evaluation method:

```
>>> evaluate(gru_embeddings, similar)
([0.99, 1.0, 0.94, 1.0, 0.92], 0.97)
>>> evaluate(gru_embeddings, dissimilar)
([0.86, 1.0, 0.91, 0.85, 0.9], 0.9)
```

Likewise, we get a high score for the dissimilar list. This is not what we want from sentence embeddings.

Transformer-based BERT embeddings

The following execution instantiates a `bert-base-uncased` model that pools the final layer:

```
from flair.embeddings import TransformerDocumentEmbeddings
from flair.data import Sentence
bert_embeddings = TransformerDocumentEmbeddings(
                                      'bert-base-uncased')
```

Run the evaluation, as follows:

```
>>> evaluate(bert_embeddings, similar)
([0.85, 0.9, 0.96, 0.91, 0.89], 0.9)
>>> evaluate(bert_embeddings, dissimilar)
([0.93, 0.94, 0.86, 0.93, 0.92], 0.92)
```

This is worse! The score of the dissimilar list is higher than that of the similar list.

Sentence-BERT embeddings

Now, let's apply Sentence-BERT to the problem of distinguishing similar pairs from dissimilar ones, as follows:

1. First of all, a warning: we need to ensure that the `sentence-transformers` package has already been installed:

   Copy Explain

   ```
   !pip install sentence-transformers
   ```

2. As we mentioned previously, Sentence-BERT provides a variety of pre-trained models. We will pick the `bert-base-nli-mean-tokens` model for evaluation. Here is the code:

   Copy Explain

   ```
   from flair.data import Sentence
   from flair.embeddings \
         import SentenceTransformerDocumentEmbeddings
   sbert_embeddings = SentenceTransformerDocumentEmbeddings(
                      'bert-base-nli-mean-tokens')
   ```

3. Let's evaluate the model:

   Copy Explain

   ```
   >>> evaluate(sbert_embeddings, similar)
   ([0.98, 0.95, 0.96, 0.99, 0.98], 0.97)
   >>> evaluate(sbert_embeddings, dissimilar)
   ([0.48, 0.41, 0.19, -0.05, 0.0], 0.21)
   ```

   Well done! The SBERT model produced better results. The model produced a low similarity score for the dissimilar list, which is what we expect.

4. Now, we will do a harder test, where we pass contradicting sentences to the models. We will define some tricky sentence pairs, as follows:

   Copy Explain

   ```
   >>> tricky_pairs=[
   ("An elephant is bigger than a lion",
   "A lion is bigger than an elephant") ,
   ("the cat sat on the mat",
   "the mat sat on the cat")]
   >>> evaluate(glove_pool_embeddings, tricky_pairs)
   ([1.0, 1.0], 1.0)
   >>> evaluate(gru_embeddings, tricky_pairs)
   ([0.87, 0.65], 0.76)
   >>> evaluate(bert_embeddings, tricky_pairs)
   ([1.0, 0.98], 0.99)
   >>> evaluate(sbert_embeddings, tricky_pairs)
   ([0.93, 0.97], 0.95)
   ```

Interesting! The scores are very high since the sentence similarity model works similar to topic detection and measures content similarity. When we look at the sentences, they share the same content, even though they contradict each other. The content is about lion and elephant or cat and mat. Therefore, the models produce a high similarity score. Since the GloVe embedding method pools the average of the words without caring about word order, it measures two sentences as being the same. On the other hand, the GRU model produced lower values as it cares about word order. Surprisingly, even the SBERT model does not produce efficient scores. This may be due to the content similarity-based supervision that's used in the SBERT model.

5. To correctly detect the semantics of two sentence pairs with three classes – that is, Neutral, Contradiction, and Entailment – we must use a fine-tuned model on MNLI. The following code block shows an example of using XLM-Roberta, fine-tuned on XNLI with the same examples:

Copy Explain

```
from transformers \
Import AutoModelForSequenceClassification, AutoTokenizer
nli_model = AutoModelForSequenceClassification\
                .from_pretrained(
                      'joeddav/xlm-roberta-large-xnli')
tokenizer = AutoTokenizer\
                .from_pretrained(
                      'joeddav/xlm-roberta-large-xnli')
import numpy as np
for permise, hypothesis in tricky_pairs:
  x = tokenizer.encode(premise,
                      hypothesis,
                      return_tensors='pt',
                      truncation_strategy='only_first')
  logits = nli_model(x)[0]
  print(f"Permise: {permise}")
  print(f"Hypothesis: {hypothesis}")
  print("Top Class:")
 print(nli_model.config.id2label[np.argmax(
                       logits[0].detach().numpy()). ])
  print("Full softmax scores:")
  for i in range(3):
    print(nli_model.config.id2label[i],
           logits.softmax(dim=1)[0][i].detach().numpy())
  print("="*20)
```

6. The output will show the correct labels for each:

```
Permise: An elephant is bigger than a lion
Hypothesis: A lion is bigger than an elephant
Top Class:
contradiction
Full softmax scores:
contradiction 0.7731286
neutral 0.2203285
entailment 0.0065428796
====================
Permise: the cat sat on the mat
Hypothesis: the mat sat on the cat
Top Class:
entailment
Full softmax scores:
contradiction 0.49365467
neutral 0.007260764
entailment 0.49908453
====================
```

In some problems, In some problems, NLI is a higher priority than semantic textual because it is intended to find the contradiction or entailment rather than the raw similarity score. For the next sample, use two sentences for entailment and contradiction at the same time. This is a bit subjective, but to the model, the second sentence pair seems to be a very close call between entailment and contradiction.

Text clustering with Sentence-BERT

For clustering algorithms, we will need a model that's suitable for textual similarity. Let's use the `paraphrase-distilroberta-base-v1` model here for a change. We will start by loading the Amazon Polarity dataset for our clustering experiment. This dataset includes Amazon web page reviews spanning a period of 18 years up to March 2013. The original dataset includes over 35 million reviews. These reviews include product information, user information, user ratings, and user reviews. Let's get started:

1. First, randomly select 10K reviews by shuffling, as follows:

Copy Explain

```
import pandas as pd, numpy as np
import torch, os, scipy
from datasets import load_dataset
dataset = load_dataset("amazon_polarity",split="train")
corpus=dataset.shuffle(seed=42)[:10000]['content']
```

2. The corpus is now ready for clustering. The following code instantiates a sentence-transformer object using the pre-trained `paraphrase-distilroberta-base-v1` model:

Copy Explain

```
from sentence_transformers import SentenceTransformer
model_path="paraphrase-distilroberta-base-v1"
model = SentenceTransformer(model_path)
```

3. The entire corpus is encoded with the following execution, where the model maps a list of sentences to a list of embedding vectors:

Copy Explain

```
>>> corpus_embeddings = model.encode(corpus)
>>> corpus_embeddings.shape
(10000, 768)
```

4. Here, the vector size is `768`, which is the default embedding size of the BERT-base model. From now on, we will proceed with traditional clustering methods. We will choose *Kmeans* here since it is a fast and widely used clustering algorithm. We just need to set the cluster number (*K*) to `5`. Actually, this number may not be optimal. There are many techniques that can determine the optimal number of clusters, such as the Elbow or Silhouette method. However, let's leave these issues aside. Here is the execution:

Copy Explain

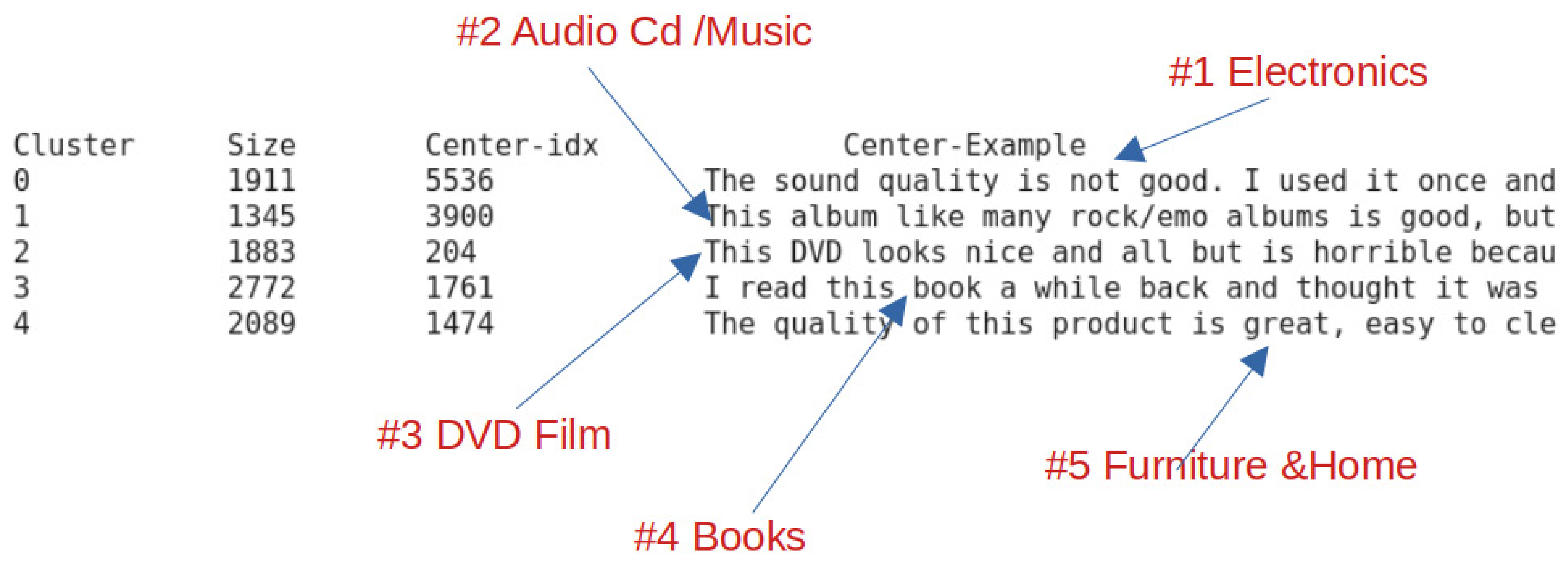
```
>>> from sklearn.cluster import KMeans
>>> K=5
>>> kmeans = KMeans(
           n_clusters=5,
           random_state=0).fit(corpus_embeddings)
>>> cls_dist=pd.Series(kmeans.labels_).value_counts()
>>> cls_dist
3 2772
4 2089
0 1911
2 1883
1 1345
```

Here, we have obtained five clusters of reviews. As we can see from the output, we have fairly distributed clusters. Another issue with clustering is that we need to understand what these clusters mean. As a suggestion, we can apply topic analysis to each cluster or check cluster-based TF-IDF to understand the content. Now, let's look at another way to do this based on the cluster centers. The Kmeans algorithm computes cluster centers, called centroids, that are kept in the `kmeans.cluster_centers_` attribute. The centroids are simply the average of the vectors in each cluster. Therefore, they are all imaginary points, not the existing data points. Let's assume that the sentences closest to the centroid will be the most representative example for the corresponding cluster.

5. Let's try to find only one real sentence embedding, closest to each centroid point. If you like, you can capture more than one sentence. Here is the code:

```
distances = \
scipy.spatial.distance.cdist(kmeans.cluster_centers_, corpus_embeddings)
centers={}
print("Cluster", "Size", "Center-idx",
                      "Center-Example", sep="\t\t")
for i,d in enumerate(distances):
    ind = np.argsort(d, axis=0)[0]
    centers[i]=ind
    print(i,cls_dist[i], ind, corpus[ind] ,sep="\t\t")
```

The output is as follows:



```
Cluster      Size         Center-idx           Center-Example
0            1911         5536          The sound quality is not good. I used it once and
1            1345         3900          This album like many rock/emo albums is good, but
2            1883         204           This DVD looks nice and all but is horrible becau
3            2772         1761          I read this book a while back and thought it was
4            2089         1474          The quality of this product is great, easy to cle
```

Figure 7.7 – Centroids of the cluster

From these representative sentences, we can reason about the clusters. It seems to be that Kmeans clusters the reviews into five distinct categories: *Electronics*, *Audio Cd/Music*, *DVD Film*, *Books*, and *Furniture & Home*. Now, let's visualize both sentence points and cluster centroids in a 2D space. We will use the **Uniform Manifold Approximation and Projection (UMAP)** library to reduce dimensionality. Other widely used dimensionality reduction techniques in NLP that you can use include t-SNE and PCA (see Chapter 1, *From Bag-of-Words to the Transformers*).

6. We need to install the `umap` library, as follows:

```
!pip install umap-learn
```

7. The following execution reduces all the embeddings and maps them into a 2D space:

```
import matplotlib.pyplot as plt
import umap
X = umap.UMAP(
           n_components=2,
           min_dist=0.0).fit_transform(corpus_embeddings)
labels= kmeans.labels_fig, ax = plt.subplots(figsize=(12,
8))
plt.scatter(X[:,0], X[:,1], c=labels, s=1, cmap='Paired')
for c in centers:
    plt.text(X[centers[c],0], X[centers[c], 1],"CLS-"+ str(c), fontsize=18)
    plt.colorbar()
```

The output is as follows:

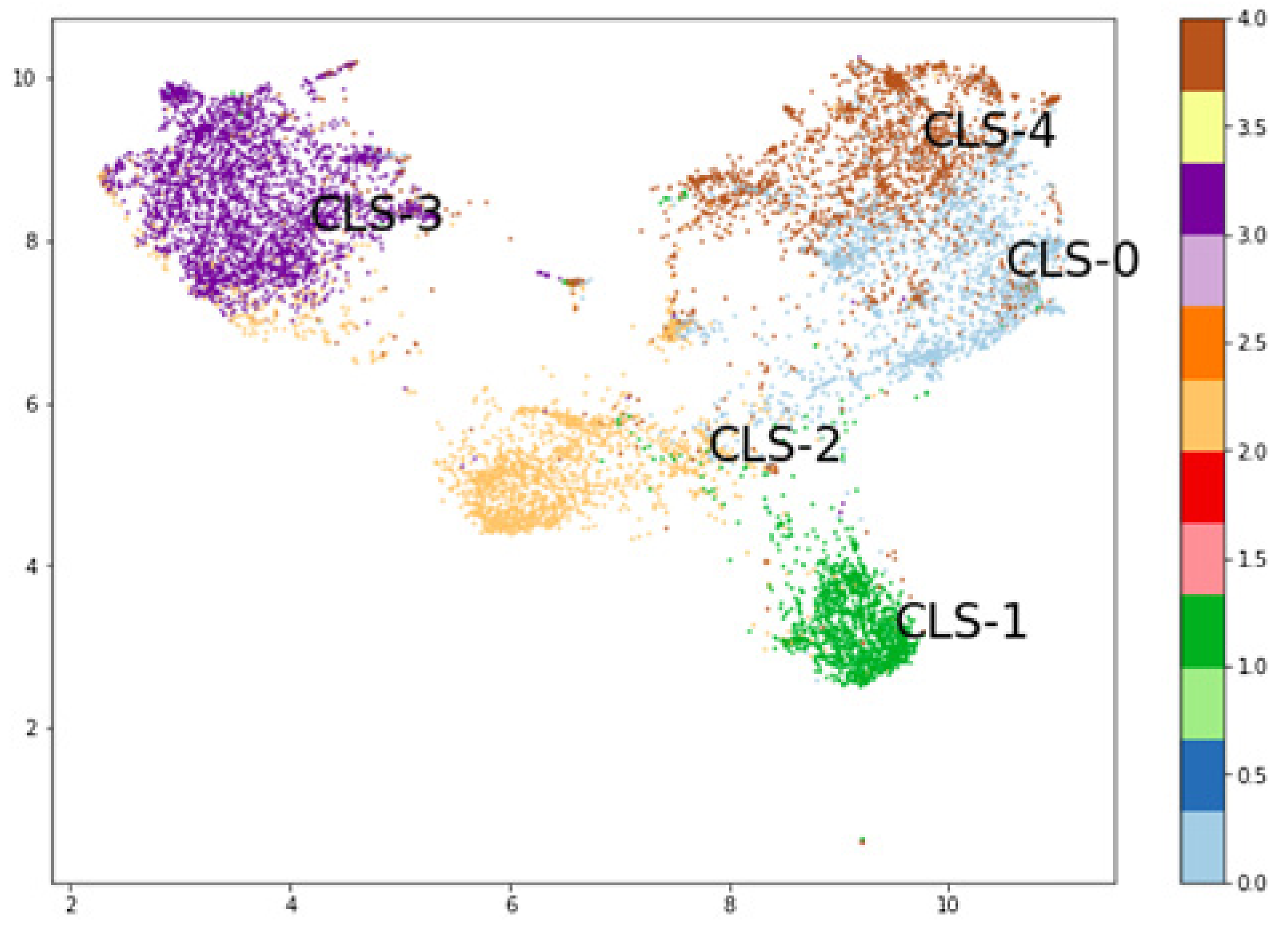


Figure 7.8 – Cluster points visualization

In the preceding output, the points have been colored according to their cluster membership and centroids. It looks like we have picked the right number of clusters.

To capture the topics and interpret the clusters, we simply located the sentences (one single sentence for each cluster) close to the centroids of the clusters. Now, let's look at a more accurate way of capturing the topic with topic modeling.

Topic modeling with BERTopic

You may be familiar with many unsupervised topic modeling techniques that are used to extract topics from documents; **Latent-Dirichlet Allocation** (**LDA**) topic modeling and **Non-Negative Matrix Factorization** (**NMF**) are well-applied traditional techniques in the literature. BERTopic and Top2Vec are two important transformer-based topic modeling projects. In this section, we will apply the BERTopic model to our Amazon corpus. It leverages BERT embeddings and the class-based TF-IDF method to get easily interpretable topics.

First, the BERTopic model starts by encoding the sentences with sentence transformers or any sentence embedding model, which is followed by the clustering step. The clustering step has two phases: the embedding's dimensionality is reduced by **UMAP** and then the reduced vectors are clustered by **Hierarchical Density-Based Spatial Clustering of Applications with Noise** (**HDBSCAN**), which yields groups of similar documents. At the final stage, the topics are captured by cluster-wise TF-IDF, where the model extracts the most important words per cluster rather than per document and obtains descriptions of the topics for each cluster. Let's get started:

1. First, let's install the necessary library, as follows:

   Copy Explain
   ```
   !pip install bertopic
   ```

   Important note

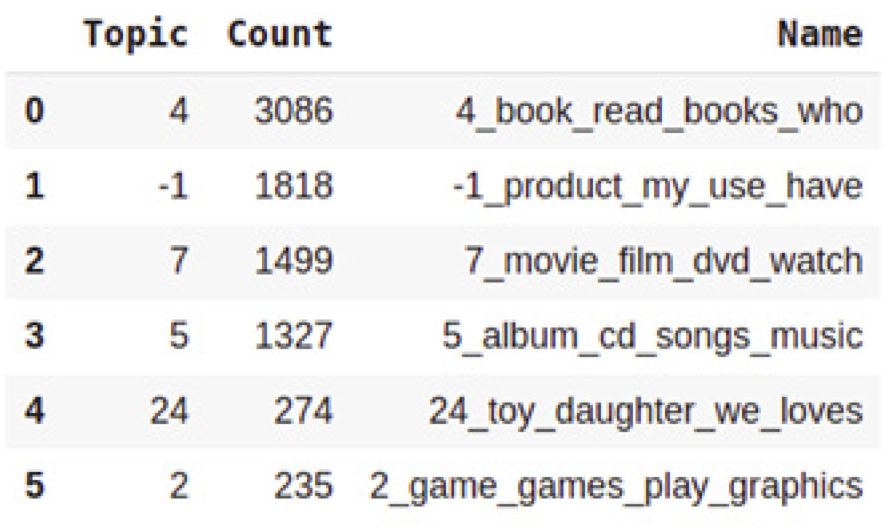
   You may need to restart the runtime since this installation will update some packages that have already been loaded. So, from the Jupyter notebook, go to **Runtime** | **Restart Runtime**.

2. If you want to use your own embedding model, you need to instantiate and pass it through the BERTopic model. We will instantiate a Sentence Transformer model and pass it to the constructor of BERTopic, as follows:

Copy Explain

```
from bertopic import BERTopic
sentence_model = SentenceTransformer(
                  "paraphrase-distilroberta-base-v1")
topic_model = BERTopic(embedding_model=sentence_model)
     topics, _ = topic_model.fit_transform(corpus)
     topic_model.get_topic_info()[:6]
```

The output is as follows:



	Topic	Count	Name
0	4	3086	4_book_read_books_who
1	-1	1818	-1_product_my_use_have
2	7	1499	7_movie_film_dvd_watch
3	5	1327	5_album_cd_songs_music
4	24	274	24_toy_daughter_we_loves
5	2	235	2_game_games_play_graphics

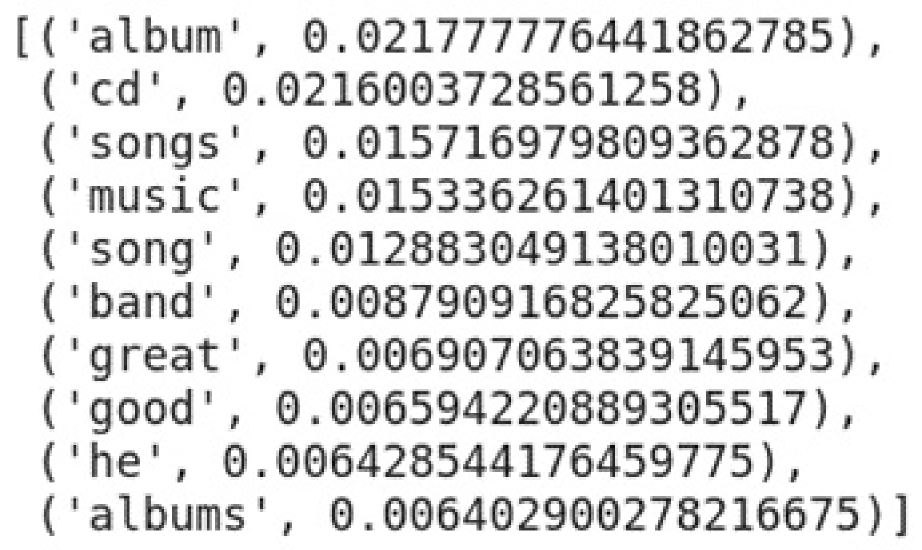
Figure 7.9 – BERTopic results

Please note that different BERTopic runs with the same parameters can yield different results since the UMAP model is stochastic. Now, let's see the word distribution of topic five, as follows:

Copy Explain

```
topic_model.get_topic(5)
```

The output is as follows:



```
[('album', 0.021777776441862785),
 ('cd', 0.0216003728561258),
 ('songs', 0.015716979809362878),
 ('music', 0.015336261401310738),
 ('song', 0.012883049138010031),
 ('band', 0.008790916825825062),
 ('great', 0.006907063839145953),
 ('good', 0.006594220889305517),
 ('he', 0.006428544176459775),
 ('albums', 0.006402900278216675)]
```

Figure 7.10 – The fifth topic words of the topic model

The topic words are those words whose vectors are close to the topic vector in the semantic space. In this experiment, we did not cluster the corpus; instead, we applied the technique to the entire corpus. In our previous example, we analyzed the clusters with the closest sentence. Now, we can find the topics by applying the topic model separately to each cluster. This is pretty straightforward, and you can run it yourself.

Please see the Top2Vec project for more details and interesting topic modeling applications at https://github.com/ddangelov/Top2Vec.

Semantic search with Sentence-BERT

We may already be familiar with keyword-based search (Boolean model), where, for a given keyword or pattern, we can retrieve the results that match the pattern. Alternatively, we can use regular expressions, where we can define advanced patterns such as the lexico-syntactic pattern. These traditional approaches cannot handle synonym (for example, *car* is the same as *automobile*) or word sense problems (for example, *bank* as the side of a river or *bank* as a financial institute). While the first synonym case causes low recall due to missing out the documents that shouldn't be missed, the second causes low precision due to catching the documents not to be caught. Vector-based or semantic search approaches can overcome these drawbacks by building a dense numerical representation of both queries and documents.

Let's set up a case study for **Frequently Asked Questions (FAQs)** that are idle on websites. We will exploit FAQ resources within a semantic search problem. FAQs contain frequently asked questions. We will be using the FAQ from the **World Wide Fund for Nature (WWF)**, a nature non-governmental organization (https://www.wwf.org.uk/).

Given these descriptions, it is easy to understand that performing a semantic search using semantic models is very similar to a one-shot learning problem, where we just have a single shot of the class (a single sample), and we want to reorder the rest of the data (sentences) according to it. You can redefine the problem as searching for samples that are semantically close to the given sample, or a binary classification according to the sample. Your model can provide a similarity metric, and the results for all the other samples will be reordered using this metric. The final ordered list is the search result, which is reordered according to semantic representation and the similarity metric.

WWF has 18 questions and answers on their web page. We defined them as a Python list object called `wf_faq` for this experiment:

- I haven't received my adoption pack. What should I do?
- How quickly will I receive my adoption pack?
- How can I renew my adoption?
- How do I change my address or other contact details?
- Can I adopt an animal if I don't live in the UK?
- If I adopt an animal, will I be the only person who adopts that animal?
- My pack doesn't contain a certificate?
- My adoption is a gift but won't arrive on time. What can I do?
- Can I pay for an adoption with a one-off payment?
- Can I change the delivery address for my adoption pack after I've placed my order?
- How long will my adoption last for?
- How often will I receive updates about my adopted animal?
- What animals do you have for adoption?
- How can I find out more information about my adopted animal?
- How is my adoption money spent?
- What is your refund policy?
- An error has been made with my Direct Debit payment; can I receive a refund?
- How do I change how you contact me?

Users are free to ask any question they want. We need to evaluate which question in the FAQ is the most similar to the user's question, which is the objective of the `quora-distilbert-base` model. There are two options in the SBERT hub – one is for English and another for multilingual, as follows:

- `quora-distilbert-base`: This is fine-tuned for Quora Duplicate Questions detection retrieval.

`quora-distilbert-multilingual`: This is a multilingual version of `quora-distilbert-base`. It's fine-tuned with parallel data for 50+ languages.

Let's build a semantic search model by following these steps:

1. The following is the SBERT model's instantiation:

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('quora-distilbert-base')
```

2. Let's encode the FAQ, as follows:

```
faq_embeddings = model.encode(wwf_faq)
```

3. Let's prepare five questions so that they are similar to the first five questions in the FAQ, respectively; that is, our first test question should be similar to the first question in the FAQ, the second question should be similar to the second question, and so on, so that we can easily follow the results. Let's define the questions in the `test_questions` list object and encode it, as follows:

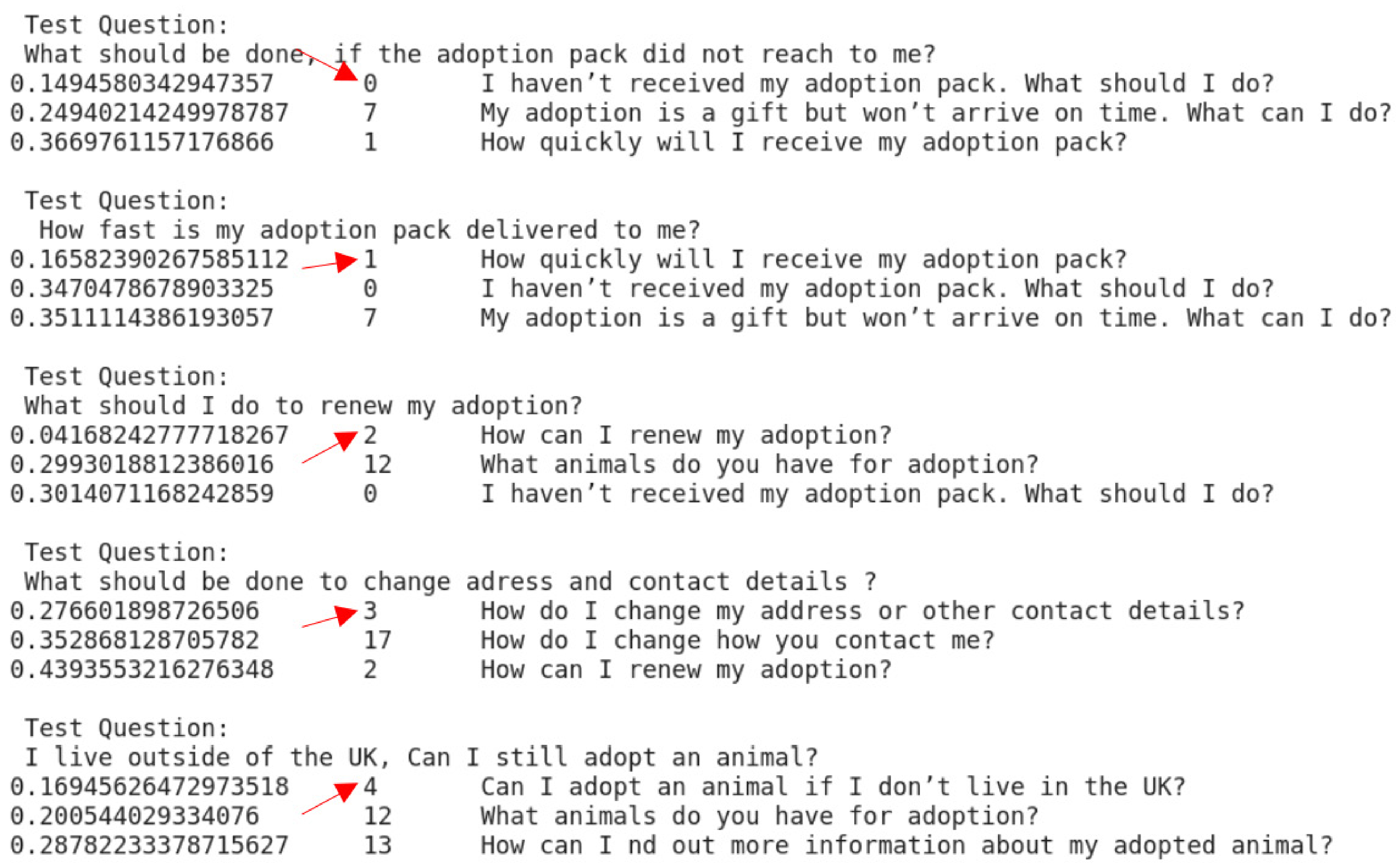
```
test_questions=["What should be done, if the adoption pack did not reach to me?",
" How fast is my adoption pack delivered to me?",
"What should I do to renew my adoption?",
"What should be done to change address and contact details ?",
"I live outside of the UK, Can I still adopt an animal?"]
test_q_emb= model.encode(test_questions)
```

4. The following code measures the similarity between each test question and each question in the FAQ and then ranks them:

Copy Explain

```
from scipy.spatial.distance import cdist
for q, qe in zip(test_questions, test_q_emb):
    distances = cdist([qe], faq_embeddings, "cosine")[0]
    ind = np.argsort(distances, axis=0)[:3]
    print("\n Test Question: \n "+q)
    for i,(dis,text) in enumerate(
                       zip(
                       distances[ind],
                       [wwf_faq[i] for i in ind])):
        print(dis,ind[i],text, sep="\t")
```

The output is as follows:



```
 Test Question:
 What should be done, if the adoption pack did not reach to me?
0.1494580342947357      0       I haven't received my adoption pack. What should I do?
0.24940214249978787     7       My adoption is a gift but won't arrive on time. What can I do?
0.3669761157176866      1       How quickly will I receive my adoption pack?

 Test Question:
  How fast is my adoption pack delivered to me?
0.16582390267585112     1       How quickly will I receive my adoption pack?
0.3470478678903325      0       I haven't received my adoption pack. What should I do?
0.3511114386193057      7       My adoption is a gift but won't arrive on time. What can I do?

 Test Question:
 What should I do to renew my adoption?
0.04168242777718267     2       How can I renew my adoption?
0.2993018812386016      12      What animals do you have for adoption?
0.3014071168242859      0       I haven't received my adoption pack. What should I do?

 Test Question:
 What should be done to change adress and contact details ?
0.276601898726506       3       How do I change my address or other contact details?
0.352868128705782       17      How do I change how you contact me?
0.4393553216276348      2       How can I renew my adoption?

 Test Question:
 I live outside of the UK, Can I still adopt an animal?
0.16945626472973518     4       Can I adopt an animal if I don't live in the UK?
0.200544029334076       12      What animals do you have for adoption?
0.28782233378715627     13      How can I nd out more information about my adopted animal?
```

Figure 7.11 – Question-question similarity

Here, we can see indexes `0`, `1`, `2`, `3`, and `4` in order, which means the model successfully found the similar questions as expected.

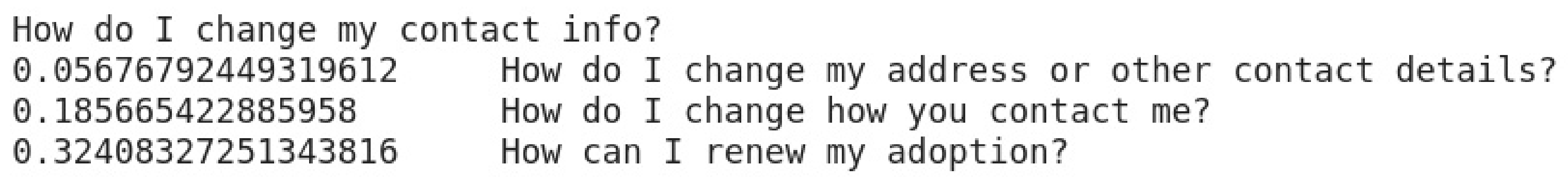
5. For the deployment, we can design the following `getBest()` function, which takes a question and returns `K` most similar questions in the FAQ:

```
def get_best(query, K=5):
    query_emb = model.encode([query])
    distances = cdist(query_emb,faq_embeddings,"cosine")[0]
    ind = np.argsort(distances, axis=0)
    print("\n"+query)
    for c,i in list(zip(distances[ind], ind))[:K]:
        print(c,wwf_faq[i], sep="\t")
```

6. Let's ask a question:

```
get_best("How do I change my contact info?",3)
```

The output is as follows:



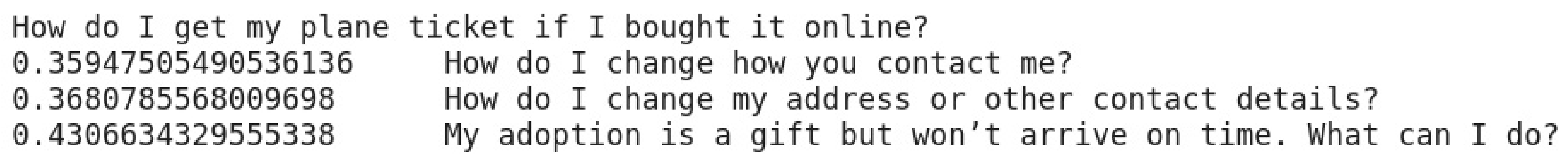
```
How do I change my contact info?
0.05676792449319612     How do I change my address or other contact details?
0.185665422885958       How do I change how you contact me?
0.32408327251343816     How can I renew my adoption?
```

Figure 7.12 – Similar question similarity results

7. What if a question that's used as input is not similar to one from the FAQ? Here is such a question:

```
get_best("How do I get my plane ticket \
    if I bought it online?")
```

The output is as follows:



```
How do I get my plane ticket if I bought it online?
0.35947505490536136     How do I change how you contact me?
0.3680785568009698      How do I change my address or other contact details?
0.4306634329555338      My adoption is a gift but won't arrive on time. What can I do?
```

Figure 7.13 – Dissimilar question similarity results

The best dissimilarity score is 0.35. So, we need to define a threshold such as 0.3 so that the model ignores such questions that are higher than that threshold and says `no similar answer found`.

Other than question-question symmetric search similarity, we can also utilize SBERT's question-answer asymmetric search models, such as `msmarco-distilbert-base-v3`, which is trained on a dataset of around 500K Bing search queries. It is known as

MSMARCO Passage Ranking. This model helps us measure how related the question and context are and checks whether the answer to the question is in the passage.

Summary

In this chapter, we learned about text representation methods. We learned how it is possible to perform tasks such as zero-/few-/one-shot learning using different and diverse semantic models. We also learned about NLI and its importance in capturing semantics of text. Moreover, we looked at some useful use cases such as semantic search, semantic clustering, and topic modeling using Transformer-based semantic models. We learned how to visualize the clustering results and understood the importance of centroids in such problems.

In the next chapter, you will learn about efficient Transformer models. You will learn about distillation, pruning, and quantizing Transformer-based models. You will also learn about different and efficient Transformer architectures that make improvements to computational and memory efficiency, as well as how to use them in NLP problems.

Further reading

Please refer to the following works/papers for more information about the topics that were covered in this chapter:

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., ... & Zettlemoyer, L. (2019). *Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension*. arXiv preprint arXiv:1910.13461.

Pushp, P. K., & Srivastava, M. M. (2017). *Train once, test anywhere: Zero-shot learning for text classification*. arXiv preprint arXiv:1712.05972.

Reimers, N., & Gurevych, I. (2019). *Sentence-bert: Sentence embeddings using siamese bert-networks*. arXiv preprint arXiv:1908.10084.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... & Stoyanov, V. (2019). *Roberta: A robustly optimized bert pretraining approach*. arXiv preprint arXiv:1907.11692.

Williams, A., Nangia, N., & Bowman, S. R. (2017). *A broad-coverage challenge corpus for sentence understanding through inference*. arXiv preprint arXiv:1704.05426.

Cer, D., Yang, Y., Kong, S. Y., Hua, N., Limtiaco, N., John, R. S., ... & Kurzweil, R. (2018). *Universal sentence encoder*. arXiv preprint arXiv:1803.11175.

Yang, Y., Cer, D., Ahmad, A., Guo, M., Law, J., Constant, N., ... & Kurzweil, R. (2019). *Multilingual universal sentence encoder for semantic retrieval*. arXiv preprint arXiv:1907.04307.

Humeau, S., Shuster, K., Lachaux, M. A., & Weston, J. (2019). *Poly-encoders: Transformer architectures and pre-training strategies for fast and accurate multi-sentence scoring.* arXiv preprint arXiv:1905.01969.

Previous Chapter

Next Chapter