Part III

Data Preparation

Chapter 5

How to Clean Text Manually and with NLTK

You cannot go straight from raw text to fitting a machine learning or deep learning model. You must clean your text first, which means splitting it into words and handling punctuation and case. In fact, there is a whole suite of text preparation methods that you may need to use, and the choice of methods really depends on your natural language processing task. In this tutorial, you will discover how you can clean and prepare your text ready for modeling with machine learning. After completing this tutorial, you will know:

- How to get started by developing your own very simple text cleaning tools.
- How to take a step up and use the more sophisticated methods in the NLTK library.
- Considerations when preparing text for natural language processing models.

Let's get started.

5.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Metamorphosis by Franz Kafka
2. Text Cleaning is Task Specific
3. Manual Tokenization
4. Tokenization and Cleaning with NLTK
5. Additional Text Cleaning Considerations

5.2 Metamorphosis by Franz Kafka

Let's start off by selecting a dataset. In this tutorial, we will use the text from the book Metamorphosis by Franz Kafka. No specific reason, other than it's short, I like it, and you may like it too. I expect it's one of those classics that most students have to read in school. The full text for Metamorphosis is available for free from Project Gutenberg. You can download the ASCII text version of the text here:

- Metamorphosis by Franz Kafka Plain Text UTF-8 (may need to load the page twice). http://www.gutenberg.org/cache/epub/5200/pg5200.txt

Download the file and place it in your current working directory with the file name `metamorphosis.txt`. The file contains header and footer information that we are not interested in, specifically copyright and license information. Open the file and delete the header and footer information and save the file as `metamorphosis_clean.txt`. The start of the clean file should look like:

> One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin.

The file should end with:

> And, as if in confirmation of their new dreams and good intentions, as soon as they reached their destination Grete was the first to get up and stretch out her young body.

Poor Gregor...

5.3 Text Cleaning Is Task Specific

After actually getting a hold of your text data, the first step in cleaning up text data is to have a strong idea about what you're trying to achieve, and in that context review your text to see what exactly might help. Take a moment to look at the text. What do you notice? Here's what I see:

- It's plain text so there is no markup to parse (yay!).
- The translation of the original German uses UK English (e.g. *travelling*).
- The lines are artificially wrapped with new lines at about 70 characters (meh).
- There are no obvious typos or spelling mistakes.
- There's punctuation like commas, apostrophes, quotes, question marks, and more.
- There's hyphenated descriptions like *armour-like*.
- There's a lot of use of the em dash (-) to continue sentences (maybe replace with commas?).
- There are names (e.g. *Mr. Samsa*)

- There does not appear to be numbers that require handling (e.g. 1999)
- There are section markers (e.g. *II* and *III*).

I'm sure there is a lot more going on to the trained eye. We are going to look at general text cleaning steps in this tutorial. Nevertheless, consider some possible objectives we may have when working with this text document. For example:

- If we were interested in developing a Kafkaesque language model, we may want to keep all of the case, quotes, and other punctuation in place.
- If we were interested in classifying documents as *Kafka* and *Not Kafka*, maybe we would want to strip case, punctuation, and even trim words back to their stem.

Use your task as the lens by which to choose how to ready your text data.

5.4 Manual Tokenization

Text cleaning is hard, but the text we have chosen to work with is pretty clean already. We could just write some Python code to clean it up manually, and this is a good exercise for those simple problems that you encounter. Tools like regular expressions and splitting strings can get you a long way.

5.4.1 Load Data

Let's load the text data so that we can work with it. The text is small and will load quickly and easily fit into memory. This will not always be the case and you may need to write code to memory map the file. Tools like NLTK (covered in the next section) will make working with large files much easier. We can load the entire `metamorphosis_clean.txt` into memory as follows:

```
# load text
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
```

Listing 5.1: Manually load the file.

5.4.2 Split by Whitespace

Clean text often means a list of words or tokens that we can work with in our machine learning models. This means converting the raw text into a list of words and saving it again. A very simple way to do this would be to split the document by white space, including " " (space), new lines, tabs and more. We can do this in Python with the `split()` function on the loaded string.

```
# load text
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
```

```
file.close()
# split into words by white space
words = text.split()
print(words[:100])
```

Listing 5.2: Manually split words by white space.

Running the example splits the document into a long list of words and prints the first 100 for us to review. We can see that punctuation is preserved (e.g. *wasn't* and *armour-like*), which is nice. We can also see that end of sentence punctuation is kept with the last word (e.g. *thought.*), which is not great.

```
['One', 'morning,', 'when', 'Gregor', 'Samsa', 'woke', 'from', 'troubled', 'dreams,', 'he',
    'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'horrible',
    'vermin.', 'He', 'lay', 'on', 'his', 'armour-like', 'back,', 'and', 'if', 'he',
    'lifted', 'his', 'head', 'a', 'little', 'he', 'could', 'see', 'his', 'brown', 'belly,',
    'slightly', 'domed', 'and', 'divided', 'by', 'arches', 'into', 'stiff', 'sections.',
    'The', 'bedding', 'was', 'hardly', 'able', 'to', 'cover', 'it', 'and', 'seemed',
    'ready', 'to', 'slide', 'off', 'any', 'moment.', 'His', 'many', 'legs,', 'pitifully',
    'thin', 'compared', 'with', 'the', 'size', 'of', 'the', 'rest', 'of', 'him,', 'waved',
    'about', 'helplessly', 'as', 'he', 'looked.', '"What\'s', 'happened', 'to', 'me?"',
    'he', 'thought.', 'It', "wasn't", 'a', 'dream.', 'His', 'room,', 'a', 'proper', 'human']
```

Listing 5.3: Example output of splitting words by white space.

5.4.3 Select Words

Another approach might be to use the regex model (`re`) and split the document into words by selecting for strings of alphanumeric characters (a-z, A-Z, 0-9 and '_'). For example:

```
import re
# load text
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split based on words only
words = re.split(r'\W+', text)
print(words[:100])
```

Listing 5.4: Manually select words with regex.

Again, running the example we can see that we get our list of words. This time, we can see that *armour-like* is now two words *armour* and *like* (fine) but contractions like *What's* is also two words *What* and *s* (not great).

```
['One', 'morning', 'when', 'Gregor', 'Samsa', 'woke', 'from', 'troubled', 'dreams', 'he',
    'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'horrible',
    'vermin', 'He', 'lay', 'on', 'his', 'armour', 'like', 'back', 'and', 'if', 'he',
    'lifted', 'his', 'head', 'a', 'little', 'he', 'could', 'see', 'his', 'brown', 'belly',
    'slightly', 'domed', 'and', 'divided', 'by', 'arches', 'into', 'stiff', 'sections',
    'The', 'bedding', 'was', 'hardly', 'able', 'to', 'cover', 'it', 'and', 'seemed',
    'ready', 'to', 'slide', 'off', 'any', 'moment', 'His', 'many', 'legs', 'pitifully',
    'thin', 'compared', 'with', 'the', 'size', 'of', 'the', 'rest', 'of', 'him', 'waved',
    'about', 'helplessly', 'as', 'he', 'looked', 'What', 's', 'happened', 'to', 'me', 'he',
    'thought', 'It', 'wasn', 't', 'a', 'dream', 'His', 'room']
```

Listing 5.5: Example output of selecting words with regex.

5.4.4 Split by Whitespace and Remove Punctuation

We may want the words, but without the punctuation like commas and quotes. We also want to keep contractions together. One way would be to split the document into words by white space (as in the section *Split by Whitespace*), then use string translation to replace all punctuation with nothing (e.g. remove it). Python provides a constant called string.punctuation that provides a great list of punctuation characters. For example:

```
print(string.punctuation)
```

Listing 5.6: Print the known punctuation characters.

Results in:

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

Listing 5.7: Example output of printing the known punctuation characters.

We can use regular expressions to select for the punctuation characters and use the `sub()` function to replace them with nothing. For example:

```
re_punc = re.compile('[%s]' % re.escape(string.punctuation))
# remove punctuation from each word
stripped = [re_punc.sub('', w) for w in words]
```

Listing 5.8: Example of constructing a translation table that will remove punctuation.

We can put all of this together, load the text file, split it into words by white space, then translate each word to remove the punctuation.

```
import string
import re
# load text
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words by white space
words = text.split()
# prepare regex for char filtering
re_punc = re.compile('[%s]' % re.escape(string.punctuation))
# remove punctuation from each word
stripped = [re_punc.sub('', w) for w in words]
print(stripped[:100])
```

Listing 5.9: Manually remove punctuation.

We can see that this has had the desired effect, mostly. Contractions like *What's* have become *Whats* but *armour-like* has become *armourlike*.

```
['One', 'morning', 'when', 'Gregor', 'Samsa', 'woke', 'from', 'troubled', 'dreams', 'he',
    'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'horrible',
    'vermin', 'He', 'lay', 'on', 'his', 'armourlike', 'back', 'and', 'if', 'he', 'lifted',
    'his', 'head', 'a', 'little', 'he', 'could', 'see', 'his', 'brown', 'belly',
    'slightly', 'domed', 'and', 'divided', 'by', 'arches', 'into', 'stiff', 'sections',
    'The', 'bedding', 'was', 'hardly', 'able', 'to', 'cover', 'it', 'and', 'seemed',
    'ready', 'to', 'slide', 'off', 'any', 'moment', 'His', 'many', 'legs', 'pitifully',
    'thin', 'compared', 'with', 'the', 'size', 'of', 'the', 'rest', 'of', 'him', 'waved',
    'about', 'helplessly', 'as', 'he', 'looked', 'Whats', 'happened', 'to', 'me', 'he',
    'thought', 'It', 'wasnt', 'a', 'dream', 'His', 'room', 'a', 'proper', 'human']
```

Listing 5.10: Example output of removing punctuation with translation tables.

Sometimes text data may contain non-printable characters. We can use a similar approach to filter out all non-printable characters by selecting the inverse of the `string.printable` constant. For example:

```
...
re_print = re.compile('[^%s]' % re.escape(string.printable))
result = [re_print.sub('', w) for w in words]
```

Listing 5.11: Example of removing non-printable characters.

5.4.5 Normalizing Case

It is common to convert all words to one case. This means that the vocabulary will shrink in size, but some distinctions are lost (e.g. *Apple* the company vs *apple* the fruit is a commonly used example). We can convert all words to lowercase by calling the `lower()` function on each word. For example:

```
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words by white space
words = text.split()
# convert to lower case
words = [word.lower() for word in words]
print(words[:100])
```

Listing 5.12: Manually normalize case.

Running the example, we can see that all words are now lowercase.

```
['one', 'morning,', 'when', 'gregor', 'samsa', 'woke', 'from', 'troubled', 'dreams,', 'he',
    'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'horrible',
    'vermin.', 'he', 'lay', 'on', 'his', 'armour-like', 'back,', 'and', 'if', 'he',
    'lifted', 'his', 'head', 'a', 'little', 'he', 'could', 'see', 'his', 'brown', 'belly,',
    'slightly', 'domed', 'and', 'divided', 'by', 'arches', 'into', 'stiff', 'sections.',
    'the', 'bedding', 'was', 'hardly', 'able', 'to', 'cover', 'it', 'and', 'seemed',
    'ready', 'to', 'slide', 'off', 'any', 'moment.', 'his', 'many', 'legs,', 'pitifully',
    'thin', 'compared', 'with', 'the', 'size', 'of', 'the', 'rest', 'of', 'him,', 'waved',
    'about', 'helplessly', 'as', 'he', 'looked.', '"what\'s', 'happened', 'to', 'me?"',
    'he', 'thought.', 'it', "wasn't", 'a', 'dream.', 'his', 'room,', 'a', 'proper', 'human']
```

Listing 5.13: Example output of removing punctuation.

5.4.6 Note on Cleaning Text

Cleaning text is really hard, problem specific, and full of tradeoffs. Remember, simple is better. Simpler text data, simpler models, smaller vocabularies. You can always make things more complex later to see if it results in better model skill. Next, we'll look at some of the tools in the NLTK library that offer more than simple string splitting.

5.5 Tokenization and Cleaning with NLTK

The Natural Language Toolkit, or NLTK for short, is a Python library written for working and modeling text. It provides good tools for loading and cleaning text that we can use to get our data ready for working with machine learning and deep learning algorithms.

5.5.1 Install NLTK

You can install NLTK using your favorite package manager, such as pip. On a POSIX-compaitable machine, this would be:

```
sudo pip install -U nltk
```

Listing 5.14: Command to install the NLTK library.

After installation, you will need to install the data used with the library, including a great set of documents that you can use later for testing other tools in NLTK. There are few ways to do this, such as from within a script:

```
import nltk
nltk.download()
```

Listing 5.15: NLTK script to download required text data.

Or from the command line:

```
python -m nltk.downloader all
```

Listing 5.16: Command to download NLTK required text data.

5.5.2 Split into Sentences

A good useful first step is to split the text into sentences. Some modeling tasks prefer input to be in the form of paragraphs or sentences, such as Word2Vec. You could first split your text into sentences, split each sentence into words, then save each sentence to file, one per line. NLTK provides the `sent_tokenize()` function to split text into sentences. The example below loads the `metamorphosis_clean.txt` file into memory, splits it into sentences, and prints the first sentence.

```
from nltk import sent_tokenize
# load data
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
```

```
# split into sentences
sentences = sent_tokenize(text)
print(sentences[0])
```

Listing 5.17: NLTK script to split text into sentences.

Running the example, we can see that although the document is split into sentences, that each sentence still preserves the new line from the artificial wrap of the lines in the original document.

```
One morning, when Gregor Samsa woke from troubled dreams, he found
himself transformed in his bed into a horrible vermin.
```

Listing 5.18: Example output of splitting text into sentences.

5.5.3 Split into Words

NLTK provides a function called `word_tokenize()` for splitting strings into tokens (nominally words). It splits tokens based on white space and punctuation. For example, commas and periods are taken as separate tokens. Contractions are split apart (e.g. *What's* becomes *What* and *'s*). Quotes are kept, and so on. For example:

```
from nltk.tokenize import word_tokenize
# load data
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words
tokens = word_tokenize(text)
print(tokens[:100])
```

Listing 5.19: NLTK script to split text into words.

Running the code, we can see that punctuation are now tokens that we could then decide to specifically filter out.

```
['One', 'morning', ',', 'when', 'Gregor', 'Samsa', 'woke', 'from', 'troubled', 'dreams',
    ',', 'he', 'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a',
    'horrible', 'vermin', '.', 'He', 'lay', 'on', 'his', 'armour-like', 'back', ',', 'and',
    'if', 'he', 'lifted', 'his', 'head', 'a', 'little', 'he', 'could', 'see', 'his',
    'brown', 'belly', ',', 'slightly', 'domed', 'and', 'divided', 'by', 'arches', 'into',
    'stiff', 'sections', '.', 'The', 'bedding', 'was', 'hardly', 'able', 'to', 'cover',
    'it', 'and', 'seemed', 'ready', 'to', 'slide', 'off', 'any', 'moment', '.', 'His',
    'many', 'legs', ',', 'pitifully', 'thin', 'compared', 'with', 'the', 'size', 'of',
    'the', 'rest', 'of', 'him', ',', 'waved', 'about', 'helplessly', 'as', 'he', 'looked',
    '.', '``', 'What', "'s", 'happened', 'to']
```

Listing 5.20: Example output of splitting text into words.

5.5.4 Filter Out Punctuation

We can filter out all tokens that we are not interested in, such as all standalone punctuation. This can be done by iterating over all tokens and only keeping those tokens that are all alphabetic. Python has the function `isalpha()` that can be used. For example:

```
from nltk.tokenize import word_tokenize
# load data
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words
tokens = word_tokenize(text)
# remove all tokens that are not alphabetic
words = [word for word in tokens if word.isalpha()]
print(words[:100])
```

Listing 5.21: NLTK script to remove punctuation.

Running the example, you can see that not only punctuation tokens, but examples like *armour-like* and *'s* were also filtered out.

```
['One', 'morning', 'when', 'Gregor', 'Samsa', 'woke', 'from', 'troubled', 'dreams', 'he',
    'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'horrible',
    'vermin', 'He', 'lay', 'on', 'his', 'back', 'and', 'if', 'he', 'lifted', 'his', 'head',
    'a', 'little', 'he', 'could', 'see', 'his', 'brown', 'belly', 'slightly', 'domed',
    'and', 'divided', 'by', 'arches', 'into', 'stiff', 'sections', 'The', 'bedding', 'was',
    'hardly', 'able', 'to', 'cover', 'it', 'and', 'seemed', 'ready', 'to', 'slide', 'off',
    'any', 'moment', 'His', 'many', 'legs', 'pitifully', 'thin', 'compared', 'with', 'the',
    'size', 'of', 'the', 'rest', 'of', 'him', 'waved', 'about', 'helplessly', 'as', 'he',
    'looked', 'What', 'happened', 'to', 'me', 'he', 'thought', 'It', 'was', 'a', 'dream',
    'His', 'room', 'a', 'proper', 'human', 'room']
```

Listing 5.22: Example output of removing punctuation.

5.5.5 Filter out Stop Words (and Pipeline)

Stop words are those words that do not contribute to the deeper meaning of the phrase. They are the most common words such as: *the*, *a*, and *is*. For some applications like documentation classification, it may make sense to remove stop words. NLTK provides a list of commonly agreed upon stop words for a variety of languages, such as English. They can be loaded as follows:

```
from nltk.corpus import stopwords
stop_words = stopwords.words('english')
print(stop_words)
```

Listing 5.23: NLTK script print stop words.

You can see the full list as follows:

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', 'yours',
    'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', 'her', 'hers',
    'herself', 'it', 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves',
    'what', 'which', 'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is', 'are',
    'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does',
    'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until',
    'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into',
    'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down',
    'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here',
```

```
    'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more',
    'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so',
    'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', 'should', 'now', 'd',
    'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', 'couldn', 'didn', 'doesn', 'hadn',
    'hasn', 'haven', 'isn', 'ma', 'mightn', 'mustn', 'needn', 'shan', 'shouldn', 'wasn',
    'weren', 'won', 'wouldn']
```

Listing 5.24: Example output of printing stop words.

You can see that they are all lower case and have punctuation removed. You could compare your tokens to the stop words and filter them out, but you must ensure that your text is prepared the same way. Let's demonstrate this with a small pipeline of text preparation including:

- Load the raw text.
- Split into tokens.
- Convert to lowercase.
- Remove punctuation from each token.
- Filter out remaining tokens that are not alphabetic.
- Filter out tokens that are stop words.

```
import string
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
# load data
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words
tokens = word_tokenize(text)
# convert to lower case
tokens = [w.lower() for w in tokens]
# prepare regex for char filtering
re_punc = re.compile('[%s]' % re.escape(string.punctuation))
# remove punctuation from each word
stripped = [re_punc.sub('', w) for w in tokens]
# remove remaining tokens that are not alphabetic
words = [word for word in stripped if word.isalpha()]
# filter out stop words
stop_words = set(stopwords.words('english'))
words = [w for w in words if not w in stop_words]
print(words[:100])
```

Listing 5.25: NLTK script filter out stop words.

Running this example, we can see that in addition to all of the other transforms, stop words like *a* and *to* have been removed. I note that we are still left with tokens like *nt*. The rabbit hole is deep; there's always more we can do.

```
['one', 'morning', 'gregor', 'samsa', 'woke', 'troubled', 'dreams', 'found', 'transformed',
    'bed', 'horrible', 'vermin', 'lay', 'armourlike', 'back', 'lifted', 'head', 'little',
    'could', 'see', 'brown', 'belly', 'slightly', 'domed', 'divided', 'arches', 'stiff',
    'sections', 'bedding', 'hardly', 'able', 'cover', 'seemed', 'ready', 'slide', 'moment',
    'many', 'legs', 'pitifully', 'thin', 'compared', 'size', 'rest', 'waved', 'helplessly',
    'looked', 'happened', 'thought', 'nt', 'dream', 'room', 'proper', 'human', 'room',
    'although', 'little', 'small', 'lay', 'peacefully', 'four', 'familiar', 'walls',
    'collection', 'textile', 'samples', 'lay', 'spread', 'table', 'samsa', 'travelling',
    'salesman', 'hung', 'picture', 'recently', 'cut', 'illustrated', 'magazine', 'housed',
    'nice', 'gilded', 'frame', 'showed', 'lady', 'fitted', 'fur', 'hat', 'fur', 'boa',
    'sat', 'upright', 'raising', 'heavy', 'fur', 'muff', 'covered', 'whole', 'lower',
    'arm', 'towards', 'viewer']
```

Listing 5.26: Example output of filtering out stop words.

5.5.6 Stem Words

Stemming refers to the process of reducing each word to its root or base. For example *fishing*, *fished*, *fisher* all reduce to the stem *fish*. Some applications, like document classification, may benefit from stemming in order to both reduce the vocabulary and to focus on the sense or sentiment of a document rather than deeper meaning. There are many stemming algorithms, although a popular and long-standing method is the Porter Stemming algorithm. This method is available in NLTK via the PorterStemmer class. For example:

```
from nltk.tokenize import word_tokenize
from nltk.stem.porter import PorterStemmer
# load data
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words
tokens = word_tokenize(text)
# stemming of words
porter = PorterStemmer()
stemmed = [porter.stem(word) for word in tokens]
print(stemmed[:100])
```

Listing 5.27: NLTK script stem words.

Running the example, you can see that words have been reduced to their stems, such as *trouble* has become *troubl*. You can also see that the stemming implementation has also reduced the tokens to lowercase, likely for internal look-ups in word tables.

```
['one', 'morn', ',', 'when', 'gregor', 'samsa', 'woke', 'from', 'troubl', 'dream', ',',
    'he', 'found', 'himself', 'transform', 'in', 'hi', 'bed', 'into', 'a', 'horribl',
    'vermin', '.', 'He', 'lay', 'on', 'hi', 'armour-lik', 'back', ',', 'and', 'if', 'he',
    'lift', 'hi', 'head', 'a', 'littl', 'he', 'could', 'see', 'hi', 'brown', 'belli', ',',
    'slightli', 'dome', 'and', 'divid', 'by', 'arch', 'into', 'stiff', 'section', '.',
    'the', 'bed', 'wa', 'hardli', 'abl', 'to', 'cover', 'it', 'and', 'seem', 'readi', 'to',
    'slide', 'off', 'ani', 'moment', '.', 'hi', 'mani', 'leg', ',', 'piti', 'thin',
    'compar', 'with', 'the', 'size', 'of', 'the', 'rest', 'of', 'him', ',', 'wave',
    'about', 'helplessli', 'as', 'he', 'look', '.', '``', 'what', "'s", 'happen', 'to'
```

Listing 5.28: Example output of stemming words.

There is a nice suite of stemming and lemmatization algorithms to choose from in NLTK, if reducing words to their root is something you need for your project.

5.6 Additional Text Cleaning Considerations

We are only getting started. Because the source text for this tutorial was reasonably clean to begin with, we skipped many concerns of text cleaning that you may need to deal with in your own project. Here is a shortlist of additional considerations when cleaning text:

- Handling large documents and large collections of text documents that do not fit into memory.
- Extracting text from markup like HTML, PDF, or other structured document formats.
- Transliteration of characters from other languages into English.
- Decoding Unicode characters into a normalized form, such as UTF8.
- Handling of domain specific words, phrases, and acronyms.
- Handling or removing numbers, such as dates and amounts.
- Locating and correcting common typos and misspellings.
- And much more...

The list could go on. Hopefully, you can see that getting truly clean text is impossible, that we are really doing the best we can based on the time, resources, and knowledge we have. The idea of *clean* is really defined by the specific task or concern of your project.

A pro tip is to continually review your tokens after every transform. I have tried to show that in this tutorial and I hope you take that to heart. Ideally, you would save a new file after each transform so that you can spend time with all of the data in the new form. Things always jump out at you when to take the time to review your data.

5.7 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Metamorphosis by Franz Kafka on Project Gutenberg.
  http://www.gutenberg.org/ebooks/5200
- Installing NLTK.
  http://www.nltk.org/install.html
- Installing NLTK Data.
  http://www.nltk.org/data.html

- Python `isalpha()` function.
  https://docs.python.org/3/library/stdtypes.html#str.isalpha
- Stop Words on Wikipedia.
  https://en.wikipedia.org/wiki/Stop_words
- Stemming on Wikipedia.
  https://en.wikipedia.org/wiki/Stemming
- `nltk.tokenize` package API.
  http://www.nltk.org/api/nltk.tokenize.html
- Porter Stemming algorithm.
  https://tartarus.org/martin/PorterStemmer/
- `nltk.stem` package API.
  http://www.nltk.org/api/nltk.stem.html
- *Processing Raw Text*, *Natural Language Processing with Python*.
  http://www.nltk.org/book/ch03.html

5.8 Summary

In this tutorial, you discovered how to clean text or machine learning in Python.

Specifically, you learned:

- How to get started by developing your own very simple text cleaning tools.
- How to take a step up and use the more sophisticated methods in the NLTK library.
- Considerations when preparing text for natural language processing models.

5.8.1 Next

In the next chapter, you will discover how you can encode text data using the scikit-learn Python library.

Chapter 6

How to Prepare Text Data with scikit-learn

Text data requires special preparation before you can start using it for predictive modeling. The text must be parsed to remove words, called tokenization. Then the words need to be encoded as integers or floating point values for use as input to a machine learning algorithm, called feature extraction (or vectorization). The scikit-learn library offers easy-to-use tools to perform both tokenization and feature extraction of your text data. In this tutorial, you will discover exactly how you can prepare your text data for predictive modeling in Python with scikit-learn. After completing this tutorial, you will know:

- How to convert text to word count vectors with `CountVectorizer`.
- How to convert text to word frequency vectors with `TfidfVectorizer`.
- How to convert text to unique integers with `HashingVectorizer`.

Let's get started.

6.1 The Bag-of-Words Model

We cannot work with text directly when using machine learning algorithms. Instead, we need to convert the text to numbers. We may want to perform classification of documents, so each document is an *input* and a class label is the *output* for our predictive algorithm. Algorithms take vectors of numbers as input, therefore we need to convert documents to fixed-length vectors of numbers.

A simple and effective model for thinking about text documents in machine learning is called the Bag-of-Words Model, or BoW. Note, that we cover the BoW model in great detail in the next part, starting with Chapter 8. The model is simple in that it throws away all of the order information in the words and focuses on the occurrence of words in a document. This can be done by assigning each word a unique number. Then any document we see can be encoded as a fixed-length vector with the length of the vocabulary of known words. The value in each position in the vector could be filled with a count or frequency of each word in the encoded document.

This is the bag-of-words model, where we are only concerned with encoding schemes that represent what words are present or the degree to which they are present in encoded documents without any information about order. There are many ways to extend this simple method, both by better clarifying what a *word* is and in defining what to encode about each word in the vector. The scikit-learn library provides 3 different schemes that we can use, and we will briefly look at each.

6.2 Word Counts with CountVectorizer

The `CountVectorizer` provides a simple way to both tokenize a collection of text documents and build a vocabulary of known words, but also to encode new documents using that vocabulary. You can use it as follows:

- Create an instance of the `CountVectorizer` class.
- Call the `fit()` function in order to learn a vocabulary from one or more documents.
- Call the `transform()` function on one or more documents as needed to encode each as a vector.

An encoded vector is returned with a length of the entire vocabulary and an integer count for the number of times each word appeared in the document. Because these vectors will contain a lot of zeros, we call them sparse. Python provides an efficient way of handling sparse vectors in the `scipy.sparse` package. The vectors returned from a call to `transform()` will be sparse vectors, and you can transform them back to NumPy arrays to look and better understand what is going on by calling the `toarray()` function. Below is an example of using the `CountVectorizer` to tokenize, build a vocabulary, and then encode a document.

```
from sklearn.feature_extraction.text import CountVectorizer
# list of text documents
text = ["The quick brown fox jumped over the lazy dog."]
# create the transform
vectorizer = CountVectorizer()
# tokenize and build vocab
vectorizer.fit(text)
# summarize
print(vectorizer.vocabulary_)
# encode document
vector = vectorizer.transform(text)
# summarize encoded vector
print(vector.shape)
print(type(vector))
print(vector.toarray())
```

Listing 6.1: Example of training a `CountVectorizer`.

Above, you can see that we access the vocabulary to see what exactly was tokenized by calling:

```
print(vectorizer.vocabulary_)
```

Listing 6.2: Print the learned vocabulary.

We can see that all words were made lowercase by default and that the punctuation was ignored. These and other aspects of tokenizing can be configured and I encourage you to review all of the options in the API documentation. Running the example first prints the vocabulary, then the shape of the encoded document. We can see that there are 8 words in the vocab, and therefore encoded vectors have a length of 8. We can then see that the encoded vector is a sparse matrix. Finally, we can see an array version of the encoded vector showing a count of 1 occurrence for each word except the (index and id 7) that has an occurrence of 2.

```
{'dog': 1, 'fox': 2, 'over': 5, 'brown': 0, 'quick': 6, 'the': 7, 'lazy': 4, 'jumped': 3}
(1, 8)
<class 'scipy.sparse.csr.csr_matrix'>
[[1 1 1 1 1 1 1 2]]
```

Listing 6.3: Example output of training a `CountVectorizer`.

Importantly, the same vectorizer can be used on documents that contain words not included in the vocabulary. These words are ignored and no count is given in the resulting vector. For example, below is an example of using the vectorizer above to encode a document with one word in the vocab and one word that is not.

```
# encode another document
text2 = ["the puppy"]
vector = vectorizer.transform(text2)
print(vector.toarray())
```

Listing 6.4: Example of encoding another document with the fit `CountVectorizer`.

Running this example prints the array version of the encoded sparse vector showing one occurrence of the one word in the vocab and the other word not in the vocab completely ignored.

```
[[0 0 0 0 0 0 0 1]]
```

Listing 6.5: Example output of encoding another document.

The encoded vectors can then be used directly with a machine learning algorithm.

6.3 Word Frequencies with `TfidfVectorizer`

Word counts are a good starting point, but are very basic. One issue with simple counts is that some words like *the* will appear many times and their large counts will not be very meaningful in the encoded vectors. An alternative is to calculate word frequencies, and by far the most popular method is called TF-IDF. This is an acronym that stands for *Term Frequency - Inverse Document* Frequency which are the components of the resulting scores assigned to each word.

- **Term Frequency**: This summarizes how often a given word appears within a document.
- **Inverse Document Frequency**: This downscales words that appear a lot across documents.

Without going into the math, TF-IDF are word frequency scores that try to highlight words that are more interesting, e.g. frequent in a document but not across documents. The `TfidfVectorizer` will tokenize documents, learn the vocabulary and inverse document

frequency weightings, and allow you to encode new documents. Alternately, if you already have a learned CountVectorizer, you can use it with a TfidfTransformer to just calculate the inverse document frequencies and start encoding documents. The same create, fit, and transform process is used as with the CountVectorizer. Below is an example of using the TfidfVectorizer to learn vocabulary and inverse document frequencies across 3 small documents and then encode one of those documents.

```
from sklearn.feature_extraction.text import TfidfVectorizer
# list of text documents
text = ["The quick brown fox jumped over the lazy dog.",
    "The dog.",
    "The fox"]
# create the transform
vectorizer = TfidfVectorizer()
# tokenize and build vocab
vectorizer.fit(text)
# summarize
print(vectorizer.vocabulary_)
print(vectorizer.idf_)
# encode document
vector = vectorizer.transform([text[0]])
# summarize encoded vector
print(vector.shape)
print(vector.toarray())
```

Listing 6.6: Example of training a TfidfVectorizer.

A vocabulary of 8 words is learned from the documents and each word is assigned a unique integer index in the output vector. The inverse document frequencies are calculated for each word in the vocabulary, assigning the lowest score of 1.0 to the most frequently observed word: *the* at index 7. Finally, the first document is encoded as an 8-element sparse array and we can review the final scorings of each word with different values for *the*, *fox*, and *dog* from the other words in the vocabulary.

```
{'fox': 2, 'lazy': 4, 'dog': 1, 'quick': 6, 'the': 7, 'over': 5, 'brown': 0, 'jumped': 3}
[ 1.69314718 1.28768207 1.28768207 1.69314718 1.69314718 1.69314718
1.69314718 1. ]
(1, 8)
[[ 0.36388646 0.27674503 0.27674503 0.36388646 0.36388646 0.36388646
0.36388646 0.42983441]]
```

Listing 6.7: Example output of training a TfidfVectorizer.

The scores are normalized to values between 0 and 1 and the encoded document vectors can then be used directly with most machine learning algorithms.

6.4 Hashing with HashingVectorizer

Counts and frequencies can be very useful, but one limitation of these methods is that the vocabulary can become very large. This, in turn, will require large vectors for encoding documents and impose large requirements on memory and slow down algorithms. A clever work around is to use a one way hash of words to convert them to integers. The clever part is that no vocabulary is required and you can choose an arbitrary-long fixed length vector. A downside

is that the hash is a one-way function so there is no way to convert the encoding back to a word (which may not matter for many supervised learning tasks).

The `HashingVectorizer` class implements this approach that can be used to consistently hash words, then tokenize and encode documents as needed. The example below demonstrates the `HashingVectorizer` for encoding a single document. An arbitrary fixed-length vector size of 20 was chosen. This corresponds to the range of the hash function, where small values (like 20) may result in hash collisions. Remembering back to Computer Science classes, I believe there are heuristics that you can use to pick the hash length and probability of collision based on estimated vocabulary size (e.g. a load factor of 75%). See any good textbook on the topic. Note that this vectorizer does not require a call to fit on the training data documents. Instead, after instantiation, it can be used directly to start encoding documents.

```
from sklearn.feature_extraction.text import HashingVectorizer
# list of text documents
text = ["The quick brown fox jumped over the lazy dog."]
# create the transform
vectorizer = HashingVectorizer(n_features=20)
# encode document
vector = vectorizer.transform(text)
# summarize encoded vector
print(vector.shape)
print(vector.toarray())
```

Listing 6.8: Example of training a `HashingVectorizer`.

Running the example encodes the sample document as a 20-element sparse array. The values of the encoded document correspond to normalized word counts by default in the range of -1 to 1, but could be made simple integer counts by changing the default configuration.

```
(1, 20)
[[ 0.         0.         0.         0.         0.         0.33333333
   0.        -0.33333333 0.33333333 0.         0.         0.33333333
   0.         0.         0.        -0.33333333 0.         0.
  -0.66666667 0.        ]]
```

Listing 6.9: Example output of training a `HashingVectorizer`.

6.5 Further Reading

This section provides more resources on the topic if you are looking go deeper.

6.5.1 Natural Language Processing

- Bag-of-words model on Wikipedia.
  https://en.wikipedia.org/wiki/Bag-of-words_model
- Tokenization on Wikipedia.
  https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization
- TF-IDF on Wikipedia.
  https://en.wikipedia.org/wiki/Tf%E2%80%93idf

6.5.2 sciki-learn

- Section 4.2. Feature extraction, scikit-learn User Guide.
  http://scikit-learn.org/stable/modules/feature_extraction.html
- sckit-learn Feature Extraction API.
  http://scikit-learn.org/stable/modules/classes.html#module-sklearn.feature_extraction
- Working With Text Data, scikit-learn Tutorial.
  http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html

6.5.3 Class APIs

- `CountVectorizer` scikit-learn API.
  http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
- `TfidfVectorizer` scikit-learn API.
  http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
- `TfidfTransformer` scikit-learn API.
  http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html
- `HashingVectorizer` scikit-learn API.
  http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html

6.6 Summary

In this tutorial, you discovered how to prepare text documents for machine learning with scikit-learn for bag-of-words models. Specifically, you learned:

- How to convert text to word count vectors with `CountVectorizer`.
- How to convert text to word frequency vectors with `TfidfVectorizer`.
- How to convert text to unique integers with `HashingVectorizer`.

We have only scratched the surface in these examples and I want to highlight that there are many configuration details for these classes to influence the tokenizing of documents that are worth exploring.

6.6.1 Next

In the next chapter, you will discover how you can prepare text data using the Keras deep learning library.

Chapter 7

How to Prepare Text Data With Keras

You cannot feed raw text directly into deep learning models. Text data must be encoded as numbers to be used as input or output for machine learning and deep learning models, such as word embeddings. The Keras deep learning library provides some basic tools to help you prepare your text data. In this tutorial, you will discover how you can use Keras to prepare your text data. After completing this tutorial, you will know:

- About the convenience methods that you can use to quickly prepare text data.
- The `Tokenizer` API that can be fit on training data and used to encode training, validation, and test documents.
- The range of 4 different document encoding schemes offered by the `Tokenizer` API.

Let's get started.

7.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Split words with `text_to_word_sequence`.
2. Encoding with `one_hot`.
3. Hash Encoding with `hashing_trick`.
4. `Tokenizer` API

7.2 Split Words with `text_to_word_sequence`

A good first step when working with text is to split it into words. Words are called tokens and the process of splitting text into tokens is called tokenization. Keras provides the `text_to_word_sequence()` function that you can use to split text into a list of words. By default, this function automatically does 3 things:

- Splits words by space.

- Filters out punctuation.
- Converts text to lowercase (lower=True).

You can change any of these defaults by passing arguments to the function. Below is an example of using the text_to_word_sequence() function to split a document (in this case a simple string) into a list of words.

```
from keras.preprocessing.text import text_to_word_sequence
# define the document
text = 'The quick brown fox jumped over the lazy dog.'
# tokenize the document
result = text_to_word_sequence(text)
print(result)
```

Listing 7.1: Example splitting words with the Tokenizer.

Running the example creates an array containing all of the words in the document. The list of words is printed for review.

```
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
```

Listing 7.2: Example output for splitting words with the Tokenizer.

This is a good first step, but further pre-processing is required before you can work with the text.

7.3 Encoding with one_hot

It is popular to represent a document as a sequence of integer values, where each word in the document is represented as a unique integer. Keras provides the one_hot() function that you can use to tokenize and integer encode a text document in one step. The name suggests that it will create a one hot encoding of the document, which is not the case. Instead, the function is a wrapper for the hashing_trick() function described in the next section. The function returns an integer encoded version of the document. The use of a hash function means that there may be collisions and not all words will be assigned unique integer values. As with the text_to_word_sequence() function in the previous section, the one_hot() function will make the text lower case, filter out punctuation, and split words based on white space.

In addition to the text, the vocabulary size (total words) must be specified. This could be the total number of words in the document or more if you intend to encode additional documents that contains additional words. The size of the vocabulary defines the hashing space from which words are hashed. By default, the *hash* function is used, although as we will see in the next section, alternate hash functions can be specified when calling the hashing_trick() function directly.

We can use the text_to_word_sequence() function from the previous section to split the document into words and then use a set to represent only the unique words in the document. The size of this set can be used to estimate the size of the vocabulary for one document. For example:

```
from keras.preprocessing.text import text_to_word_sequence
# define the document
text = 'The quick brown fox jumped over the lazy dog.'
```

```
# estimate the size of the vocabulary
words = set(text_to_word_sequence(text))
vocab_size = len(words)
print(vocab_size)
```

Listing 7.3: Example of preparing a vocabulary.

We can put this together with the one_hot() function and encode the words in the document. The complete example is listed below. The vocabulary size is increased by one-third to minimize collisions when hashing words.

```
from keras.preprocessing.text import one_hot
from keras.preprocessing.text import text_to_word_sequence
# define the document
text = 'The quick brown fox jumped over the lazy dog.'
# estimate the size of the vocabulary
words = set(text_to_word_sequence(text))
vocab_size = len(words)
print(vocab_size)
# integer encode the document
result = one_hot(text, round(vocab_size*1.3))
print(result)
```

Listing 7.4: Example of one hot encoding.

Running the example first prints the size of the vocabulary as 8. The encoded document is then printed as an array of integer encoded words.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
8
[5, 9, 8, 7, 9, 1, 5, 3, 8]
```

Listing 7.5: Example output for one hot encoding with the Tokenizer.

7.4 Hash Encoding with hashing_trick

A limitation of integer and count base encodings is that they must maintain a vocabulary of words and their mapping to integers. An alternative to this approach is to use a one-way hash function to convert words to integers. This avoids the need to keep track of a vocabulary, which is faster and requires less memory.

Keras provides the hashing_trick() function that tokenizes and then integer encodes the document, just like the one_hot() function. It provides more flexibility, allowing you to specify the hash function as either hash (the default) or other hash functions such as the built in md5 function or your own function. Below is an example of integer encoding a document using the md5 hash function.

```
from keras.preprocessing.text import hashing_trick
from keras.preprocessing.text import text_to_word_sequence
# define the document
text = 'The quick brown fox jumped over the lazy dog.'
```

```
# estimate the size of the vocabulary
words = set(text_to_word_sequence(text))
vocab_size = len(words)
print(vocab_size)
# integer encode the document
result = hashing_trick(text, round(vocab_size*1.3), hash_function='md5')
print(result)
```

Listing 7.6: Example of hash encoding.

Running the example prints the size of the vocabulary and the integer encoded document. We can see that the use of a different hash function results in consistent, but different integers for words as the `one_hot()` function in the previous section.

```
8
[6, 4, 1, 2, 7, 5, 6, 2, 6]
```

Listing 7.7: Example output for hash encoding with the `Tokenizer`.

7.5 Tokenizer API

So far we have looked at one-off convenience methods for preparing text with Keras. Keras provides a more sophisticated API for preparing text that can be fit and reused to prepare multiple text documents. This may be the preferred approach for large projects. Keras provides the `Tokenizer` class for preparing text documents for deep learning. The `Tokenizer` must be constructed and then fit on either raw text documents or integer encoded text documents. For example:

```
from keras.preprocessing.text import Tokenizer
# define 5 documents
docs = ['Well done!',
    'Good work',
    'Great effort',
    'nice work',
    'Excellent!']
# create the tokenizer
t = Tokenizer()
# fit the tokenizer on the documents
t.fit_on_texts(docs)
```

Listing 7.8: Example of fitting a `Tokenizer`.

Once fit, the `Tokenizer` provides 4 attributes that you can use to query what has been learned about your documents:

- `word_counts`: A dictionary mapping of words and their occurrence counts when the `Tokenizer` was fit.
- `word_docs`: A dictionary mapping of words and the number of documents that reach appears in.
- `word_index`: A dictionary of words and their uniquely assigned integers.

- document_count: A dictionary mapping and the number of documents they appear in calculated during the fit.

For example:

```
# summarize what was learned
print(t.word_counts)
print(t.document_count)
print(t.word_index)
print(t.word_docs)
```

Listing 7.9: Summarize the output of the fit Tokenizer.

Once the Tokenizer has been fit on training data, it can be used to encode documents in the train or test datasets. The texts_to_matrix() function on the Tokenizer can be used to create one vector per document provided per input. The length of the vectors is the total size of the vocabulary. This function provides a suite of standard bag-of-words model text encoding schemes that can be provided via a mode argument to the function. The modes available include:

- binary: Whether or not each word is present in the document. This is the default.
- count: The count of each word in the document.
- tfidf: The Text Frequency-Inverse DocumentFrequency (TF-IDF) scoring for each word in the document.
- freq: The frequency of each word as a ratio of words within each document.

We can put all of this together with a worked example.

```
from keras.preprocessing.text import Tokenizer
# define 5 documents
docs = ['Well done!',
    'Good work',
    'Great effort',
    'nice work',
    'Excellent!']
# create the tokenizer
t = Tokenizer()
# fit the tokenizer on the documents
t.fit_on_texts(docs)
# summarize what was learned
print(t.word_counts)
print(t.document_count)
print(t.word_index)
print(t.word_docs)
# integer encode documents
encoded_docs = t.texts_to_matrix(docs, mode='count')
print(encoded_docs)
```

Listing 7.10: Example of fitting and encoding with the Tokenizer.

Running the example fits the `Tokenizer` with 5 small documents. The details of the fit `Tokenizer` are printed. Then the 5 documents are encoded using a word count. Each document is encoded as a 9-element vector with one position for each word and the chosen encoding scheme value for each word position. In this case, a simple word count mode is used.

```
OrderedDict([('well', 1), ('done', 1), ('good', 1), ('work', 2), ('great', 1), ('effort',
    1), ('nice', 1), ('excellent', 1)])
5
{'work': 1, 'effort': 6, 'done': 3, 'great': 5, 'good': 4, 'excellent': 8, 'well': 2,
    'nice': 7}
{'work': 2, 'effort': 1, 'done': 1, 'well': 1, 'good': 1, 'great': 1, 'excellent': 1,
    'nice': 1}
[[ 0.  0.  1.  1.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  1.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1.]]
```

Listing 7.11: Example output from fitting and encoding with the `Tokenizer`.

The `Tokenizer` will be the key way we will prepare text for word embeddings throughout this book.

7.6 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Text Preprocessing Keras API.
  https://keras.io/preprocessing/text/
- `text_to_word_sequence` Keras API.
  https://keras.io/preprocessing/text/#text_to_word_sequence
- `one_hot` Keras API.
  https://keras.io/preprocessing/text/#one_hot
- `hashing_trick` Keras API.
  https://keras.io/preprocessing/text/#hashing_trick
- `Tokenizer` Keras API.
  https://keras.io/preprocessing/text/#tokenizer

7.7 Summary

In this tutorial, you discovered how you can use the Keras API to prepare your text data for deep learning. Specifically, you learned:

- About the convenience methods that you can use to quickly prepare text data.
- The `Tokenizer` API that can be fit on training data and used to encode training, validation, and test documents.
- The range of 4 different document encoding schemes offered by the `Tokenizer` API.

7.7.1 Next

This is the last chapter in the data preparation part. In the next part, you will discover how to develop bag-of-words models.