When studying transformer models, we tend to focus on their architecture and the datasets provided to train them. This book covers the original Transformer, BERT, RoBERTa, ChatGPT, GPT4, PaLM, LaMBDA, DALL-E, and more. In addition, the book reviews several benchmark tasks and datasets. We have fine-tuned a BERT-like model and trained a RoBERTa tokenizer using tokenizers to encode data. In the previous *Chapter 9, Shattering the Black Box with Interpretable Tools*, we also shattered the black box and analyzed the inner workings of a transformer model.However, we did not explore the critical role tokenizers play and evaluate how they shape the models we build. AI is data-driven. *Raffel* et al. (2019), like all the authors cited in this book, spent time preparing datasets for transformer models. In this chapter, we will go through some of the issues of tokenizers that hinder or boost the performance of transformer models. Do not take tokenizers at face value. You might have a specific dictionary of words you use (advanced medical language, for example) with words poorly processed by a generic tokenizer.We will start by introducing some tokenizer-agnostic best practices to measure the quality of a tokenizer. We will describe basic guidelines for datasets and tokenizers from a tokenization perspective.Then, we will see some of the possible limits of tokenizers with a Word2Vec tokenizer to describe the problems we face with any tokenizing method. The limits will be illustrated with a Python program.We will continue our investigation by exploring word and subword tokenizers.We will begin with sentence and word tokenizers. These tokenizers provide valuable natural language processing tools. However, they do not match the more efficient subword tokenizers for transformer model training.Therefore, we will continue with the more efficient subword tokenizers for transformer models. Subword tokenizers show how a tokenizer can shape a transformer model's training and performance.We will see how to detect which subword tokenizer was applied to create a dictionary. Finally, we will build a function to display and control the token-ID mappings.This chapter covers the following topics:

- Basic guidelines to control the output of tokenizers
- Raw data strategies and preprocessing data strategies
- Word2Vec tokenization problems and limits
- Creating a Python program to evaluate Word2Vec tokenizers
- Sentence and word tokenizers
- Subword tokenizers
- Tokenizer detection
- Displaying and controlling token-ID mappings

Our first step will be exploring the text-to-text methodology Raffel et al. (2019) defined.

Matching datasets and tokenizers

Downloading benchmark datasets to train transformers has many advantages. The data has been prepared, and every research lab uses the same references. Also, the performance of a transformer model can be compared to another model with the same data.However, more needs to be done to improve the performance of transformers. Furthermore, implementing a transformer model in production requires careful planning and defining best practices.In this section, we will define some best practices to avoid critical stumbling blocks.Then, we will go through a few examples in Python using cosine similarity to measure the limits of tokenization and encoding datasets.Let's start with best practices.

Best practices

Raffel et al. (2019) defined a standard text-to-text T5 transformer model. They also went further. They contributed to the destruction of the myth of using raw data without preprocessing it first.Preprocessing data reduces training time. Common Crawl, for example, contains unlabeled text obtained through web extraction. Non-text and markup have been removed from the dataset.However, the Google T5 team found that much of the text obtained through Common Crawl did not reach the level of natural language or English. So, they decided that datasets need to be cleaned before using them.We will take the recommendations *Raffel* et al. (2019) made and apply corporate quality control best practices to the preprocessing and quality control phases. Among many other rules to apply, the examples described show the tremendous work required to obtain acceptable real-life project datasets.Quality control is divided into the preprocessing phase (*Step 1*) when training a transformer and quality control when the transformer is in production (*Step 2*). A third phase (*Step 3*) has become mandatory for cutting-edge technology: *Continuous human quality control.*Let's go through some of the main aspects of the preprocessing phase.

Step 1: Preprocessing

Raffel et al. (2019) recommended preprocessing datasets before training models on them, and I added some extra ideas.Transformers have become language learners, and we have become their teachers. But to teach a machine-student a language, we must explain what proper English is, for example.We need to apply some standard heuristics to datasets before using them:**Sentences with punctuation marks**The recommendation is to select sentences that end with punctuation marks such as a period or a question mark.**Remove bad words**Bad words should be removed. Lists can be found at the following site, for example https://github.com/LDNOOBW/List-of-Dirty-Naughty-Obscene-and-Otherwise-Bad-Words.**Remove code**This is tricky because sometimes, code is the content we are looking for. However, it is generally best to remove code from content for NLP tasks.**Language detection**Sometimes, websites contain pages with the default "lorem ipsum" text. It is necessary to ensure that all of a dataset's content is in the language we wish. An excellent way to start is with `langdetect`, which can detect 50+ languages: https://pypi.org/project/langdetect/.**Removing references to discrimination**This is a must. I recommend building a knowledge base with everything you can scrape on the web or from specific datasets you can get your hands on. *Suppress any form of discrimination*. You certainly want your machine to be ethical!**Logic check**It could be a good idea to run a trained transformer model on a dataset that performs **Natural Language Inferences** (**NLI**) to filter sentences that make no sense.**Bad information references**Eliminate text that refers to links that do not work, unethical websites, or people. This is a tough job, but certainly worthwhile.This list contains some of the primary best practices. However, more is required, such as filtering privacy law violations and other actions for specific projects.Once a transformer is trained to learn proper English, we must help it detect problems in the input texts in the production phase.

Step 2: Quality control

A trained model will behave like a person who learned a language. It will understand what it can and learn from input data. Input data should go through the same process as *Step 1: Preprocessing* and adding new information to the training dataset. The training dataset, in turn, can become the knowledge base of a corporate project. Users will be able to run NLP tasks on the dataset and obtain reliable answers to questions, useful summaries of specific documents, and more.We should apply the best practices described in *Step 1: Preprocessing* to real-time input data. For example, a transformer can be running on input from a user or an NLP task, such as summarizing a list of documents.Transformers are the most powerful NLP models ever. This means that our ethical responsibility is heightened as well.Let's go through some of the best practices:**Check input text in real time**Do not accept bad information. Instead, parse the input in real time and filter the unacceptable data (see *Step 1*).**Real-time messages**Store the rejected data and the reason it was filtered so that users can consult the logs. Display real-time messages if a transformer is asked to answer an unfitting question.**Language conversions**You can convert rare vocabulary into standard vocabulary when it is possible. See *Case 4* of the *Word2Vec tokenization* section in this chapter. This is not always possible. When it is, it could represent a step forward.**Privacy checks**Whether you are streaming data into a transformer model or analyzing user input, private data must be excluded from the dataset and tasks unless authorized by the user or country the transformer is running in. It's a tricky topic. Consult a legal adviser when necessary.We just went through some of the best practices. Let's now see why human quality control is mandatory.

Step 3: Continuous human quality control

Cutting-edge software quality control requires continual human intervention to monitor innovations and improve the systems in production. Transformers will progressively take over most of the complex NLP tasks. However, human intervention remains mandatory. We think social media giants have automized everything. Then, we discover content managers sometimes manually decide what is good or bad for their platform.The right approach is to train a transformer, implement it, control the output, and feed the significant results back into the training set. Thus, the training set will continuously improve, and the transformer will continue to learn.*Figure 10.2* shows how continuous quality control will help the transformer's training dataset grow and increase its performance in production:

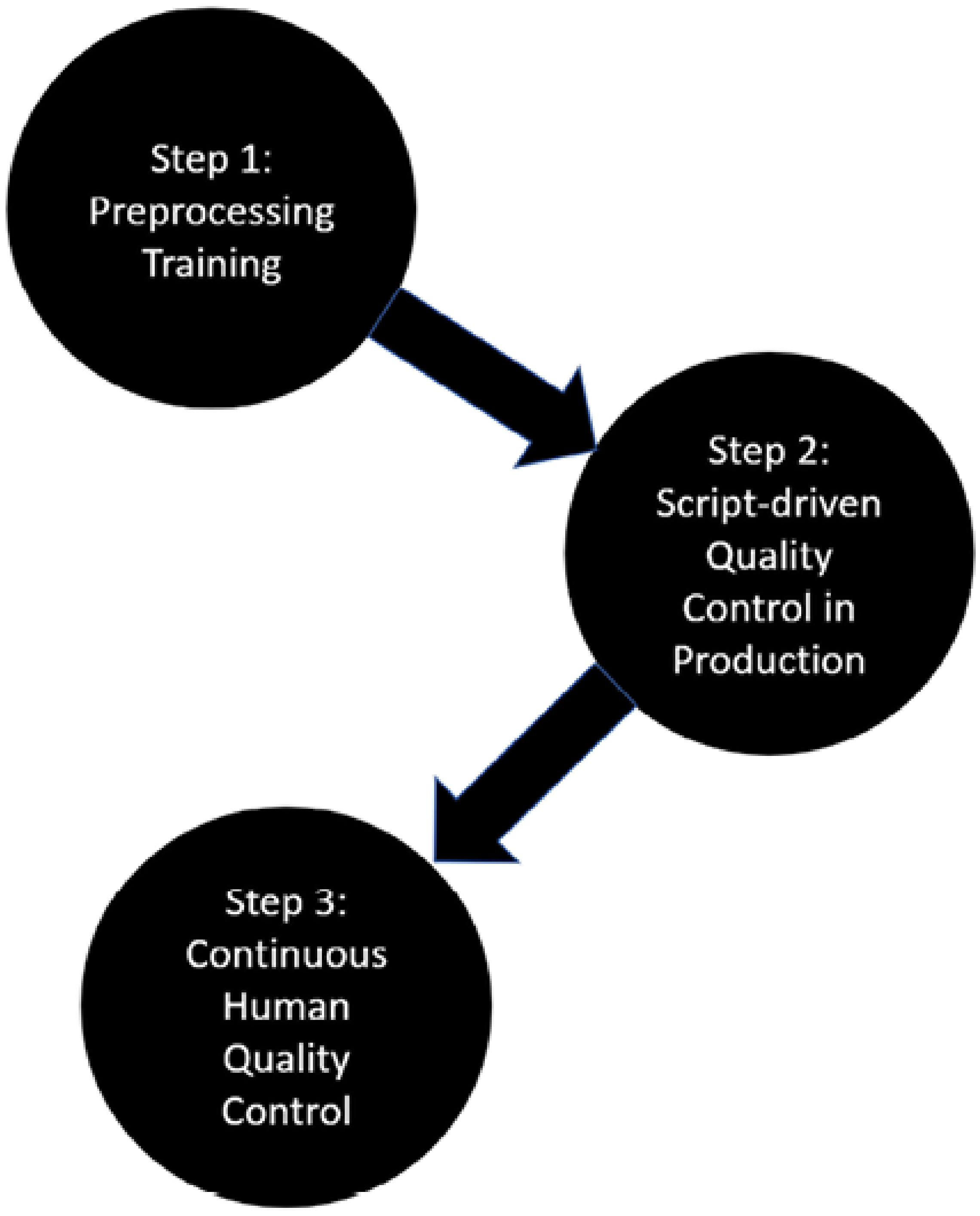

Figure 10.2:Once the training is done and the model is in production with its quality control, continuous human quality control is required to measure the overall lifecycle of the system.

We have gone through several best practices that Raffel et al. (2019) described, and I have added some guidance based on my experience in corporate AI project management.Let's go through a Python program with some examples of some of the limits encountered with tokenizers.

Word2Vec tokenization

As long as things go well, nobody thinks about pretrained tokenizers. It's like in real life. We can drive a car for years without thinking about the engine. Then, one day, our car breaks down, and we try to find the reasons to explain the situation.The same happens with pretrained tokenizers. Sometimes, the results are not what we expect. For example, some word pairs don't fit together in the context of the text of the Declaration of Independence, as we can see in *Figure 10.3*:

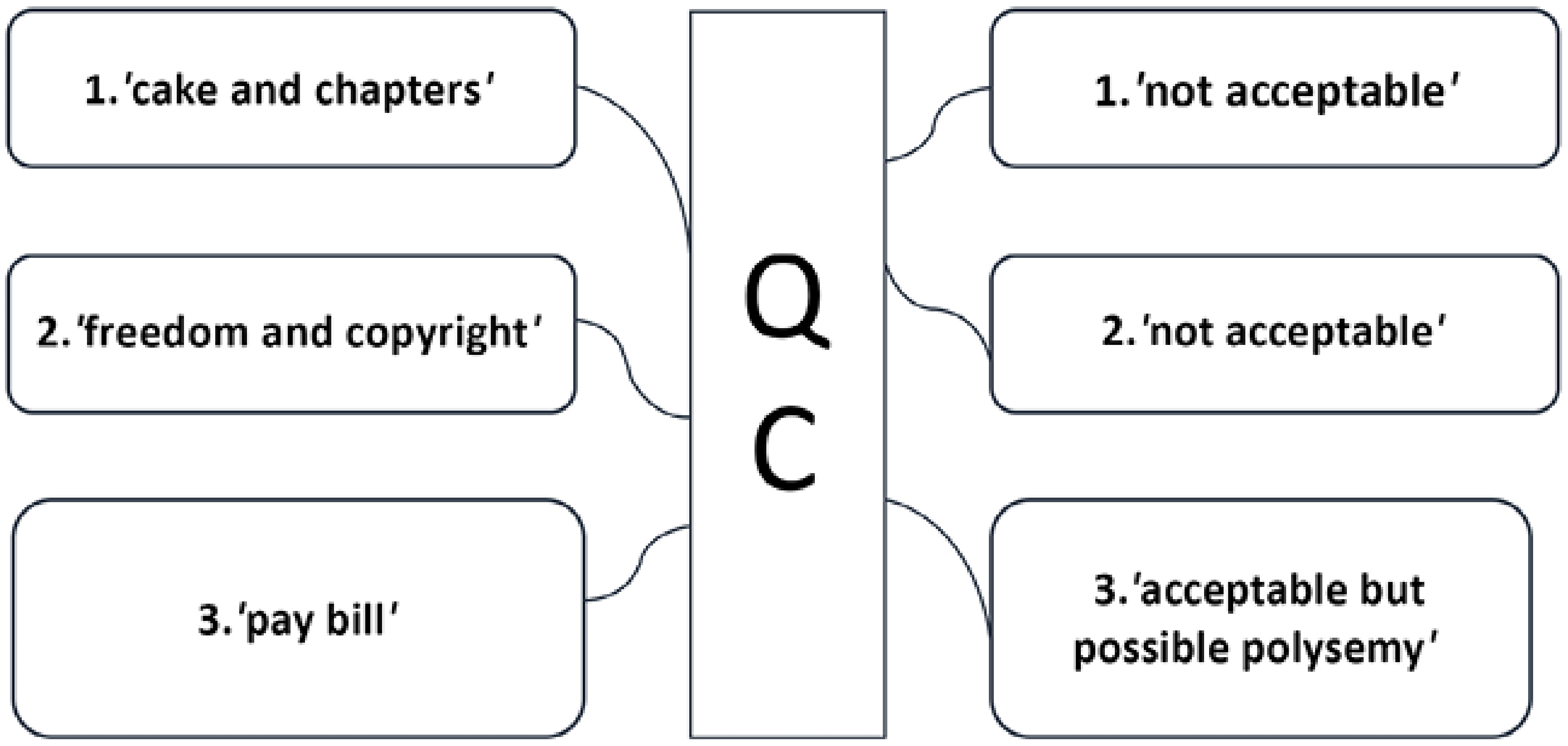

Figure 10.3: Word pairs that tokenizers miscalculated and detected during QC (Quality Control)

The examples shown in *Figure 10.3* are drawn from the *American Declaration of Independence*, the *Bill of Rights*, and the *English Magna Carta*:`cake` and `chapters` do not fit together, although a tokenizer computed them as having a high cosine similarity value.`freedom` refers to the freedom of speech, for example. `copyright` refers to the note written by the editor of the free ebook.`pay` and `bill` fit together in everyday English. `polysemy` is when a word can have several meanings. For example, `Bill` means an amount to pay but also refers to the `Bill of Rights`. The result is acceptable, but it may be pure luck.Before continuing, let's take a moment to clarify some points. **QC** refers to **quality control**. In any strategic corporate project, QC is mandatory. The quality of the output will determine the survival of a critical project. If the project is not strategic, errors will sometimes be acceptable. In a strategic project, even a few errors imply a risk management audit's intervention to see if the project should be continued or abandoned.From the perspectives of quality control and risk management, tokenizing irrelevant datasets (too many useless words or missing critical words) will confuse the embedding algorithms and produce "poor results." That is why I use the word "tokenizing" loosely in this chapter, including some embedding because of the impact of one upon the other.In a strategic AI project, "poor results" can be a single error with a dramatic consequence (especially in the medical sphere, airplane or rocket assembly, or other critical domains).Open `Tokenizer.ipynb`, based on `positional_encoding.ipynb`, which we created in *Chapter 2, Getting Started with the Architecture of the Transformer Model*.Results might vary from one run to another due to the stochastic nature of Word2Vec algorithms.The prerequisites are installed and imported first:

Copy Explain

```
!pip install gensim==3.8.3
import nltk
nltk.download('punkt')
import math
import numpy as np
from nltk.tokenize import sent_tokenize, word_tokenize
import gensim
from gensim.models import Word2Vec
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings(action = 'ignore')
```

We now load our sample dataset:

Copy Explain

```
#1.Load text.txt using the Colab file manager
#2.Downloading the file from GitHub
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers-for-NLP-and-Computer-
Vision-3rd-Edition/master/Chapter09/text.txt --output "text.txt"
```

`text.txt`, our dataset, contains the *American Declaration of Independence*, the *Bill of Rights*, the *Magna Carta*, the works of Immanuel Kant, and other texts.We will now tokenize `text.txt` and train a word2vec model:

Copy Explain

```
#'text.txt' file
sample = open("text.txt", "r")
s = sample.read()
# processing escape characters
f = s.replace("\n", " ")
data = []
# Sentence parsing
for i in sent_tokenize(f):
  temp = []
  # tokenize the Sentence into words
  for j in word_tokenize(i):
    temp.append(j.lower())
  data.append(temp)
# Creating Skip Gram model
model2 = gensim.models.Word2Vec(data, min_count = 1, size = 512,window = 5, sg = 1)
print(model2)
```

`window = 5` is an interesting parameter. It limits the *distance* between the current word and the predicted word in an input sentence. `sg = 1` means a skip-gram training algorithm is used.The output shows that the size of the vocabulary is `10816`, the dimensionality of the embeddings is `512`, and the learning rate was set to `alpha=0.025`:

Copy Explain

```
Word2Vec(vocab=10816, size=512, alpha=0.025)
```

We have a word representation model with embedding and can create a cosine similarity function named `similarity(word1,word2)`. We will send `word1` and `word2` to the function, which will return a cosine similarity value between them. The higher the value, the higher the similarity.The function will first detect unknown words, `[unk]`, and display a message:

Copy Explain

```
def similarity(word1,word2):
        cosine=False #default value
        try:
                a=model2[word1]
                cosine=True
        except KeyError:     #The KeyError exception is raised
                print(word1, ":[unk] key not found in dictionary")#False implied
        try:
                b=model2[word2]#a=True implied
        except KeyError:       #The KeyError exception is raised
                cosine=False   #both a and b must be true
                print(word2, ":[unk] key not found in dictionary")
```

Cosine similarity will only be calculated if `cosine==True`, which means that both `word1` and `word2` are known:

Copy Explain

```
        if(cosine==True):
                b=model2[word2]
                # compute cosine similarity
                dot = np.dot(a, b)
                norma = np.linalg.norm(a)
                normb = np.linalg.norm(b)
                cos = dot / (norma * normb)
                aa = a.reshape(1,512)
                ba = b.reshape(1,512)
                #print("Word1",aa)
                #print("Word2",ba)
                cos_lib = cosine_similarity(aa, ba)
                #print(cos_lib,"word similarity")

        if(cosine==False):cos_lib=0;
        return cos_lib
```

The function will return `cos_lib`, the computed value of cosine similarity.We will now go through six cases. We will name `text.txt` the "dataset."Let's begin with *Case 0*.

Case 0: Words in the dataset and the dictionary

The words `freedom` and `liberty` are in the dataset, and their cosine similarity can be computed:

Copy Explain

```
word1="freedom";word2="liberty"
print("Similarity",similarity(word1,word2),word1,word2)
```

The similarity is limited to `0.79` because a lot of content was inserted from various texts to explore the limits of the function:

Copy Explain

```
Similarity [[0.79085565]] freedom liberty
```

The similarity algorithm is not an iterative deterministic calculation. It's a stochastic algorithm. This section's results might change with the dataset's content, the dataset's size after another run, or the module's versions. For example, if you run the cell 10 times, you may or may not obtain different values, such as in the following 10 runs.In the following case, I obtained the same result 10 times with a Google Colab VM and a CPU:

```
Run 1: Similarity [[0.62018466]] freedom liberty
Run 2: Similarity [[0.62018466]] freedom liberty
...
Run 10: Similarity [[0.62018466]] freedom liberty
```

However, I did a "factory reset runtime" of the runtime menu in Google Colab. With a new VM and a CPU, I obtained:

```
Run 1: Similarity [[0.51549244]] freedom liberty
Run 2: Similarity [[0.51549244]] freedom liberty
...
Run 10: Similarity [[0.51549244]] freedom liberty
```

I performed another "factory reset runtime" of the runtime menu in Google Colab. I also activated the GPU. With a new VM and GPU, I obtained:

```
Run 1: Similarity [[0.58365834]] freedom liberty
Run 2: Similarity [[0.58365834]] freedom liberty
...
Run 10: Similarity [[0.58365834]] freedom liberty
```

The conclusion here is that stochastic algorithms are based on probabilities. Therefore, running a prediction n times if necessary is good practice.Let's now see what happens when a word is missing.

Case 1: Words not in the dataset or the dictionary

A missing word means trouble in many ways. In this case, we send `corporations` and `rights` to the similarity function:

```
word1="corporations";word2="rights"
print("Similarity",similarity(word1,word2),word1,word2)
```

The dictionary does not contain the word `corporations`:

```
corporations :[unk] key not found in dictionary
Similarity 0 corporations rights
```

Dead end! The word is an unknown `[unk]` token.The missing word will provoke a chain of events and problems that distort the transformer model's output if the word is important. We will refer to the missing word as `unk`.Several possibilities need to be checked and questions answered:`unk` was in the dataset but was not selected to be in the tokenized dictionary.`unk` was not in the dataset, which is the case for the word `corporations`. This explains why it's not in the dictionary in this case.`unk` will now appear in production if a user sends an input to the transformer that contains the token and it is not tokenized.`unk` was not an important word for the dataset but was for the usage of the transformer.The list of problems will continue to grow if the transformer produces terrible results in some cases. We can consider `0.8` to be an excellent performance for a transformer model for a specific downstream task during the training phase. But in real life, who wants to work with a system that's wrong 20% of the time:

> A doctor?
>
> A lawyer?
>
> A nuclear plant maintenance team?

`0.8` is satisfactory in a fuzzy environment like social media, in which many of the messages lack proper language structure anyway.Now comes the worst part. Suppose an NLP team discovers this problem and tries to solve it with byte-level BPE(Byte-Pair Encoding), as we have been doing throughout this book. If necessary, take a few minutes and return to *Chapter 6, Pretraining a Transformer from Scratch through RoBERTa , Step 3: Training a tokenizer*.The nightmare begins if a team only uses byte-level BPE to fix the problem:`unk` will be broken down into word pieces. For example, we could end up with `corporations` becoming `corp` + `o` + `ra` + `tion` + `s`. One or several of these tokens have a high probability of being found in the dataset.`unk` will become a set of sub-words represented by tokens that exist in the dataset but do not convey the original token's meaning.The transformer will train well, and nobody will notice that `unk` was broken into pieces and trained meaninglessly.The transformer might produce excellent results and increase its performance from `0.8` to `0.9`.Everybody will applaud until a professional user applies an erroneous result in a critical situation. For example, in English, `corp` can mean `corporation` or `corporal`. This could create confusion and bad associations between `corp` and other words.We can see that the standard of social media might be enough to use transformers for trivial topics. But in real-life corporate projects, producing a pretrained tokenizer that matches the datasets will take hard work. In real life, datasets grow every day with user inputs. User inputs become part of the datasets of models that should be trained and updated regularly.For example, one way to ensure quality control can be through the following 3 steps:

Step 1: Train a tokenizer with a byte-level BPE algorithm.

Step 2: Control the results with a program such as the one we will create in the *Controlling tokenized data* section of this chapter.

Step 3: Also, train a tokenizer with a Word2Vec algorithm, which will only be used for quality control, then parse the dataset, find the `unk` tokens, and store them in the database. Run queries to check if critical words are missing.It might seem unnecessary to check the process in such detail, and you might be tempted to rely on a transformer's ability to make inferences with unseen words.However, I recommend running several quality control methods in a strategic project with critical decision-making. For example, in a legal summary of a law, one word can make the difference between losing and winning a case in court. In an aerospace project (airplanes, rockets), there is a `0` error tolerance standard.The more quality control processes you run, the more reliable your transformer solution will be.We can see that it takes much legwork to obtain a reliable dataset! Every paper written on transformers refers in one way or another to the work it took to produce acceptable datasets.Noisy relationships also cause problems.

Case 2: Noisy relationships

In this case, the dataset contained the words `etext` and `declaration`:

Copy Explain

```
word1="etext";word2="declaration"
print("Similarity",similarity(word1,word2),word1,word2)
```

Furthermore, they both ended up in the tokenized dictionary:

Copy Explain

```
Similarity [[0.880751]] etext declaration
```

Even better, their cosine similarity seems to be sure about its prediction and exceeds `0.5`. The stochastic nature of the algorithm might produce different results on various runs.At a trivial or social media level, everything looks good.However, at a professional level, the result is disastrous!`etext,` which is a word in the text file processed in the notebook, refers to *Project Gutenberg*'s preface to each ebook on their site, as explained in the *Matching datasets and tokenizers* section of this chapter. This means that the word `etext` is the editor's text file but has nothing to do with the `declaration`(Declaration of Independence. What is the goal of the transformer for a specific task:

> To understand an editor's preface?
>
> Or to understand the content of the book?

It depends on the usage of the transformer and might take a few days to sort out. For example, suppose an editor wants to understand prefaces automatically and uses a transformer to generate preface text. Should we take the content out? `declaration` is a meaningful word related to the actual content of the *Declaration of Independence*.`etext` is part of a preface that *Project Gutenberg* adds to its ebooks.This might produce erroneous natural language inferences, such as *etext is a declaration* when the transformer is asked to generate text. *etext,* a word used by the editor of the file, has nothing to do with *declaration* in the text file we are processing. *declaration* is part of the Declaration of Independence. The Declaration of Independence dates back to 1776, and *etext* (electronic texts) date back to the 20th century. An NLP model that would speak about the Declaration of Independence, including electronic text vocabulary, would be making an error.Let's look into a missing word issue.

Case 3: Words in the text but not in the dictionary

Sometimes, a word may be in a text but not in the dictionary. This will distort the results.Let's take the words `pie` and `logic`:

```
word1="pie";word2="logic"
print("Similarity",similarity(word1,word2),word1,word2)
```

The word `pie` is not in the dictionary:

```
pie :[unk] key not found in dictionary
Similarity 0 pie logic
```

We can assume that the word `pie` would be in a tokenized dictionary. But what if it isn't, or another word isn't? The word `pie` is not in the text file.Therefore, we should have functions in the pipeline to detect words not in the dictionary and implement corrections or alternatives. Also, we should have functions in the pipeline to detect words in the datasets that may be important. For example, a project manager could run hundreds of documents through the tokenizer to detect unknown words that would be stored in a file to analyze them. This is only an example, each project requires specific quality control actions.Let's see the problem we face with rare words.

Case 4: Rare words

Rare words produce devasting effects on the output of transformers for specific tasks that go beyond simple applications.Managing rare words extends to many domains of natural language. For example:

- Rare words can occur in datasets but go unnoticed, or models are poorly trained to deal with them.
- Rare words can be medical, legal, engineering terms, or any other professional jargon.
- Rare words can be slang.
- There are hundreds of variations of the English language. For example, different English words are used in certain parts of the United States, the United Kingdom, Singapore, India, Australia, and many other countries.
- Rare words can come from texts written centuries ago that are forgotten or that only specialists use.

For example, in this case, we are using the word `eleutheromania` (intense longing for freedom):

Copy Explain

```
word1="eleutheromania";word2="liberty"
print("Similarity",similarity(word1,word2),word1,word2)
```

The the system doesn't recognize `eleutheromania`:

Copy Explain

```
The word  eleutheromania  does not exist in the dictionary
Similarity 0 eleutheromania liberty
```

Unfortunately, if a rare word is used, the program will get confused, and we will obtain unexpected results after each run. If this occurs during a project, additional texts containing rare words that went unrecognized will have to be fed into the tokenizer training process.For example, if we implement a transformer model in a law firm to summarize documents or other tasks, we must be careful!Let's now see some methods we could use to solve a rare word problem.

Case 5: Replacing rare words

Replacing rare words represents a project in itself. This work is reserved for specific tasks and projects. Suppose a corporate budget can cover the cost of having a knowledge base in aeronautics, for example. In that case, it is worth spending time querying the tokenized directory to find words it missed.Problems can be grouped by topic solved, and the knowledge base will be updated regularly.In *Case 4*, we stumbled on the word `eleutheromania`. We could replace the word `eleutheromania` with `freedom`, which conveys the same meta-concept:

Copy Explain

```
#@title Case 5: Replacing rare words
word1="freedom";word2="liberty"
print("Similarity",similarity(word1,word2),word1,word2)
```

It produces an interesting result:

Copy Explain

```
Similarity [[0.99956566]] freedom liberty
```

In any case, some rare words need to be replaced by more mainstream words.We could create queries with replacement words that we run until we find correlations that are over `0.9`, for example. Moreover, if we manage a critical legal project, we could have essential documents that contain rare words translated into standard English. Thus, the transformer's performance with NLP tasks would increase, and the knowledge base of the corporation would progressively increase.Now, we will see how to use cosine similarity for entailment verification.

Exploring Sentence and Wordpiece tokenizers to understand the efficiency of subword tokenizers for transformers

Transformer models commonly use BPE and Wordpiece tokenization. In this section, we will understand why choosing a subword tokenizer over other tokenizers significantly impacts transformer models. The goal of this section will thus be to first review some of the main word and Sentence tokenizers. We will continue and implement subword tokenizers. But, first, we will detect if the tokenizer is a BPE or a Wordpiece.Then, we'll create a function to display the token-ID mappings.Finally, we'll analyze and control the quality of token-ID mappings.The first step is to review some of the main word and Sentence tokenizers.

Word and sentence tokenizers

Choosing a tokenizer depends on the objectives of the NLP project. Although subword tokenizers are more efficient for transformer models, word and Sentence tokenizers provide useful functionality. Sentence and word tokenizers are useful for many tasks, including:

- Text processing at a sentence or word level can be sufficient for NLP tasks such as text classification. Also, splitting texts into sentences and words can be part of a pipeline to create a supervised training dataset for transformer models. The sentences obtained, for example, can be normalized, and then the labeling function is added to the output. From there, a subword tokenizer can take over to begin the pretraining process of a transformer.
- Sentence or word level tokenization can be sufficient for simpler machine learning tasks such as classifiers for spam detection, for example.
- Part-of-Speech (POS) tagging and Named Entity Recognition (NER) can benefit from word or sentence tokenization when performed at word level.

The choice of a tokenizer is thus a critical decision that will have a lasting impact on the NLP functions, machine learning, or transformer model to implement.Open `Exploring tokenizers.ipynb` in the GitHub repository of this chapter.The first cells install the necessary libraries to run this notebook:

Copy Explain

```
#Hugging Face Transformers
!pip install transformers
#Printing tabular data in Python
!pip install tabulate
#Natural Language Toolkit
!pip install nltk
We installed Hugging Face transformers for subword tokenizers, the tabulate library to
print tabular data, and the Natural Language Toolkit, nltk,  for the word and sentence
tokenizers.
```

Note: The titles of the sub-sections in this section are the same as in the notebook.

Let's use the Natural Language Toolkit to review some of the main word and Sentence tokenizers.

Copy Explain

```
from nltk.tokenize import sent_tokenize, word_tokenize, RegexpTokenizer, \
```

Sentence tokenization

Sentence tokenization splits a text into separate sentences. It breaks a paragraph or document into units of sentences:

Copy Explain

```
# Sentence Tokenization
text = "This is a sentence. This is another one."
sentences = sent_tokenize(text)
print("Sentence Tokenization:")
print(sentences)
print()
sent_tokenize(text) breaks the sequence into sentences with separators:
Sentence Tokenization:
['This is a sentence.', 'This is another one.']
```

Word tokenization

Word Tokenization breaks a sequence(Sentence, text) into individual words. It detects punctuation marks and whitespaces such as quotation marks, tabs, tabs, and newlines:

Copy Explain

```
# Word Tokenization
sentence = "This sentence contains several words."
words = word_tokenize(sentence)
print("Word Tokenization:")
print(words)
print()
word_tokenize(Sentence) produces words with separators:
Word Tokenization:
['This', 'sentence', 'contains', 'several', 'words', '.']
```

Regular Expression Tokenization

Regular Expressions Tokenization uses regular expressions. Therefore, the function can be customized to define rules and patterns:

Copy Explain

```
# Regular Expression Tokenization
tokenizer = RegexpTokenizer(r'\w+')
text = "Let's see how to tokenize a sentence."
tokens = tokenizer.tokenize(text)
print("Regular Expression Tokenization:")
print(tokens)
print()
RegexpTokenizer(r'\w+') interprets r as a raw character. Otherwise, the backlash would
be interpreted as an escape sequence. Then, it interprets \w as a character class for
any alphanumerical character. Finally, it interprets + as one or more. The output is a
well-defined segmentation of the Sentence:
Regular Expression Tokenization:
['Let', 's', 'see', 'how', 'to', 'tokenize', 'a', 'sentence']
```

Treebank Tokenization

The Treebank Tokenizer is based on the corpora of the University of Pennsylvania, which contains, among other annotations, parts-of-speech (POS), syntactic structures, and semantics roles:

Copy Explain

```
# Treebank Tokenization
tokenizer = TreebankWordTokenizer()
text = "There aren't that many tokenizers."
tokens = tokenizer.tokenize(text)
print("Treebank Tokenization:")
print(tokens)
print()
The TreebankWordTokenizer() breaks a sequence down into words, taking into account
advanced issues such as contractions:
Treebank Tokenization:
['There', 'are', "n't", 'that', 'many', 'tokenizers', '.']
```

White Space Tokenization

White Space Tokenization processes non-white spaces as tokens:

Copy Explain

```
# White Space Tokenizationtokenizer
from nltk.tokenize import WhitespaceTokenizer
tokenizer=WhitespaceTokenizer()
text = "Tokenize this sequence of words using white space. There aren't many words."
tokens = tokenizer.tokenize(text)
print("White Space Tokenization:")
print(tokens)
print()
WhitespaceTokenizer has separated the words using non-white spaces as tokens:
White Space Tokenization:
['Tokenize', 'this', 'sequence', 'of', 'words', 'using', 'white', 'space.', 'There',
"aren't", 'many', 'words.']
Note that, unlike the TreebankWordTokenizer, WhitespaceTokenizer did not process the
contraction  aren't because it doesn't contain a white space.
```

Punkt Tokenization

A Punkt tokenizer splits sequences into sentences after going through unsupervised pretraining. It will not require labels:

Copy Explain

```
# Punkt Sentence Tokenization
tokenizer = PunktSentenceTokenizer()
text = "A tokenizer can be trained. Many tokenizers aren't trained."
sentences = tokenizer.tokenize(text)
print("Punkt Sentence Tokenization:")
print(sentences)
print()
PunktSentenceTokenizer()produces individual sentences:
Punkt Sentence Tokenization:
['A tokenizer can be trained.', "Many tokenizers aren't trained."]
```

Word Punctuation Tokenization

Word Punctuation tokenization splits sequences into words based on white spaces and punctuation:

Copy Explain

```
# Word Punctuation Tokenization
tokenizer = WordPunctTokenizer()
text = "They won a prize! They were overjoyed."
tokens = tokenizer.tokenize(text)
print("Word Punctuation Tokenization:")
print(tokens)
print()
The output of WordPunctTokenizer produces words with separators:
Word Punctuation Tokenization:
['They', 'won', 'a', 'prize', '!', 'They', 'were', 'overjoyed', '.']
```

Multi-Word Tokenization

Multi-Word tokenization splits sequences into words but preserves multi-word expressions such as "cannot":

Copy Explain

```
# Multi-Word Expression Tokenization
tokenizer = MWETokenizer()
tokenizer.add_mwe(("can", "not"))
text = "I cannot go to the movies today"
tokens = tokenizer.tokenize(text.split())
print("Multi-Word Expression Tokenization:")
print(tokens)
print()
In this case, MWETokenizer preserved "cannot":
Multi-Word Expression Tokenization:
['I', 'cannot', 'go', 'to', 'the', 'movies', 'today']
```

Sentence and word tokenizers are not to be taken lightly only because recent large-scale language models prefer subword tokenizers. For example, in some cases, sentence and word tokenizers can be part of a preprocessing pipeline to create labeled datasets for transformer model training.However, in some cases, subword tokenizers are sufficient to process raw data.We will now explore subword tokenizers.

Subword tokenizers

Transformer models are large-scale, Large Language Models (LLM). The size of the models and the number of tasks they perform require highly efficient tokenizers. Subword tokenizers for LLMs are the best choice for many reasons, including:

Out-of-Vocabulary (OOV) words

A subword tokenizer can capture words that were not part of the training phase (OOV). The tokenizer will break OOV into small units that a transformer model can process.

Vocabulary optimization

Subword tokenizers break sequences down into smaller units than sentence and word tokenizers. The size of the vocabulary is optimized.

Morphological flexibility

Subword tokenizers break words down into smaller units. The units obtained open the door to generalizations with other small units, deepening the model's ability to understand language.

Noise resistance

Even if a word is misspelled or contains a typo, a subword tokenizer can still capture and process its meaning.

Multiple Languages

Word-level tokenizers are related to a language. Subword tokenizers aren't.

BPE and Wordpiece are commonly used for transformer models. Understanding the principles of these two subword tokenizers will help you understand how any subword tokenizer works. Though we are focusing on BPE and Wordpiece, they are not the only subword tokenizers. Depending on your project, other subword tokenizers might be more efficient. So, before going through BPE and Wordpiece, let's go through two main ones to see how text sequences are broken down into subwords.Open `Sub_word_tokenizers.ipynb`First, we install Hugging Face libraries and a Python wrapper for Sentencepiece :

Copy Explain

```
!pip install transformers -qq
!pip install sentencepiece -qq
```

Now, we will go through two subword libraries that are not BPE or Wordpiece.

Unigram Language Model Tokenization

Unigram Language Model Tokenization was developed by Google. It trains using subword units. It drops the infrequent units. Unigram Language Model Tokenization is non-deterministic. As such, it will not always produce the same tokens for the same input. On the contrary, BPE is deterministic and will produce the same output tokens for an input.In the following example, we run a small training session on a sample of sentences and then tokenize a sentence.We first import the necessary modules:

Copy Explain

```
from tokenizers import Tokenizer
from tokenizers.models import Unigram
from tokenizers.trainers import UnigramTrainer
from tokenizers.pre_tokenizers import Whitespace
```

Then, we define a sample corpus:

Copy Explain

```
# Define a sample corpus
corpus = [
    "Subword tokenizers break text sequences into subwords.",
    "This sentence is another part of the corpus.",
    "Tokenization is the process of breaking text down into smaller units.",
    "These smaller units can be words, subwords, or even individual characters.",
    "Transformer models often use subword tokenization."
]
```

We train the tokenizer using a whitespace pre-tokenizer:

Copy Explain

```
# Instantiate a Unigram tokenizer model
tokenizer = Tokenizer(Unigram([]))
# Add a pre-tokenizer
tokenizer.pre_tokenizer = Whitespace()
# Train the tokenizer model
trainer = UnigramTrainer(vocab_size=5000)  # Here you set the desired vocabulary size
tokenizer.train_from_iterator(corpus, trainer)
```

Finally, we tokenize the target sentence:

Copy Explain

```
# Now let's tokenize the original sentence
output = tokenizer.encode("Subword tokenizers break text sequences into subwords.")
print(output.tokens)
```

As you can see, the output is quite different from the word and sentence tokenizers. The output is not a set of words or sentences but subwords:

Copy Explain

```
['S', 'ubword', 'tokeniz', 'er', 's', 'break', 'te', 'x', 't', 'se', 'q', 'u', 'ence',
's', 'in', 'to', 'subword', 's', '.']
```

SentencePiece

A SentencePiece tokenizer adds a BPE approach to the Unigram Language Model Tokenizer. It doesn't require a pre-tokenizer and can process raw data.In the following example, we train a small sample with no pre-tokenizer:First, we import the necessary modules:

Copy Explain

```
import sentencepiece as spm
import random
```

Then, we create a small sample corpus:

Copy Explain

```
# Define a basic corpus
basic_corpus = [
    "Subword tokenizers break text sequences into subwords.",
    "This sentence is another part of the corpus.",
    "Tokenization is the process of breaking text down into smaller units.",
    "These smaller units can be words, subwords, or even individual characters.",
    "Transformer models often use subword tokenization."
]
```

We perform some data augmentation randomly:

Copy Explain

```
# Generate a larger corpus by repeating sentences from the basic corpus
corpus = [random.choice(basic_corpus) for _ in range(10000)]
```

We save the corpus:

Copy Explain

```
# Write the corpus to a text file
with open('large_corpus.txt', 'w') as f:
    for Sentence in corpus:
        f.write(sentence + '\n')
```

We train the tokenizer:

Copy Explain

```
# Train the SentencePiece model
spm.SentencePieceTrainer.train(input='large_corpus.txt', model_prefix='m',
vocab_size=88)
```

We load the trained model:

Copy Explain

```
# Load the trained model
sp = spm.SentencePieceProcessor()
sp.load('m.model')
```

We tokenize our target sequence:

Copy Explain

```
# Tokenize the original sentence
tokens = sp.encode_as_pieces("Subword tokenizers break text sequences into subwords.")
print(tokens)
```

We have gone through two main subword tokenizers, not BPE or Wordpiece. However, you can see that, unlike sentence and word tokenizers, we obtain subwords through different approaches. There are also other subword tokenizers, each with its specific approach.The choice of a tokenizer remains a decision for each type of project.In *Chapter 6, Pretraining a Transformer from Scratch through RoBERTa*, *section 3, Step 3: Training a tokenizer,* we implemented a BPE tokenizer. In this section, we will focus on Wordpiece tokenization.Before creating subword tokenization in code with WordPiece, let's sum up BPE training.

Byte Pair Encoding (BPE)

BPE begins with a vocabulary of individual characters. It then merges the most frequent pair of consecutive characters. A hyperparameter determines the number of times the process is repeated. The result is a set of merged characters that can be an individual character, subwords, or words. The result is stored in a merge file, in **merge.txt**, for example, as in *Chapter 6*. A second file is generated, **vocab.json**, for instance, as in Chapter 6, that contains the dictionary of subwords with their unique ID, an integer.The sequence is broken down into subwords found in the merge dictionary during the tokenization process. The process is iterative. The tokenizer will try to find larger subwords n times. When an optimum result is reached, the iteratively merged subwords, and the tokenizers assign an ID to the subword with the information subword IDs contained in the vocab.json. You might want to go through the code of Chapter 6 again. You can select a corpus of your choice, train the tokenizer, train the model and measure the impact of the tokenization process on the outputs of a transformer model.

WordPiece

WordPiece, as BPE, begins with a vocabulary of individual characters. This ensures that any word can be tokenized. Then, the training process builds subwords. It uses an optimization process to minimize the number of subwords. When the training process is completed, the tokenizer will break sequences down into the longest sequence words in its vocabulary. Subwords not at the beginning of an original word contain a prefix' ##.' For example, "undo" will be represented as ["un"," ##do"]. Keep this in mind because it will help us identify Wordpiece tokenizers. Tokenizers will have a strong impact on the training of a transformer model. Choosing the right tokenizer will often determine its outcome from the start.Now, let's explore the outputs of a Wordpiece tokenizer.

Exploring in code

To get started, go back to `Exploring tokenizers.ipynb` in the GitHub repository of this chapter. Then, scroll down to the *Subwork tokenizers* section.Let's begin by detecting whether the tokenizer is a Wordpiece tokenizer.

Detecting the type of tokenizer

In this example, our first step will be to detect whether our tokenizer is a Wordpiece tokenizer. To achieve this, we will load the merge.txt and vocab.json file generated in Chapter 6:

Copy Explain

```
#1.Load merges.txt using the Colab file manager
#2.Downloading the file from GitHub
!curl -L https://raw.githubusercontent.com/Denis2054 Transformers-for-NLP-and-Computer-Vision-3rd-Edition /main/Chapter09/merges.txt --output "merges.txt"
#1.Load vocab.json using the Colab file manager
#2.Downloading the file from GitHub
!curl -L https://raw.githubusercontent.com/Denis2054/ Transformers-for-NLP-and-Computer-Vision-3rd-Edition/main/Chapter09/vocab.txt --output "vocab.json"
```

Now, let's import a Hugging Face RobertaTokenizer:

Copy Explain

```
from transformers import RobertaTokenizer
tokenizer = RobertaTokenizer.from_pretrained("/content", max_length=512)
# Get the vocabulary
vocab = tokenizer.get_vocab()
# Check if WordPiece or BPE
is_wordpiece = any(token.startswith('##') for token in vocab)
# Print the tokenizer type
if is_wordpiece:
    print("Tokenizer type: WordPiece")
else:
    print("Tokenizer type: BPE")
```

The output shows it's a BPE tokenizer. This example is simplified. We suppose the tokenizer can only be a BPE or a Wordpiece tokenizer. Also, we are sure it's a BPE because we trained it in Chapter 6.

Copy Explain

```
Tokenizer type: BPE
```

Now, we'll load a tokenizer trained on Hugging Face's platform:

Copy Explain

```
from transformers import BertTokenizer
# Load the BERT tokenizer
model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_name)
# Get the vocabulary
vocab = tokenizer.get_vocab()
```

We rerun the detection process:

Copy Explain

```
# Check if WordPiece or another type of tokenization was used
is_wordpiece = any(token.startswith('##') for token in vocab)
# Print the tokenizer type
if is_wordpiece:
    print("Tokenizer type: WordPiece")
else:
    print("Tokenizer type: BPE")
```

This time, we can see it's a Wordpiece tokenizer because the "##" prefix was detected:

```
Tokenizer type: Wordpiece
```

To make sure it's a Wordpiece, let's print the vocabulary:

```
# Print the vocabulary
for token, id in vocab.items():
    print(f'{token}: {id}')
```

The output displays the "##" prefix that signals that the subword is not the beginning of a word, as shown in the following excerpt:

```
bien: 29316
eels: 29317
marek: 29318
##ayton: 29319
##cence: 29320
```

Now, let's display the data using the tabulate library.

Displaying token-ID mappings

This section aims to look into the token-ID mappings of the Wordpiece vocabulary. We import the tabulate modules and widgets to create an interactive interface, display modules and the tokenizer we just detected:

```
from tabulate import tabulate
import ipywidgets as widgets
from IPython.display import display
from transformers import BertTokenizer
```

We load the model and tokenizer:

```
# Load the BERT tokenizer
model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_name)
```

We retrieve the vocabulary:

Copy Explain

```
# Get the vocabulary
vocab = tokenizer.get_vocab()
```

We create a list with items in the vocabulary:

Copy Explain

```
# Convert the vocabulary to a list of tuples
vocab_list = list(vocab.items())
```

We sort the vocabulary and create a filter with a widget for our interactive interface:

Copy Explain

```
# Sort the vocabulary by token
sorted_vocab = sorted(vocab_list, key=lambda x: x[0])
# Create a text input widget for filtering
filter_widget = widgets.Text(placeholder='Filter vocabulary')
```

We create a function to filter and display the vocabulary:

Copy Explain

```
# Function to filter and display the vocabulary
def filter_vocabulary(filter_text):
    filtered_vocab = [word for word in sorted_vocab if word[0].startswith(filter_text)]
    table = tabulate(filtered_vocab, headers=['Token', 'ID'])
    display(widgets.HTML(table))
```

Finally, we call and display the filter:

Copy Explain

```
# Call the filter function when the widget value changes
filter_widget.observe(lambda event: filter_vocabulary(event.new), names='value')
# Display the filter widget
display(filter_widget)
```

A pretty filter interface is displayed:

Figure 10.xxx: the filter of our interface

Type a character, and the words beginning with that character will appear. "t," for example, will generate an output with all the token-IDs beginning with that letter, as shown in the following excerpt:

```
…tomorrow 4826 ton 10228 tone 430…
```

To run another filter, rerun the cell. Again, take the time to observe the token-IDs to see the characters, prefixes (' ##'), partial words, and words present in the vocabulary. It will give you a good mental representation of how tokenizers base their tokenization process.

Analyzing and controlling the quality of token-ID mappings

In the previous section, we went through the token-ID mappings of the vocabulary of a Wordpiece tokenizer.In this section, we will do the opposite: we will take a word and break it down into its tokenized token-ID mappings. We will go from words to tokens.We first load the transformer model:

```
# Load the BERT tokenizer
model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_name)
```

We define the function to tokenize a word and display the output:

```
# Function to tokenize a word and provide information
def tokenize_word(word):
    # Tokenize the word
    tokens = tokenizer.tokenize(word)
    # Check if the word was found directly or is the result of a subword process
    if len(tokens) == 1 and tokens[0] == word:
        process = "Direct"
    else:
        process = "Subword"
    # Display the word and process information
    print("Word:", word)
    print("Tokenized Tokens:", tokens)
    print("Tokenization Process:", process)
```

We now create a widget to enter a word:

Copy Explain

```
# Create a widget for entering the word
word_input = widgets.Text(description='Enter a Word:')
display(word_input)
```

We create an event handler to tokenize the word.

Copy Explain

```
# Create an event handler for the widget
def on_button_click(b):
    word = word_input.value
    tokenize_word(word)
```

Finally, we create a button to trigger the tokenization function which also displays the output:

Copy Explain

```
# Create a button widget for triggering the tokenization process
button = widgets.Button(description="Tokenize")
button.on_click(on_button_click)
display(button)
```

If we type a random sequence such as "dsdf" we can see that "ds" exists in the and that "##df" is not the beginning of a word. "dsdf" is a subword.

Copy Explain

```
Word: dsdf
Tokenized Tokens: ['ds', '##df']
Tokenization Process: Subword
```

If we type "word", we can see that the "word" is present in the vocabulary:

Copy Explain

```
Word: word
Tokenized Tokens: ['word']
Tokenization Process: Direct
```

Now, let's enter a more difficult word such as "amoeboid:"

Copy Explain

```
Word: amoeboid
Tokenized Tokens: ['am', '##oe', '##bo', '##id']
Tokenization Process: Subword
```

The word "amoeboid" is broken down into subwords with multiple prefixes.The `am` token in `amoeboid` brings polysemy (several meanings for the same sequence) into the problem at a low level. `Am` can be a prefix, the word `am` as in `I` + `am`, or a sub-word such as in `am` + `bush`. Attention layers could associate the `am` of one token with another `am`, creating relationships that do not exist. Polysemy on complex words is a challenging problem. We can see that even subword tokenizers can face problems with rare and complex words. These words might not be encountered often in training datasets leading them to be underestimated. However, such words might be critical in medical or legal papers.As a cornerstone of transformer models, tokenization will profoundly impact the models' accuracy and performance.

Summary

In this chapter, we measured the impact of tokenization on the subsequent layers of a transformer model. A transformer model can only attend to tokens from a stack's embedding and positional encoding sub-layers. It does not matter if the model is an encoder-decoder, encoder-only, or decoder-only model. Furthermore, whether the dataset seems good enough to train does not matter.If the tokenization process fails, even partly, our transformer model will miss critical tokens.We first saw that raw datasets might be enough for standard language tasks to train a transformer.However, we discovered that even if a pretrained tokenizer has gone through a billion words, it only creates a dictionary with a small portion of the vocabulary it comes across. Like us, a tokenizer captures the essence of the language it is learning and only *remembers* the most important words if these words are also frequently used. This approach works well for a standard task and creates problems with specific tasks and vocabulary.We looked for ideas, among many, to work around the limits of standard tokenizers. We applied a language-checking method to adapt the text we wish to process, such as how a tokenizer *thinks* and encodes data.We then explored sentence and word tokenizers to understand how text sequences can be broken down into sentences and words. We reviewed several sentence and word tokenizers, including sentence tokenization and regular expression tokenization.Sentence and word tokenizers are useful for many NLP tasks, including, in some cases, preprocessing datasets for transformer model training. However, on large-scale corpora, they will generate large dictionaries that will slow down the training process of a transformer.Therefore, we explored subword tokenizers such as Unigram Language Model Tokenization, SentencePiece, BPE, and Wordpiece. We focused on Wordpiece tokenizers to explore token-ID mappings in detail. The skill of tokenization requires balancing preserving information and optimizing computational performances. You must choose a method that fits your task and model.Once the tokenizing phase of a project has been completed, embedding is the next logical step.The next chapter, *Understanding and Applying Embeddings with OpenAI APIs*, will take us into embeddings and transfer learning.

Questions

1. A tokenized dictionary contains every word that exists in a language. (True/False)
2. Pretrained tokenizers can encode any dataset. (True/False)
3. It is good practice to check a database before using it. (True/False)
4. It is good practice to eliminate obscene data from datasets. (True/False)
5. It is good practice to delete data containing discriminating assertions. (True/False)
6. Raw datasets might sometimes produce relationships between noisy content and useful content. (True/False)
7. A standard pretrained tokenizer contains the English vocabulary of the past 700 years. (True/False)
8. Old English can create problems when encoding data with a tokenizer trained in modern English. (True/False)
9. Medical and other types of jargon can create problems when encoding data with a tokenizer trained in modern English. (True/False)
10. Controlling the output of the encoded data produced by a pretrained tokenizer is good practice. (True/False)

References

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu, 2019, Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer: https://arxiv.org/pdf/1910.10683.pdf

Gensim: https://radimrehurek.com/gensim/intro.html

Further Reading

Tokenization Tractability for Human and Machine Learning Model: An Annotation Study,Hiraoka et al.(2023), https://arxiv.org/abs/2304.10813

Join our book's Discord space

Previous Chapter

Next Chapter