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
- Porer 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 of words and their counts.
- `word_docs`: An integer count of the total number of documents that were used to fit the `Tokenizer`.
- `word_index`: A dictionary of words and their uniquely assigned integers.
- `document_count`: A dictionary of words and how many documents each appeared in.

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.

Part IV

Bag-of-Words

Chapter 8

The Bag-of-Words Model

The bag-of-words model is a way of representing text data when modeling text with machine learning algorithms. The bag-of-words model is simple to understand and implement and has seen great success in problems such as language modeling and document classification. In this tutorial, you will discover the bag-of-words model for feature extraction in natural language processing. After completing this tutorial, you will know:

- What the bag-of-words model is and why it is needed to represent text.
- How to develop a bag-of-words model for a collection of documents.
- How to use different techniques to prepare a vocabulary and score words.

Let's get started.

8.1 Tutorial Overview

This tutorial is divided into the following parts:

1. The Problem with Text
2. What is a Bag-of-Words?
3. Example of the Bag-of-Words Model
4. Managing Vocabulary
5. Scoring Words
6. Limitations of Bag-of-Words

8.2 The Problem with Text

A problem with modeling text is that it is messy, and techniques like machine learning algorithms prefer well defined fixed-length inputs and outputs. Machine learning algorithms cannot work with raw text directly; the text must be converted into numbers. Specifically, vectors of numbers.

> In language processing, the vectors x are derived from textual data, in order to reflect various linguistic properties of the text.
>
> — Page 65, *Neural Network Methods in Natural Language Processing*, 2017.

This is called feature extraction or feature encoding. A popular and simple method of feature extraction with text data is called the bag-of-words model of text.

8.3 What is a Bag-of-Words?

A bag-of-words model, or BoW for short, is a way of extracting features from text for use in modeling, such as with machine learning algorithms. The approach is very simple and flexible, and can be used in a myriad of ways for extracting features from documents. A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

- A vocabulary of known words.
- A measure of the presence of known words.

It is called a *bag-of-words* , because any information about the order or structure of words in the document is discarded. The model is only concerned with whether known words occur in the document, not where in the document.

> A very common feature extraction procedures for sentences and documents is the bag-of-words approach (BOW). In this approach, we look at the histogram of the words within the text, i.e. considering each word count as a feature.
>
> — Page 69, *Neural Network Methods in Natural Language Processing*, 2017.

The intuition is that documents are similar if they have similar content. Further, that from the content alone we can learn something about the meaning of the document. The bag-of-words can be as simple or complex as you like. The complexity comes both in deciding how to design the vocabulary of known words (or tokens) and how to score the presence of known words. We will take a closer look at both of these concerns.

8.4 Example of the Bag-of-Words Model

Let's make the bag-of-words model concrete with a worked example.

8.4.1 Step 1: Collect Data

Below is a snippet of the first few lines of text from the book *A Tale of Two Cities* by Charles Dickens, taken from Project Gutenberg.

```
It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness,
```

Listing 8.1: Sample of text from *A Tale of Two Cities* by Charles Dickens.

For this small example, let's treat each line as a separate *document* and the 4 lines as our entire corpus of documents.

8.4.2 Step 2: Design the Vocabulary

Now we can make a list of all of the words in our model vocabulary. The unique words here (ignoring case and punctuation) are:

```
it
was
the
best
of
times
worst
age
wisdom
foolishness
```

Listing 8.2: List of unique words.

That is a vocabulary of 10 words from a corpus containing 24 words.

8.4.3 Step 3: Create Document Vectors

The next step is to score the words in each document. The objective is to turn each document of free text into a vector that we can use as input or output for a machine learning model. Because we know the vocabulary has 10 words, we can use a fixed-length document representation of 10, with one position in the vector to score each word. The simplest scoring method is to mark the presence of words as a boolean value, 0 for absent, 1 for present. Using the arbitrary ordering of words listed above in our vocabulary, we can step through the first document (*It was the best of times*) and convert it into a binary vector. The scoring of the document would look as follows:

```
it = 1
was = 1
the = 1
best = 1
of = 1
times = 1
worst = 0
age = 0
wisdom = 0
foolishness = 0
```

Listing 8.3: List of unique words and their occurrence in a document.

As a binary vector, this would look as follows:

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

Listing 8.4: First line of text as a binary vector.

The other three documents would look as follows:

```
"it was the worst of times" = [1, 1, 1, 0, 1, 1, 1, 0, 0, 0]
"it was the age of wisdom" = [1, 1, 1, 0, 1, 0, 0, 1, 1, 0]
"it was the age of foolishness" = [1, 1, 1, 0, 1, 0, 0, 1, 0, 1]
```

Listing 8.5: Remaining three lines of text as binary vectors.

All ordering of the words is nominally discarded and we have a consistent way of extracting features from any document in our corpus, ready for use in modeling. New documents that overlap with the vocabulary of known words, but may contain words outside of the vocabulary, can still be encoded, where only the occurrence of known words are scored and unknown words are ignored. You can see how this might naturally scale to large vocabularies and larger documents.

8.5 Managing Vocabulary

As the vocabulary size increases, so does the vector representation of documents. In the previous example, the length of the document vector is equal to the number of known words. You can imagine that for a very large corpus, such as thousands of books, that the length of the vector might be thousands or millions of positions. Further, each document may contain very few of the known words in the vocabulary.

This results in a vector with lots of zero scores, called a sparse vector or sparse representation. Sparse vectors require more memory and computational resources when modeling and the vast number of positions or dimensions can make the modeling process very challenging for traditional algorithms. As such, there is pressure to decrease the size of the vocabulary when using a bag-of-words model.

There are simple text cleaning techniques that can be used as a first step, such as:

- Ignoring case.
- Ignoring punctuation.
- Ignoring frequent words that don't contain much information, called stop words, like *a*, *of*, etc.
- Fixing misspelled words.
- Reducing words to their stem (e.g. *play* from *playing*) using stemming algorithms.

A more sophisticated approach is to create a vocabulary of grouped words. This both changes the scope of the vocabulary and allows the bag-of-words to capture a little bit more meaning from the document. In this approach, each word or token is called a *gram*. Creating a vocabulary of two-word pairs is, in turn, called a bigram model. Again, only the bigrams that appear in the corpus are modeled, not all possible bigrams.

> An n-gram is an n-token sequence of words: a 2-gram (more commonly called a bigram) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram (more commonly called a trigram) is a three-word sequence of words like "please turn your", or "turn your homework".
>
> — Page 85, *Speech and Language Processing*, 2009.

For example, the bigrams in the first line of text in the previous section: *It was the best of times* are as follows:

```
it was
was the
the best
best of
of times
```

Listing 8.6: List of bi-grams for a document.

A vocabulary that tracks triplets of words is called a trigram model and the general approach is called the n-gram model, where `n` refers to the number of grouped words. Often a simple bigram approach is better than a 1-gram bag-of-words model for tasks like documentation classification.

> a bag-of-bigrams representation is much more powerful than bag-of-words, and in many cases proves very hard to beat.
>
> — Page 75, *Neural Network Methods in Natural Language Processing*, 2017.

8.6 Scoring Words

Once a vocabulary has been chosen, the occurrence of words in example documents needs to be scored. In the worked example, we have already seen one very simple approach to scoring: a binary scoring of the presence or absence of words. Some additional simple scoring methods include:

- **Counts**. Count the number of times each word appears in a document.
- **Frequencies**. Calculate the frequency that each word appears in a document out of all the words in the document.

8.6.1 Word Hashing

You may remember from computer science that a hash function is a bit of math that maps data to a fixed size set of numbers. For example, we use them in hash tables when programming where perhaps names are converted to numbers for fast lookup. We can use a hash representation of known words in our vocabulary. This addresses the problem of having a very large vocabulary for a large text corpus because we can choose the size of the hash space, which is in turn the size of the vector representation of the document.

Words are hashed deterministically to the same integer index in the target hash space. A binary score or count can then be used to score the word. This is called the *hash trick* or *feature hashing*. The challenge is to choose a hash space to accommodate the chosen vocabulary size to minimize the probability of collisions and trade-off sparsity.

8.6.2 TF-IDF

A problem with scoring word frequency is that highly frequent words start to dominate in the document (e.g. larger score), but may not contain as much *informational content* to the model as rarer but perhaps domain specific words. One approach is to rescale the frequency of words by how often they appear in all documents, so that the scores for frequent words like *the* that are also frequent across all documents are penalized. This approach to scoring is called Term Frequency - Inverse Document Frequency, or TF-IDF for short, where:

- **Term Frequency**: is a scoring of the frequency of the word in the current document.
- **Inverse Document Frequency**: is a scoring of how rare the word is across documents.

The scores are a weighting where not all words are equally as important or interesting. The scores have the effect of highlighting words that are distinct (contain useful information) in a given document.

> Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low.
>
> — Page 118, *An Introduction to Information Retrieval*, 2008.

8.7 Limitations of Bag-of-Words

The bag-of-words model is very simple to understand and implement and offers a lot of flexibility for customization on your specific text data. It has been used with great success on prediction problems like language modeling and documentation classification. Nevertheless, it suffers from some shortcomings, such as:

- **Vocabulary**: The vocabulary requires careful design, most specifically in order to manage the size, which impacts the sparsity of the document representations.
- **Sparsity**: Sparse representations are harder to model both for computational reasons (space and time complexity) and also for information reasons, where the challenge is for the models to harness so little information in such a large representational space.
- **Meaning**: Discarding word order ignores the context, and in turn meaning of words in the document (semantics). Context and meaning can offer a lot to the model, that if modeled could tell the difference between the same words differently arranged (*this is interesting* vs *is this interesting*), synonyms (*old bike* vs *used bike*), and much more.

8.8 Further Reading

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

8.8.1 Books

- *Neural Network Methods in Natural Language Processing*, 2017.
  http://amzn.to/2wycQKA
- *Speech and Language Processing*, 2009.
  http://amzn.to/2vaEb7T
- *An Introduction to Information Retrieval*, 2008.
  http://amzn.to/2vvnPHP
- *Foundations of Statistical Natural Language Processing*, 1999.
  http://amzn.to/2vvnPHP

8.8.2 Wikipedia

- Bag-of-words model.
  https://en.wikipedia.org/wiki/N-gram
- n-gram.
  https://en.wikipedia.org/wiki/N-gram
- Feature hashing.
  https://en.wikipedia.org/wiki/Feature_hashing
- tf-idf.
  https://en.wikipedia.org/wiki/Tf-idf

8.9 Summary

In this tutorial, you discovered the bag-of-words model for feature extraction with text data. Specifically, you learned:

- What the bag-of-words model is and why we need it.
- How to work through the application of a bag-of-words model to a collection of documents.
- What techniques can be used for preparing a vocabulary and scoring words.

8.9.1 Next

In the next chapter, you will can prepare movie review data for the bag-of-words model.

Chapter 9

How to Prepare Movie Review Data for Sentiment Analysis

Text data preparation is different for each problem. Preparation starts with simple steps, like loading data, but quickly gets difficult with cleaning tasks that are very specific to the data you are working with. You need help as to where to begin and what order to work through the steps from raw data to data ready for modeling. In this tutorial, you will discover how to prepare movie review text data for sentiment analysis, step-by-step. After completing this tutorial, you will know:

- How to load text data and clean it to remove punctuation and other non-words.
- How to develop a vocabulary, tailor it, and save it to file.
- How to prepare movie reviews using cleaning and a pre-defined vocabulary and save them to new files ready for modeling.

Let's get started.

9.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Movie Review Dataset
2. Load Text Data
3. Clean Text Data
4. Develop Vocabulary
5. Save Prepared Data

9.2 Movie Review Dataset

The Movie Review Data is a collection of movie reviews retrieved from the `imdb.com` website in the early 2000s by Bo Pang and Lillian Lee. The reviews were collected and made available as part of their research on natural language processing. The reviews were originally released in 2002, but an updated and cleaned up version was released in 2004, referred to as *v2.0*. The dataset is comprised of 1,000 positive and 1,000 negative movie reviews drawn from an archive of the rec.arts.movies.reviews newsgroup hosted at IMDB. The authors refer to this dataset as the *polarity dataset*.

> Our data contains 1000 positive and 1000 negative reviews all written before 2002, with a cap of 20 reviews per author (312 authors total) per category. We refer to this corpus as the polarity dataset.

— A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts, 2004.

The data has been cleaned up somewhat, for example:

- The dataset is comprised of only English reviews.
- All text has been converted to lowercase.
- There is white space around punctuation like periods, commas, and brackets.
- Text has been split into one sentence per line.

The data has been used for a few related natural language processing tasks. For classification, the performance of classical models (such as Support Vector Machines) on the data is in the range of high 70% to low 80% (e.g. 78%-to-82%). More sophisticated data preparation may see results as high as 86% with 10-fold cross-validation. This gives us a ballpark of low-to-mid 80s if we were looking to use this dataset in experiments on modern methods.

> ... depending on choice of downstream polarity classifier, we can achieve highly statistically significant improvement (from 82.8% to 86.4%)

— A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts, 2004.

You can download the dataset from here:

- Movie Review Polarity Dataset (`review_polarity.tar.gz`, 3MB). http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

After unzipping the file, you will have a directory called `txt_sentoken` with two sub-directories containing the text *neg* and *pos* for negative and positive reviews. Reviews are stored one per file with a naming convention from *cv000* to *cv999* for each of *neg* and *pos*. Next, let's look at loading the text data.

9.3 Load Text Data

In this section, we will look at loading individual text files, then processing the directories of files. We will assume that the review data is downloaded and available in the current working directory in the folder `txt_sentoken`. We can load an individual text file by opening it, reading in the ASCII text, and closing the file. This is standard file handling stuff. For example, we can load the first negative review file `cv000_29416.txt` as follows:

```
# load one file
filename = 'txt_sentoken/neg/cv000_29416.txt'
# open the file as read only
file = open(filename, 'r')
# read all text
text = file.read()
# close the file
file.close()
```

Listing 9.1: Example of loading a single movie review.

This loads the document as ASCII and preserves any white space, like new lines. We can turn this into a function called `load_doc()` that takes a filename of the document to load and returns the text.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text
```

Listing 9.2: Function to load a document into memory.

We have two directories each with 1,000 documents each. We can process each directory in turn by first getting a list of files in the directory using the `listdir()` function, then loading each file in turn. For example, we can load each document in the negative directory using the `load_doc()` function to do the actual loading.

```
from os import listdir

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# specify directory to load
directory = 'txt_sentoken/neg'
# walk through all files in the folder
for filename in listdir(directory):
```

```
  # skip files that do not have the right extension
  if not filename.endswith(".txt"):
    next
  # create the full path of the file to open
  path = directory + '/' + filename
  # load document
  doc = load_doc(path)
  print('Loaded %s' % filename)
```

Listing 9.3: Example of loading a all movie reviews.

Running this example prints the filename of each review after it is loaded.

```
...
Loaded cv995_23113.txt
Loaded cv996_12447.txt
Loaded cv997_5152.txt
Loaded cv998_15691.txt
Loaded cv999_14636.txt
```

Listing 9.4: Example output of loading all movie reviews.

We can turn the processing of the documents into a function as well and use it as a template later for developing a function to clean all documents in a folder. For example, below we define a `process_docs()` function to do the same thing.

```
from os import listdir

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# load all docs in a directory
def process_docs(directory):
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip files that do not have the right extension
    if not filename.endswith(".txt"):
      next
    # create the full path of the file to open
    path = directory + '/' + filename
    # load document
    doc = load_doc(path)
    print('Loaded %s' % filename)

# specify directory to load
directory = 'txt_sentoken/neg'
process_docs(directory)
```

Listing 9.5: Example of loading a all movie reviews with functions.

Now that we know how to load the movie review text data, let's look at cleaning it.

9.4 Clean Text Data

In this section, we will look at what data cleaning we might want to do to the movie review data. We will assume that we will be using a bag-of-words model or perhaps a word embedding that does not require too much preparation.

9.4.1 Split into Tokens

First, let's load one document and look at the raw tokens split by white space. We will use the `load_doc()` function developed in the previous section. We can use the `split()` function to split the loaded document into tokens separated by white space.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# load the document
filename = 'txt_sentoken/neg/cv000_29416.txt'
text = load_doc(filename)
# split into tokens by white space
tokens = text.split()
print(tokens)
```

Listing 9.6: Load a movie review and split by white space.

Running the example gives a nice long list of raw tokens from the document.

```
...
'years', 'ago', 'and', 'has', 'been', 'sitting', 'on', 'the', 'shelves', 'ever', 'since',
    '.', 'whatever', '.', '.', '.', 'skip', 'it', '!', "where's", 'joblo', 'coming',
    'from', '?', 'a', 'nightmare', 'of', 'elm', 'street', '3', '(', '7/10', ')', '-',
    'blair', 'witch', '2', '(', '7/10', ')', '-', 'the', 'crow', '(', '9/10', ')', '-',
    'the', 'crow', ':', 'salvation', '(', '4/10', ')', '-', 'lost', 'highway', '(',
    '10/10', ')', '-', 'memento', '(', '10/10', ')', '-', 'the', 'others', '(', '9/10',
    ')', '-', 'stir', 'of', 'echoes', '(', '8/10', ')']
```

Listing 9.7: Example output of spitting a review by white space.

Just looking at the raw tokens can give us a lot of ideas of things to try, such as:

- Remove punctuation from words (e.g. 'what's').
- Removing tokens that are just punctuation (e.g. '-').
- Removing tokens that contain numbers (e.g. '10/10').
- Remove tokens that have one character (e.g. 'a').
- Remove tokens that don't have much meaning (e.g. 'and').

Some ideas:

- We can filter out punctuation from tokens using regular expressions.
- We can remove tokens that are just punctuation or contain numbers by using an `isalpha()` check on each token.
- We can remove English stop words using the list loaded using NLTK.
- We can filter out short tokens by checking their length.

Below is an updated version of cleaning this review.

```
from nltk.corpus import stopwords
import string
import re

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# load the document
filename = 'txt_sentoken/neg/cv000_29416.txt'
text = load_doc(filename)
# split into tokens by white space
tokens = text.split()
# prepare regex for char filtering
re_punc = re.compile('[%s]' % re.escape(string.punctuation))
# remove punctuation from each word
tokens = [re_punc.sub('', w) for w in tokens]
# remove remaining tokens that are not alphabetic
tokens = [word for word in tokens if word.isalpha()]
# filter out stop words
stop_words = set(stopwords.words('english'))
tokens = [w for w in tokens if not w in stop_words]
# filter out short tokens
tokens = [word for word in tokens if len(word) > 1]
print(tokens)
```

Listing 9.8: Load and clean one movie review.

Running the example gives a much cleaner looking list of tokens.

```
...
'explanation', 'craziness', 'came', 'oh', 'way', 'horror', 'teen', 'slasher', 'flick',
    'packaged', 'look', 'way', 'someone', 'apparently', 'assuming', 'genre', 'still',
    'hot', 'kids', 'also', 'wrapped', 'production', 'two', 'years', 'ago', 'sitting',
    'shelves', 'ever', 'since', 'whatever', 'skip', 'wheres', 'joblo', 'coming',
    'nightmare', 'elm', 'street', 'blair', 'witch', 'crow', 'crow', 'salvation', 'lost',
    'highway', 'memento', 'others', 'stir', 'echoes']
```

Listing 9.9: Example output of cleaning one movie review.

We can put this into a function called `clean_doc()` and test it on another review, this time a positive review.

```
from nltk.corpus import stopwords
import string
import re

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load the document
filename = 'txt_sentoken/pos/cv000_29590.txt'
text = load_doc(filename)
tokens = clean_doc(text)
print(tokens)
```

Listing 9.10: Function to clean movie reviews.

Again, the cleaning procedure seems to produce a good set of tokens, at least as a first cut.

```
...
'comic', 'oscar', 'winner', 'martin', 'childs', 'shakespeare', 'love', 'production',
    'design', 'turns', 'original', 'prague', 'surroundings', 'one', 'creepy', 'place',
    'even', 'acting', 'hell', 'solid', 'dreamy', 'depp', 'turning', 'typically', 'strong',
    'performance', 'deftly', 'handling', 'british', 'accent', 'ians', 'holm', 'joe',
    'goulds', 'secret', 'richardson', 'dalmatians', 'log', 'great', 'supporting', 'roles',
    'big', 'surprise', 'graham', 'cringed', 'first', 'time', 'opened', 'mouth',
    'imagining', 'attempt', 'irish', 'accent', 'actually', 'wasnt', 'half', 'bad', 'film',
    'however', 'good', 'strong', 'violencegore', 'sexuality', 'language', 'drug', 'content']
```

Listing 9.11: Example output of a function to clean movie reviews.

There are many more cleaning steps we could take and I leave them to your imagination. Next, let's look at how we can manage a preferred vocabulary of tokens.

9.5 Develop Vocabulary

When working with predictive models of text, like a bag-of-words model, there is a pressure to reduce the size of the vocabulary. The larger the vocabulary, the more sparse the representation of each word or document. A part of preparing text for sentiment analysis involves defining and tailoring the vocabulary of words supported by the model. We can do this by loading all of the documents in the dataset and building a set of words. We may decide to support all of these words, or perhaps discard some. The final chosen vocabulary can then be saved to file for later use, such as filtering words in new documents in the future.

We can keep track of the vocabulary in a `Counter`, which is a dictionary of words and their count with some additional convenience functions. We need to develop a new function to process a document and add it to the vocabulary. The function needs to load a document by calling the previously developed `load_doc()` function. It needs to clean the loaded document using the previously developed `clean_doc()` function, then it needs to add all the tokens to the `Counter`, and update counts. We can do this last step by calling the `update()` function on the counter object. Below is a function called `add_doc_to_vocab()` that takes as arguments a document filename and a `Counter` vocabulary.

```
# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
  # load doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # update counts
  vocab.update(tokens)
```

Listing 9.12: Function add a movie review to a vocabulary.

Finally, we can use our template above for processing all documents in a directory called `process_docs()` and update it to call `add_doc_to_vocab()`.

```
# load all docs in a directory
def process_docs(directory, vocab):
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip files that do not have the right extension
    if not filename.endswith(".txt"):
      next
    # create the full path of the file to open
    path = directory + '/' + filename
    # add doc to vocab
    add_doc_to_vocab(path, vocab)
```

Listing 9.13: Updated process documents function.

We can put all of this together and develop a full vocabulary from all documents in the dataset.

```
import string
import re
from os import listdir
from collections import Counter
from nltk.corpus import stopwords
```

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
  # load doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # update counts
  vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip files that do not have the right extension
    if not filename.endswith(".txt"):
      next
    # create the full path of the file to open
    path = directory + '/' + filename
    # add doc to vocab
    add_doc_to_vocab(path, vocab)

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/neg', vocab)
process_docs('txt_sentoken/pos', vocab)
# print the size of the vocab
print(len(vocab))
```

```
# print the top words in the vocab
print(vocab.most_common(50))
```

Listing 9.14: Example of cleaning all reviews and building a vocabulary.

Running the example creates a vocabulary with all documents in the dataset, including positive and negative reviews. We can see that there are a little over 46,000 unique words across all reviews and the top 3 words are *film*, *one*, and *movie*.

```
46557
[('film', 8860), ('one', 5521), ('movie', 5440), ('like', 3553), ('even', 2555), ('good',
    2320), ('time', 2283), ('story', 2118), ('films', 2102), ('would', 2042), ('much',
    2024), ('also', 1965), ('characters', 1947), ('get', 1921), ('character', 1906),
    ('two', 1825), ('first', 1768), ('see', 1730), ('well', 1694), ('way', 1668), ('make',
    1590), ('really', 1563), ('little', 1491), ('life', 1472), ('plot', 1451), ('people',
    1420), ('movies', 1416), ('could', 1395), ('bad', 1374), ('scene', 1373), ('never',
    1364), ('best', 1301), ('new', 1277), ('many', 1268), ('doesnt', 1267), ('man', 1266),
    ('scenes', 1265), ('dont', 1210), ('know', 1207), ('hes', 1150), ('great', 1141),
    ('another', 1111), ('love', 1089), ('action', 1078), ('go', 1075), ('us', 1065),
    ('director', 1056), ('something', 1048), ('end', 1047), ('still', 1038)]
```

Listing 9.15: Example output of building a vocabulary.

Perhaps the least common words, those that only appear once across all reviews, are not predictive. Perhaps some of the most common words are not useful too. These are good questions and really should be tested with a specific predictive model. Generally, words that only appear once or a few times across 2,000 reviews are probably not predictive and can be removed from the vocabulary, greatly cutting down on the tokens we need to model. We can do this by stepping through words and their counts and only keeping those with a count above a chosen threshold. Here we will use 5 occurrences.

```
# keep tokens with > 5 occurrence
min_occurane = 5
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
```

Listing 9.16: Example of filtering the vocabulary by an occurrence count.

This reduces the vocabulary from 46,557 to 14,803 words, a huge drop. Perhaps a minimum of 5 occurrences is too aggressive; you can experiment with different values. We can then save the chosen vocabulary of words to a new file. I like to save the vocabulary as ASCII with one word per line. Below defines a function called `save_list()` to save a list of items, in this case, tokens to file, one per line.

```
def save_list(lines, filename):
  data = '\n'.join(lines)
  file = open(filename, 'w')
  file.write(data)
  file.close()
```

Listing 9.17: Function to save the vocabulary to file.

The complete example for defining and saving the vocabulary is listed below.

```
import string
import re
from os import listdir
```

```
from collections import Counter
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
  # load doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # update counts
  vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip files that do not have the right extension
    if not filename.endswith(".txt"):
      next
    # create the full path of the file to open
    path = directory + '/' + filename
    # add doc to vocab
    add_doc_to_vocab(path, vocab)

# save list to file
def save_list(lines, filename):
  data = '\n'.join(lines)
  file = open(filename, 'w')
  file.write(data)
```

```
  file.close()

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/neg', vocab)
process_docs('txt_sentoken/pos', vocab)
# print the size of the vocab
print(len(vocab))
# print the top words in the vocab
print(vocab.most_common(50))
# keep tokens with > 5 occurrence
min_occurane = 5
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')
```

Listing 9.18: Example building and saving a final vocabulary.

Running this final snippet after creating the vocabulary will save the chosen words to file. It is a good idea to take a look at, and even study, your chosen vocabulary in order to get ideas for better preparing this data, or text data in the future.

```
hasnt
updating
figuratively
symphony
civilians
might
fisherman
hokum
witch
buffoons
...
```

Listing 9.19: Sample of the saved vocabulary file.

Next, we can look at using the vocabulary to create a prepared version of the movie review dataset.

9.6 Save Prepared Data

We can use the data cleaning and chosen vocabulary to prepare each movie review and save the prepared versions of the reviews ready for modeling. This is a good practice as it decouples the data preparation from modeling, allowing you to focus on modeling and circle back to data prep if you have new ideas. We can start off by loading the vocabulary from `vocab.txt`.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
```

```
  file.close()
  return text

# load vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)
```

Listing 9.20: Load the saved vocabulary.

Next, we can clean the reviews, use the loaded vocab to filter out unwanted tokens, and save the clean reviews in a new file. One approach could be to save all the positive reviews in one file and all the negative reviews in another file, with the filtered tokens separated by white space for each review on separate lines. First, we can define a function to process a document, clean it, filter it, and return it as a single line that could be saved in a file. Below defines the `doc_to_line()` function to do just that, taking a filename and vocabulary (as a set) as arguments. It calls the previously defined `load_doc()` function to load the document and `clean_doc()` to tokenize the document.

```
# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
  # load the doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # filter by vocab
  tokens = [w for w in tokens if w in vocab]
  return ' '.join(tokens)
```

Listing 9.21: Function to filter a review by the vocabulary

Next, we can define a new version of `process_docs()` to step through all reviews in a folder and convert them to lines by calling `doc_to_line()` for each document. A list of lines is then returned.

```
# load all docs in a directory
def process_docs(directory, vocab):
  lines = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip files that do not have the right extension
    if not filename.endswith(".txt"):
      next
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
  return lines
```

Listing 9.22: Updated process docs function to filter all documents by the vocabulary.

We can then call `process_docs()` for both the directories of positive and negative reviews, then call `save_list()` from the previous section to save each list of processed reviews to a file.

The complete code listing is provided below.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# save list to file
def save_list(lines, filename):
  data = '\n'.join(lines)
  file = open(filename, 'w')
  file.write(data)
  file.close()

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
  # load the doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # filter by vocab
  tokens = [w for w in tokens if w in vocab]
  return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
  lines = list()
  # walk through all files in the folder
  for filename in listdir(directory):
```

```
    # skip files that do not have the right extension
    if not filename.endswith(".txt"):
      next
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
  return lines

# load vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)
# prepare negative reviews
negative_lines = process_docs('txt_sentoken/neg', vocab)
save_list(negative_lines, 'negative.txt')
# prepare positive reviews
positive_lines = process_docs('txt_sentoken/pos', vocab)
save_list(positive_lines, 'positive.txt')
```

Listing 9.23: Example of cleaning and filtering all reviews by the vocab and saving the results to file.

Running the example saves two new files, `negative.txt` and `positive.txt`, that contain the prepared negative and positive reviews respectively. The data is ready for use in a bag-of-words or even word embedding model.

9.7 Further Reading

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

9.7.1 Dataset

- Movie Review Data.
  http://www.cs.cornell.edu/people/pabo/movie-review-data/
- *A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts*, 2004.
  http://xxx.lanl.gov/abs/cs/0409058
- Movie Review Polarity Dataset.
  http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz
- Dataset Readme v2.0 and v1.1.
  http://www.cs.cornell.edu/people/pabo/movie-review-data/poldata.README.2.0.txt
  http://www.cs.cornell.edu/people/pabo/movie-review-data/README.1.1

9.7.2 APIs

- `nltk.tokenize` package API.
  http://www.nltk.org/api/nltk.tokenize.html
- Chapter 2, *Accessing Text Corpora and Lexical Resources*.
  http://www.nltk.org/book/ch02.html
- `os` API Miscellaneous operating system interfaces.
  https://docs.python.org/3/library/os.html
- `collections` API - Container datatypes.
  https://docs.python.org/3/library/collections.html

9.8 Summary

In this tutorial, you discovered how to prepare movie review text data for sentiment analysis, step-by-step. Specifically, you learned:

- How to load text data and clean it to remove punctuation and other non-words.
- How to develop a vocabulary, tailor it, and save it to file.
- How to prepare movie reviews using cleaning and a predefined vocabulary and save them to new files ready for modeling.

9.8.1 Next

In the next chapter, you will discover how you can develop a neural bag-of-words model for movie review sentiment analysis.

Chapter 10

Project: Develop a Neural Bag-of-Words Model for Sentiment Analysis

Movie reviews can be classified as either favorable or not. The evaluation of movie review text is a classification problem often called sentiment analysis. A popular technique for developing sentiment analysis models is to use a bag-of-words model that transforms documents into vectors where each word in the document is assigned a score. In this tutorial, you will discover how you can develop a deep learning predictive model using the bag-of-words representation for movie review sentiment classification. After completing this tutorial, you will know:

- How to prepare the review text data for modeling with a restricted vocabulary.
- How to use the bag-of-words model to prepare train and test data.
- How to develop a Multilayer Perceptron bag-of-words model and use it to make predictions on new review text data.

Let's get started.

10.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Movie Review Dataset
2. Data Preparation
3. Bag-of-Words Representation
4. Sentiment Analysis Models
5. Comparing Word Scoring Methods
6. Predicting Sentiment for New Reviews

10.2 Movie Review Dataset

In this tutorial, we will use the Movie Review Dataset. This dataset designed for sentiment analysis was described previously in Chapter 9. You can download the dataset from here:

- Movie Review Polarity Dataset (`review_polarity.tar.gz`, 3MB). http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

After unzipping the file, you will have a directory called `txt_sentoken` with two sub-directories containing the text *neg* and *pos* for negative and positive reviews. Reviews are stored one per file with a naming convention *cv000* to *cv999* for each of *neg* and *pos*. Next, let's look at loading the text data.

10.3 Data Preparation

Note: The preparation of the movie review dataset was first described in Chapter 9. In this section, we will look at 3 things:

1. Separation of data into training and test sets.
2. Loading and cleaning the data to remove punctuation and numbers.
3. Defining a vocabulary of preferred words.

10.3.1 Split into Train and Test Sets

We are pretending that we are developing a system that can predict the sentiment of a textual movie review as either positive or negative. This means that after the model is developed, we will need to make predictions on new textual reviews. This will require all of the same data preparation to be performed on those new reviews as is performed on the training data for the model.

We will ensure that this constraint is built into the evaluation of our models by splitting the training and test datasets prior to any data preparation. This means that any knowledge in the test set that could help us better prepare the data (e.g. the words used) is unavailable during the preparation of data and the training of the model. That being said, we will use the last 100 positive reviews and the last 100 negative reviews as a test set (100 reviews) and the remaining 1,800 reviews as the training dataset. This is a 90% train, 10% split of the data. The split can be imposed easily by using the filenames of the reviews where reviews named 000 to 899 are for training data and reviews named 900 onwards are for testing the model.

10.3.2 Loading and Cleaning Reviews

The text data is already pretty clean, so not much preparation is required. Without getting too much into the details, we will prepare the data using the following method:

- Split tokens on white space.

- Remove all punctuation from words.
- Remove all words that are not purely comprised of alphabetical characters.
- Remove all words that are known stop words.
- Remove all words that have a length ≤ 1 character.

We can put all of these steps into a function called `clean_doc()` that takes as an argument the raw text loaded from a file and returns a list of cleaned tokens. We can also define a function `load_doc()` that loads a document from file ready for use with the `clean_doc()` function. An example of cleaning the first positive review is listed below.

```
from nltk.corpus import stopwords
import string
import re

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load the document
filename = 'txt_sentoken/pos/cv000_29590.txt'
text = load_doc(filename)
tokens = clean_doc(text)
print(tokens)
```

Listing 10.1: Example of cleaning a movie review.

Running the example prints a long list of clean tokens. There are many more cleaning steps we may want to explore, and I leave them as further exercises. I'd love to see what you can come up with.

```
...
'creepy', 'place', 'even', 'acting', 'hell', 'solid', 'dreamy', 'depp', 'turning',
    'typically', 'strong', 'performance', 'deftly', 'handling', 'british', 'accent',
    'ians', 'holm', 'joe', 'goulds', 'secret', 'richardson', 'dalmatians', 'log', 'great',
    'supporting', 'roles', 'big', 'surprise', 'graham', 'cringed', 'first', 'time',
    'opened', 'mouth', 'imagining', 'attempt', 'irish', 'accent', 'actually', 'wasnt',
    'half', 'bad', 'film', 'however', 'good', 'strong', 'violencegore', 'sexuality',
    'language', 'drug', 'content']
```

Listing 10.2: Example output of cleaning a movie review.

10.3.3 Define a Vocabulary

It is important to define a vocabulary of known words when using a bag-of-words model. The more words, the larger the representation of documents, therefore it is important to constrain the words to only those believed to be predictive. This is difficult to know beforehand and often it is important to test different hypotheses about how to construct a useful vocabulary. We have already seen how we can remove punctuation and numbers from the vocabulary in the previous section. We can repeat this for all documents and build a set of all known words.

We can develop a vocabulary as a `Counter`, which is a dictionary mapping of words and their count that allows us to easily update and query. Each document can be added to the counter (a new function called `add_doc_to_vocab()`) and we can step over all of the reviews in the negative directory and then the positive directory (a new function called `process_docs()`). The complete example is listed below.

```
import string
import re
from os import listdir
from collections import Counter
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
```

```
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
  # load doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # update counts
  vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # add doc to vocab
    add_doc_to_vocab(path, vocab)

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/pos', vocab)
process_docs('txt_sentoken/neg', vocab)
# print the size of the vocab
print(len(vocab))
# print the top words in the vocab
print(vocab.most_common(50))
```

Listing 10.3: Example of selecting a vocabulary for the dataset.

Running the example shows that we have a vocabulary of 44,276 words. We also can see a sample of the top 50 most used words in the movie reviews. Note that this vocabulary was constructed based on only those reviews in the training dataset.

```
44276
[('film', 7983), ('one', 4946), ('movie', 4826), ('like', 3201), ('even', 2262), ('good',
    2080), ('time', 2041), ('story', 1907), ('films', 1873), ('would', 1844), ('much',
    1824), ('also', 1757), ('characters', 1735), ('get', 1724), ('character', 1703),
    ('two', 1643), ('first', 1588), ('see', 1557), ('way', 1515), ('well', 1511), ('make',
    1418), ('really', 1407), ('little', 1351), ('life', 1334), ('plot', 1288), ('people',
    1269), ('could', 1248), ('bad', 1248), ('scene', 1241), ('movies', 1238), ('never',
    1201), ('best', 1179), ('new', 1140), ('scenes', 1135), ('man', 1131), ('many', 1130),
    ('doesnt', 1118), ('know', 1092), ('dont', 1086), ('hes', 1024), ('great', 1014),
    ('another', 992), ('action', 985), ('love', 977), ('us', 967), ('go', 952),
    ('director', 948), ('end', 946), ('something', 945), ('still', 936)]
```

Listing 10.4: Example output of selecting a vocabulary for the dataset.

We can step through the vocabulary and remove all words that have a low occurrence, such as only being used once or twice in all reviews. For example, the following snippet will retrieve only the tokens that appear 2 or more times in all reviews.

```
# keep tokens with a min occurrence
min_occurane = 2
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
```

Listing 10.5: Example of filtering the vocabulary by occurrence.

Finally, the vocabulary can be saved to a new file called `vocab.txt` that we can later load and use to filter movie reviews prior to encoding them for modeling. We define a new function called `save_list()` that saves the vocabulary to file, with one word per file. For example:

```
# save list to file
def save_list(lines, filename):
  # convert lines to a single blob of text
  data = '\n'.join(lines)
  # open file
  file = open(filename, 'w')
  # write text
  file.write(data)
  # close file
  file.close()

# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')
```

Listing 10.6: Example of saving the filtered vocabulary.

Pulling all of this together, the complete example is listed below.

```
import string
import re
from os import listdir
from collections import Counter
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
```

```
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
  # load doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # update counts
  vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # add doc to vocab
    add_doc_to_vocab(path, vocab)

# save list to file
def save_list(lines, filename):
  # convert lines to a single blob of text
  data = '\n'.join(lines)
  # open file
  file = open(filename, 'w')
  # write text
  file.write(data)
  # close file
  file.close()

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/pos', vocab)
process_docs('txt_sentoken/neg', vocab)
# print the size of the vocab
print(len(vocab))
# keep tokens with a min occurrence
min_occurane = 2
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')
```

Listing 10.7: Example of filtering the vocabulary for the dataset.

Running the above example with this addition shows that the vocabulary size drops by a little more than half its size, from about 44,000 to about 25,000 words.

```
25767
```

Listing 10.8: Example output of filtering the vocabulary by min occurrence.

Running the min occurrence filter on the vocabulary and saving it to file, you should now have a new file called `vocab.txt` with only the words we are interested in.

The order of words in your file will differ, but should look something like the following:

```
aberdeen
dupe
burt
libido
hamlet
arlene
available
corners
web
columbia
...
```

Listing 10.9: Sample of the vocabulary file `vocab.txt`.

We are now ready to look at extracting features from the reviews ready for modeling.

10.4 Bag-of-Words Representation

In this section, we will look at how we can convert each review into a representation that we can provide to a Multilayer Perceptron model. A bag-of-words model is a way of extracting features from text so the text input can be used with machine learning algorithms like neural networks. Each document, in this case a review, is converted into a vector representation. The number of items in the vector representing a document corresponds to the number of words in the vocabulary. The larger the vocabulary, the longer the vector representation, hence the preference for smaller vocabularies in the previous section. The bag-of-words model was introduced previously in Chapter 8.

Words in a document are scored and the scores are placed in the corresponding location in the representation. We will look at different word scoring methods in the next section. In this section, we are concerned with converting reviews into vectors ready for training a first neural network model. This section is divided into 2 steps:

1. Converting reviews to lines of tokens.
2. Encoding reviews with a bag-of-words model representation.

10.4.1 Reviews to Lines of Tokens

Before we can convert reviews to vectors for modeling, we must first clean them up. This involves loading them, performing the cleaning operation developed above, filtering out words not in the chosen vocabulary, and converting the remaining tokens into a single string or line

ready for encoding. First, we need a function to prepare one document. Below lists the function `doc_to_line()` that will load a document, clean it, filter out tokens not in the vocabulary, then return the document as a string of white space separated tokens.

```
# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
  # load the doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # filter by vocab
  tokens = [w for w in tokens if w in vocab]
  return ' '.join(tokens)
```

Listing 10.10: Function to filter a review by pre-defined vocabulary.

Next, we need a function to work through all documents in a directory (such as `pos` and `neg`) to convert the documents into lines. Below lists the `process_docs()` function that does just this, expecting a directory name and a vocabulary set as input arguments and returning a list of processed documents.

```
# load all docs in a directory
def process_docs(directory, vocab):
  lines = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
  return lines
```

Listing 10.11: Function to filter all movie reviews by vocabulary.

We can call the `process_docs()` consistently for positive and negative reviews to construct a dataset of review text and their associated output labels, 0 for negative and 1 for positive. The `load_clean_dataset()` function below implements this behavior.

```
# load and clean a dataset
def load_clean_dataset(vocab):
  # load documents
  neg = process_docs('txt_sentoken/neg', vocab)
  pos = process_docs('txt_sentoken/pos', vocab)
  docs = neg + pos
  # prepare labels
  labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
  return docs, labels
```

Listing 10.12: Function to load movie reviews and prepare output labels.

Finally, we need to load the vocabulary and turn it into a set for use in cleaning reviews.

```
# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
```

Listing 10.13: Load the pre-defined vocabulary of words.

We can put all of this together, reusing the loading and cleaning functions developed in previous sections. The complete example is listed below, demonstrating how to prepare the positive and negative reviews from the training dataset.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
  # load the doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # filter by vocab
  tokens = [w for w in tokens if w in vocab]
  return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
  lines = list()
```

```
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
  return lines

# load and clean a dataset
def load_clean_dataset(vocab):
  # load documents
  neg = process_docs('txt_sentoken/neg', vocab)
  pos = process_docs('txt_sentoken/pos', vocab)
  docs = neg + pos
  # prepare labels
  labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
  return docs, labels

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)
# load all training reviews
docs, labels = load_clean_dataset(vocab)
# summarize what we have
print(len(docs), len(labels))
```

Listing 10.14: Filter all movie reviews by the pre-defined vocabulary.

Running this example loads and cleans the review text and returns the labels.

```
1800 1800
```

Listing 10.15: Example output from loading, cleaning and filtering movie review data by a constrained vocabulary.

10.4.2 Movie Reviews to Bag-of-Words Vectors

We will use the Keras API to convert reviews to encoded document vectors. Keras provides the `Tokenizer` class that can do some of the cleaning and vocab definition tasks that we took care of in the previous section. It is better to do this ourselves to know exactly what was done and why. Nevertheless, the `Tokenizer` class is convenient and will easily transform documents into encoded vectors. First, the `Tokenizer` must be created, then fit on the text documents in the training dataset. In this case, these are the aggregation of the `positive_lines` and `negative_lines` arrays developed in the previous section.

```
# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
```

```
  tokenizer.fit_on_texts(lines)
  return tokenizer
```

Listing 10.16: Function to fit a `Tokenizer` on the clean and filtered movie reviews.

This process determines a consistent way to convert the vocabulary to a fixed-length vector with 25,768 elements, which is the total number of words in the vocabulary file `vocab.txt`. Next, documents can then be encoded using the `Tokenizer` by calling `texts_to_matrix()`. The function takes both a list of documents to encode and an encoding mode, which is the method used to score words in the document. Here we specify `freq` to score words based on their frequency in the document. This can be used to encode the loaded training and test data, for example:

```
# encode data
Xtrain = tokenizer.texts_to_matrix(train_docs, mode='freq')
Xtest = tokenizer.texts_to_matrix(test_docs, mode='freq')
```

Listing 10.17: Encode training data.

This encodes all of the positive and negative reviews in the training dataset.

Next, the `process_docs()` function from the previous section needs to be modified to selectively process reviews in the test or train dataset. We support the loading of both the training and test datasets by adding an `is_train` argument and using that to decide what review file names to skip.

```
# load all docs in a directory
def process_docs(directory, vocab, is_train):
  lines = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if is_train and filename.startswith('cv9'):
      continue
    if not is_train and not filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
  return lines
```

Listing 10.18: Updated process documents function to load all reviews.

Similarly, the `load_clean_dataset()` dataset must be updated to load either train or test data.

```
# load and clean a dataset
def load_clean_dataset(vocab, is_train):
  # load documents
  neg = process_docs('txt_sentoken/neg', vocab, is_train)
  pos = process_docs('txt_sentoken/pos', vocab, is_train)
  docs = neg + pos
  # prepare labels
  labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
```

```
  return docs, labels
```

Listing 10.19: Function to load text data and labels.

We can put all of this together in a single example.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords
from keras.preprocessing.text import Tokenizer

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
  # load the doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # filter by vocab
  tokens = [w for w in tokens if w in vocab]
  return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab, is_train):
  lines = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if is_train and filename.startswith('cv9'):
      continue
```

```
    if not is_train and not filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
  return lines

# load and clean a dataset
def load_clean_dataset(vocab, is_train):
  # load documents
  neg = process_docs('txt_sentoken/neg', vocab, is_train)
  pos = process_docs('txt_sentoken/pos', vocab, is_train)
  docs = neg + pos
  # prepare labels
  labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
  return docs, labels

# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# encode data
Xtrain = tokenizer.texts_to_matrix(train_docs, mode='freq')
Xtest = tokenizer.texts_to_matrix(test_docs, mode='freq')
print(Xtrain.shape, Xtest.shape)
```

Listing 10.20: Complete example of preparing train and test data.

Running the example prints both the shape of the encoded training dataset and test dataset with 1,800 and 200 documents respectively, each with the same sized encoding vocabulary (vector length).

```
(1800, 25768) (200, 25768)
```

Listing 10.21: Example output of loading and preparing the datasets.

10.5 Sentiment Analysis Models

In this section, we will develop Multilayer Perceptron (MLP) models to classify encoded documents as either positive or negative. The models will be simple feedforward network models

with fully connected layers called `Dense` in the Keras deep learning library. This section is divided into 3 sections:

1. First sentiment analysis model
2. Comparing word scoring modes
3. Making a prediction for new reviews

10.5.1 First Sentiment Analysis Model

We can develop a simple MLP model to predict the sentiment of encoded reviews. The model will have an input layer that equals the number of words in the vocabulary, and in turn the length of the input documents. We can store this in a new variable called `n_words`, as follows:

```
n_words = Xtest.shape[1]
```

Listing 10.22: Example of calculating the number of words.

We can now define the network. All model configuration was found with very little trial and error and should not be considered tuned for this problem. We will use a single hidden layer with 50 neurons and a rectified linear activation function. The output layer is a single neuron with a sigmoid activation function for predicting 0 for negative and 1 for positive reviews. The network will be trained using the efficient Adam implementation of gradient descent and the binary cross entropy loss function, suited to binary classification problems. We will keep track of accuracy when training and evaluating the model.

```
# define the model
def define_model(n_words):
  # define network
  model = Sequential()
  model.add(Dense(50, input_shape=(n_words,), activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  # compile network
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model
```

Listing 10.23: Example of defining an MLP for the bag-of-words model.

Next, we can fit the model on the training data; in this case, the model is small and is easily fit in 10 epochs.

```
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
```

Listing 10.24: Example of fitting the defined model.

Finally, once the model is trained, we can evaluate its performance by making predictions in the test dataset and printing the accuracy.

```
# evaluate
loss, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %f' % (acc*100))
```

Listing 10.25: Example of evaluating the fit model.

The complete example is listed below.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords
from keras.preprocessing.text import Tokenizer
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
  # load the doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # filter by vocab
  tokens = [w for w in tokens if w in vocab]
  return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab, is_train):
```

```
  lines = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if is_train and filename.startswith('cv9'):
      continue
    if not is_train and not filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
  return lines

# load and clean a dataset
def load_clean_dataset(vocab, is_train):
  # load documents
  neg = process_docs('txt_sentoken/neg', vocab, is_train)
  pos = process_docs('txt_sentoken/pos', vocab, is_train)
  docs = neg + pos
  # prepare labels
  labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
  return docs, labels

# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer

# define the model
def define_model(n_words):
  # define network
  model = Sequential()
  model.add(Dense(50, input_shape=(n_words,), activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  # compile network
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# encode data
Xtrain = tokenizer.texts_to_matrix(train_docs, mode='freq')
```

```
Xtest = tokenizer.texts_to_matrix(test_docs, mode='freq')
# define the model
n_words = Xtest.shape[1]
model = define_model(n_words)
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
# evaluate
loss, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %f' % (acc*100))
```

Listing 10.26: Complete example of training and evaluating an MLP bag-of-words model.

Running the example first prints a summary of the defined model.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 50)                1288450
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 51
=================================================================
Total params: 1,288,501
Trainable params: 1,288,501
Non-trainable params: 0
_________________________________________________________________
```

Listing 10.27: Summary of the defined model.

A plot the defined model is then saved to file with the name `model.png`.

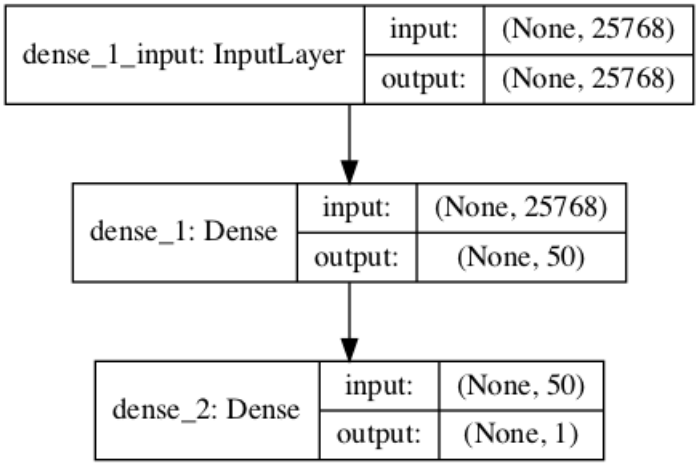

Figure 10.1: Plot of the defined bag-of-words model.

We can see that the model easily fits the training data within the 10 epochs, achieving close to 100% accuracy. Evaluating the model on the test dataset, we can see that model does well, achieving an accuracy of above 87%, well within the ballpark of low-to-mid 80s seen in the original paper. Although, it is important to note that this is not an apples-to-apples comparison, as the original paper used 10-fold cross-validation to estimate model skill instead of a single train/test split.

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

```
...
Epoch 6/10
0s - loss: 0.5319 - acc: 0.9428
Epoch 7/10
0s - loss: 0.4839 - acc: 0.9506
Epoch 8/10
0s - loss: 0.4368 - acc: 0.9567
Epoch 9/10
0s - loss: 0.3927 - acc: 0.9611
Epoch 10/10
0s - loss: 0.3516 - acc: 0.9689

Test Accuracy: 87.000000
```

Listing 10.28: Example output of training and evaluating the MLP model.

Next, let's look at testing different word scoring methods for the bag-of-words model.

10.6 Comparing Word Scoring Methods

The `texts_to_matrix()` function for the `Tokenizer` in the Keras API provides 4 different methods for scoring words; they are:

- `binary` Where words are marked as present (1) or absent (0).
- `count` Where the occurrence count for each word is marked as an integer.
- `tfidf` Where each word is scored based on their frequency, where words that are common across all documents are penalized.
- `freq` Where words are scored based on their frequency of occurrence within the document.

We can evaluate the skill of the model developed in the previous section fit using each of the 4 supported word scoring modes. This first involves the development of a function to create an encoding of the loaded documents based on a chosen scoring model. The function creates the tokenizer, fits it on the training documents, then creates the train and test encodings using the chosen model. The function `prepare_data()` implements this behavior given lists of train and test documents.

```
# prepare bag-of-words encoding of docs
def prepare_data(train_docs, test_docs, mode):
  # create the tokenizer
  tokenizer = Tokenizer()
  # fit the tokenizer on the documents
  tokenizer.fit_on_texts(train_docs)
  # encode training data set
  Xtrain = tokenizer.texts_to_matrix(train_docs, mode=mode)
  # encode training data set
  Xtest = tokenizer.texts_to_matrix(test_docs, mode=mode)
```

```
  return Xtrain, Xtest
```

Listing 10.29: Updated data preparation to take encoding mode as a parameter.

We also need a function to evaluate the MLP given a specific encoding of the data. Because neural networks are stochastic, they can produce different results when the same model is fit on the same data. This is mainly because of the random initial weights and the shuffling of patterns during mini-batch gradient descent. This means that any one scoring of a model is unreliable and we should estimate model skill based on an average of multiple runs. The function below, named `evaluate_mode()`, takes encoded documents and evaluates the MLP by training it on the train set and estimating skill on the test set 10 times and returns a list of the accuracy scores across all of these runs.

```
# evaluate a neural network model
def evaluate_mode(Xtrain, ytrain, Xtest, ytest):
  scores = list()
  n_repeats = 30
  n_words = Xtest.shape[1]
  for i in range(n_repeats):
    # define network
    model = Sequential()
    model.add(Dense(50, input_shape=(n_words,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile network
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit network
    model.fit(Xtrain, ytrain, epochs=10, verbose=2)
    # evaluate
    loss, acc = model.evaluate(Xtest, ytest, verbose=0)
    scores.append(acc)
    print('%d accuracy: %s' % ((i+1), acc))
  return scores
```

Listing 10.30: Function to create, fit and evaluate a model multiple times.

We are now ready to evaluate the performance of the 4 different word scoring methods. Pulling all of this together, the complete example is listed below.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords
from keras.preprocessing.text import Tokenizer
from keras.models import Sequential
from keras.layers import Dense
from pandas import DataFrame
from matplotlib import pyplot

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
```

```
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
  # load the doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # filter by vocab
  tokens = [w for w in tokens if w in vocab]
  return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab, is_train):
  lines = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if is_train and filename.startswith('cv9'):
      continue
    if not is_train and not filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
  return lines

# load and clean a dataset
def load_clean_dataset(vocab, is_train):
  # load documents
  neg = process_docs('txt_sentoken/neg', vocab, is_train)
  pos = process_docs('txt_sentoken/pos', vocab, is_train)
  docs = neg + pos
  # prepare labels
  labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
  return docs, labels
```

```
# define the model
def define_model(n_words):
  # define network
  model = Sequential()
  model.add(Dense(50, input_shape=(n_words,), activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  # compile network
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model

# evaluate a neural network model
def evaluate_mode(Xtrain, ytrain, Xtest, ytest):
  scores = list()
  n_repeats = 10
  n_words = Xtest.shape[1]
  for i in range(n_repeats):
    # define network
    model = define_model(n_words)
    # fit network
    model.fit(Xtrain, ytrain, epochs=10, verbose=0)
    # evaluate
    _, acc = model.evaluate(Xtest, ytest, verbose=0)
    scores.append(acc)
    print('%d accuracy: %s' % ((i+1), acc))
  return scores

# prepare bag of words encoding of docs
def prepare_data(train_docs, test_docs, mode):
  # create the tokenizer
  tokenizer = Tokenizer()
  # fit the tokenizer on the documents
  tokenizer.fit_on_texts(train_docs)
  # encode training data set
  Xtrain = tokenizer.texts_to_matrix(train_docs, mode=mode)
  # encode training data set
  Xtest = tokenizer.texts_to_matrix(test_docs, mode=mode)
  return Xtrain, Xtest

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
# run experiment
modes = ['binary', 'count', 'tfidf', 'freq']
results = DataFrame()
for mode in modes:
  # prepare data for mode
  Xtrain, Xtest = prepare_data(train_docs, test_docs, mode)
  # evaluate model on data for mode
  results[mode] = evaluate_mode(Xtrain, ytrain, Xtest, ytest)
# summarize results
print(results.describe())
```

```
# plot results
results.boxplot()
pyplot.show()
```

Listing 10.31: Complete example of comparing document encoding schemes.

At the end of the run, summary statistics for each word scoring method are provided, summarizing the distribution of model skill scores across each of the 10 runs per mode. We can see that the mean score of both the `count` and `binary` methods appear to be better than `freq` and `tfidf`.

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

```
         binary     count     tfidf      freq
count  10.000000 10.000000 10.000000 10.000000
mean    0.927000  0.903500  0.876500  0.871000
std     0.011595  0.009144  0.017958  0.005164
min     0.910000  0.885000  0.855000  0.865000
25%     0.921250  0.900000  0.861250  0.866250
50%     0.927500  0.905000  0.875000  0.870000
75%     0.933750  0.908750  0.888750  0.875000
max     0.945000  0.915000  0.910000  0.880000
```

Listing 10.32: Example output of comparing document encoding schemes.

A box and whisker plot of the results is also presented, summarizing the accuracy distributions per configuration. We can see that `binary` achieved the best results with a modest spread and might be the preferred approach for this dataset.

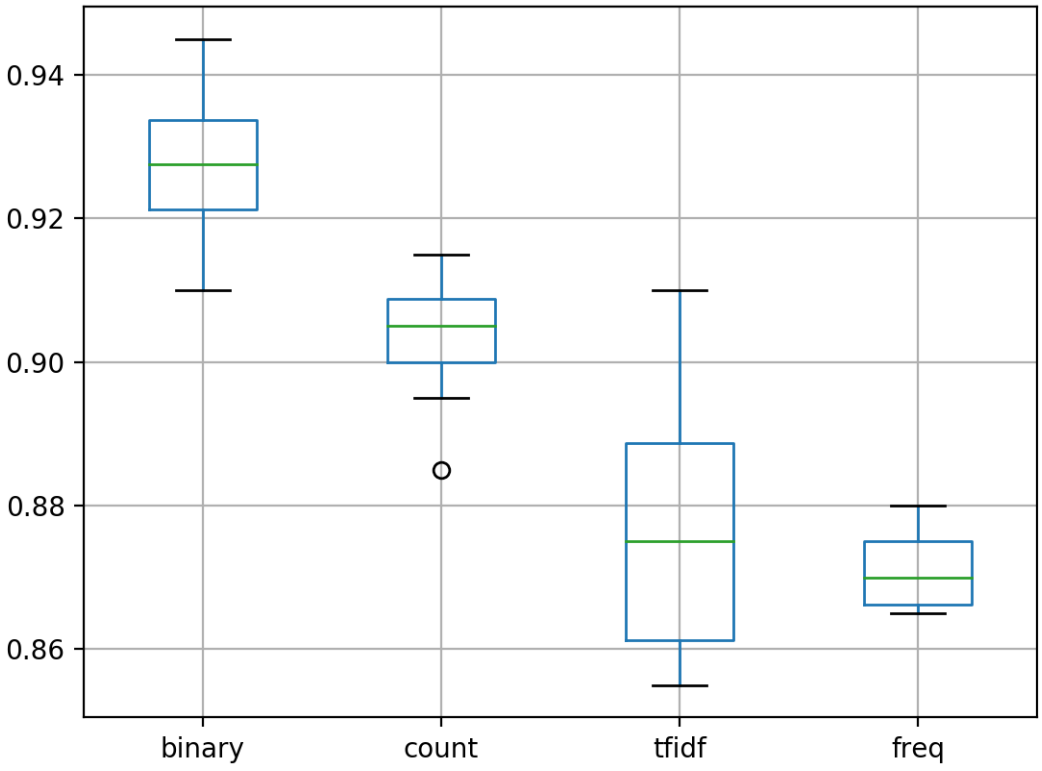

Figure 10.2: Box and Whisker Plot for Model Accuracy with Different Word Scoring Methods.

10.7 Predicting Sentiment for New Reviews

Finally, we can develop and use a final model to make predictions for new textual reviews. This is why we wanted the model in the first place. First we will train a final model on all of the available data. We will use the `binary` mode for scoring the bag-of-words model that was shown to give the best results in the previous section.

Predicting the sentiment of new reviews involves following the same steps used to prepare the test data. Specifically, loading the text, cleaning the document, filtering tokens by the chosen vocabulary, converting the remaining tokens to a line, encoding it using the `Tokenizer`, and making a prediction. We can make a prediction of a class value directly with the fit model by calling `predict()` that will return an integer of 0 for a negative review and 1 for a positive review. All of these steps can be put into a new function called `predict_sentiment()` that requires the review text, the vocabulary, the tokenizer, and the fit model and returns the predicted sentiment and an associated percentage or confidence-like output.

```
# classify a review as negative or positive
def predict_sentiment(review, vocab, tokenizer, model):
  # clean
  tokens = clean_doc(review)
  # filter by vocab
```

```
  tokens = [w for w in tokens if w in vocab]
  # convert to line
  line = ' '.join(tokens)
  # encode
  encoded = tokenizer.texts_to_matrix([line], mode='binary')
  # predict sentiment
  yhat = model.predict(encoded, verbose=0)
  # retrieve predicted percentage and label
  percent_pos = yhat[0,0]
  if round(percent_pos) == 0:
    return (1-percent_pos), 'NEGATIVE'
  return percent_pos, 'POSITIVE'
```

Listing 10.33: Function for making predictions for new reviews.

We can now make predictions for new review texts. Below is an example with both a clearly positive and a clearly negative review using the simple MLP developed above with the frequency word scoring mode.

```
# test positive text
text = 'Best movie ever! It was great, I recommend it.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))
# test negative text
text = 'This is a bad movie.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))
```

Listing 10.34: Exampling of making predictions for new reviews.

Pulling this all together, the complete example for making predictions for new reviews is listed below.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords
from keras.preprocessing.text import Tokenizer
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
```

```
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
  # load the doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # filter by vocab
  tokens = [w for w in tokens if w in vocab]
  return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
  lines = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
  return lines

# load and clean a dataset
def load_clean_dataset(vocab):
  # load documents
  neg = process_docs('txt_sentoken/neg', vocab)
  pos = process_docs('txt_sentoken/pos', vocab)
  docs = neg + pos
  # prepare labels
  labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
  return docs, labels

# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer

# define the model
def define_model(n_words):
  # define network
  model = Sequential()
  model.add(Dense(50, input_shape=(n_words,), activation='relu'))
```

```
  model.add(Dense(1, activation='sigmoid'))
  # compile network
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model

# classify a review as negative or positive
def predict_sentiment(review, vocab, tokenizer, model):
  # clean
  tokens = clean_doc(review)
  # filter by vocab
  tokens = [w for w in tokens if w in vocab]
  # convert to line
  line = ' '.join(tokens)
  # encode
  encoded = tokenizer.texts_to_matrix([line], mode='binary')
  # predict sentiment
  yhat = model.predict(encoded, verbose=0)
  # retrieve predicted percentage and label
  percent_pos = yhat[0,0]
  if round(percent_pos) == 0:
    return (1-percent_pos), 'NEGATIVE'
  return percent_pos, 'POSITIVE'

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab)
test_docs, ytest = load_clean_dataset(vocab)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# encode data
Xtrain = tokenizer.texts_to_matrix(train_docs, mode='binary')
Xtest = tokenizer.texts_to_matrix(test_docs, mode='binary')
# define network
n_words = Xtrain.shape[1]
model = define_model(n_words)
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
# test positive text
text = 'Best movie ever! It was great, I recommend it.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))
# test negative text
text = 'This is a bad movie.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))
```

Listing 10.35: Complete example of making predictions for new review data.

Running the example correctly classifies these reviews.

```
Review: [Best movie ever! It was great, I recommend it.]
```

```
Sentiment: POSITIVE (57.124%)
Review: [This is a bad movie.]
Sentiment: NEGATIVE (64.404%)
```

Listing 10.36: Example output of making predictions for new reviews.

Ideally, we would fit the model on all available data (train and test) to create a final model and save the model and tokenizer to file so that they can be loaded and used in new software.

10.8 Extensions

This section lists some extensions if you are looking to get more out of this tutorial.

- **Manage Vocabulary**. Explore using a larger or smaller vocabulary. Perhaps you can get better performance with a smaller set of words.
- **Tune the Network Topology**. Explore alternate network topologies such as deeper or wider networks. Perhaps you can get better performance with a more suited network.
- **Use Regularization**. Explore the use of regularization techniques, such as dropout. Perhaps you can delay the convergence of the model and achieve better test set performance.
- **More Data Cleaning**. Explore more or less cleaning of the review text and see how it impacts the model skill.
- **Training Diagnostics**. Use the test dataset as a validation dataset during training and create plots of train and test loss. Use these diagnostics to tune the batch size and number of training epochs.
- **Trigger Words**. Explore whether there are specific words in reviews that are highly predictive of the sentiment.
- **Use Bigrams**. Prepare the model to score bigrams of words and evaluate the performance under different scoring schemes.
- **Truncated Reviews**. Explore how using a truncated version of the movie reviews results impacts model skill, try truncating the start, end and middle of reviews.
- **Ensemble Models**. Create models with different word scoring schemes and see if using ensembles of the models results in improves to model skill.
- **Real Reviews**. Train a final model on all data and evaluate the model on real movie reviews taken from the internet.

If you explore any of these extensions, I'd love to know.

10.9 Further Reading

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

10.9.1 Dataset

- Movie Review Data.
  http://www.cs.cornell.edu/people/pabo/movie-review-data/
- *A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts*, 2004.
  http://xxx.lanl.gov/abs/cs/0409058
- Movie Review Polarity Dataset.
  http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

10.9.2 APIs

- `nltk.tokenize` package API.
  http://www.nltk.org/api/nltk.tokenize.html
- Chapter 2, *Accessing Text Corpora and Lexical Resources*.
  http://www.nltk.org/book/ch02.html
- `os` API Miscellaneous operating system interfaces.
  https://docs.python.org/3/library/os.html
- `collections` API - Container datatypes.
  https://docs.python.org/3/library/collections.html
- `Tokenizer` Keras API.
  https://keras.io/preprocessing/text/#tokenizer

10.10 Summary

In this tutorial, you discovered how to develop a bag-of-words model for predicting the sentiment of movie reviews. Specifically, you learned:

- How to prepare the review text data for modeling with a restricted vocabulary.
- How to use the bag-of-words model to prepare train and test data.
- How to develop a Multilayer Perceptron bag-of-words model and use it to make predictions on new review text data.

10.10.1 Next

This is the final chapter in the bag-of-words part. In the next part, you will discover how to develop word embedding models.

Part V

Word Embeddings

Chapter 11

The Word Embedding Model

Word embeddings are a type of word representation that allows words with similar meaning to have a similar representation. They are a distributed representation for text that is perhaps one of the key breakthroughs for the impressive performance of deep learning methods on challenging natural language processing problems. In this chapter, you will discover the word embedding approach for representing text data. After completing this chapter, you will know:

- What the word embedding approach for representing text is and how it differs from other feature extraction methods.
- That there are 3 main algorithms for learning a word embedding from text data.
- That you can either train a new embedding or use a pre-trained embedding on your natural language processing task.

Let's get started.

11.1 Overview

This tutorial is divided into the following parts:

1. What Are Word Embeddings?
2. Word Embedding Algorithms
3. Using Word Embeddings

11.2 What Are Word Embeddings?

A word embedding is a learned representation for text where words that have the same meaning have a similar representation. It is this approach to representing words and documents that may be considered one of the key breakthroughs of deep learning on challenging natural language processing problems.

> One of the benefits of using dense and low-dimensional vectors is computational: the majority of neural network toolkits do not play well with very high-dimensional, sparse vectors. ... The main benefit of the dense representations is generalization power: if we believe some features may provide similar clues, it is worthwhile to provide a representation that is able to capture these similarities.
>
> — Page 92, *Neural Network Methods in Natural Language Processing*, 2017.

Word embeddings are in fact a class of techniques where individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network, and hence the technique is often lumped into the field of deep learning.

Key to the approach is the idea of using a dense distributed representation for each word. Each word is represented by a real-valued vector, often tens or hundreds of dimensions. This is contrasted to the thousands or millions of dimensions required for sparse word representations, such as a one hot encoding.

> associate with each word in the vocabulary a distributed word feature vector ... The feature vector represents different aspects of the word: each word is associated with a point in a vector space. The number of features ... is much smaller than the size of the vocabulary
>
> — *A Neural Probabilistic Language Model*, 2003.

The distributed representation is learned based on the usage of words. This allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning. This can be contrasted with the crisp but fragile representation in a bag-of-words model where, unless explicitly managed, different words have different representations, regardless of how they are used.

There is deeper linguistic theory behind the approach, namely the *distributional hypothesis* by Zellig Harris that could be summarized as: words that have similar context will have similar meanings. For more depth see Harris' 1956 paper *Distributional structure*. This notion of letting the usage of the word define its meaning can be summarized by an oft repeated quip by John Firth:

> You shall know a word by the company it keeps!
>
> — Page 11, *A synopsis of linguistic theory 1930-1955*, in Studies in Linguistic Analysis 1930-1955, 1962.

11.3 Word Embedding Algorithms

Word embedding methods learn a real-valued vector representation for a predefined fixed sized vocabulary from a corpus of text. The learning process is either joint with the neural network model on some task, such as document classification, or is an unsupervised process, using document statistics. This section reviews three techniques that can be used to learn a word embedding from text data.

11.3.1 Embedding Layer

An embedding layer, for lack of a better name, is a word embedding that is learned jointly with a neural network model on a specific natural language processing task, such as language modeling or document classification. It requires that document text be cleaned and prepared such that each word is one hot encoded. The size of the vector space is specified as part of the model, such as 50, 100, or 300 dimensions. The vectors are initialized with small random numbers. The embedding layer is used on the front end of a neural network and is fit in a supervised way using the Backpropagation algorithm.

> ... when the input to a neural network contains symbolic categorical features (e.g. features that take one of k distinct symbols, such as words from a closed vocabulary), it is common to associate each possible feature value (i.e., each word in the vocabulary) with a d-dimensional vector for some d. These vectors are then considered parameters of the model, and are trained jointly with the other parameters.
>
> — Page 49, *Neural Network Methods in Natural Language Processing*, 2017.

The one hot encoded words are mapped to the word vectors. If a Multilayer Perceptron model is used, then the word vectors are concatenated before being fed as input to the model. If a recurrent neural network is used, then each word may be taken as one input in a sequence. This approach of learning an embedding layer requires a lot of training data and can be slow, but will learn an embedding both targeted to the specific text data and the NLP task.

11.3.2 Word2Vec

Word2Vec is a statistical method for efficiently learning a standalone word embedding from a text corpus. It was developed by Tomas Mikolov, et al. at Google in 2013 as a response to make the neural-network-based training of the embedding more efficient and since then has become the de facto standard for developing pre-trained word embedding.

Additionally, the work involved analysis of the learned vectors and the exploration of vector math on the representations of words. For example, that subtracting the *man-ness* from *King* and adding *women-ness* results in the word *Queen*, capturing the analogy *king is to queen as man is to woman*.

> We find that these representations are surprisingly good at capturing syntactic and semantic regularities in language, and that each relationship is characterized by a relation-specific vector offset. This allows vector-oriented reasoning based on the offsets between words. For example, the male/female relationship is automatically learned, and with the induced vector representations, *King - Man + Woman* results in a vector very close to *Queen*.
>
> — *Linguistic Regularities in Continuous Space Word Representations*, 2013.

Two different learning models were introduced that can be used as part of the Word2Vec approach to learn the word embedding; they are:

- Continuous Bag-of-Words, or CBOW model.
- Continuous Skip-Gram Model.

The CBOW model learns the embedding by predicting the current word based on its context. The continuous skip-gram model learns by predicting the surrounding words given a current word.

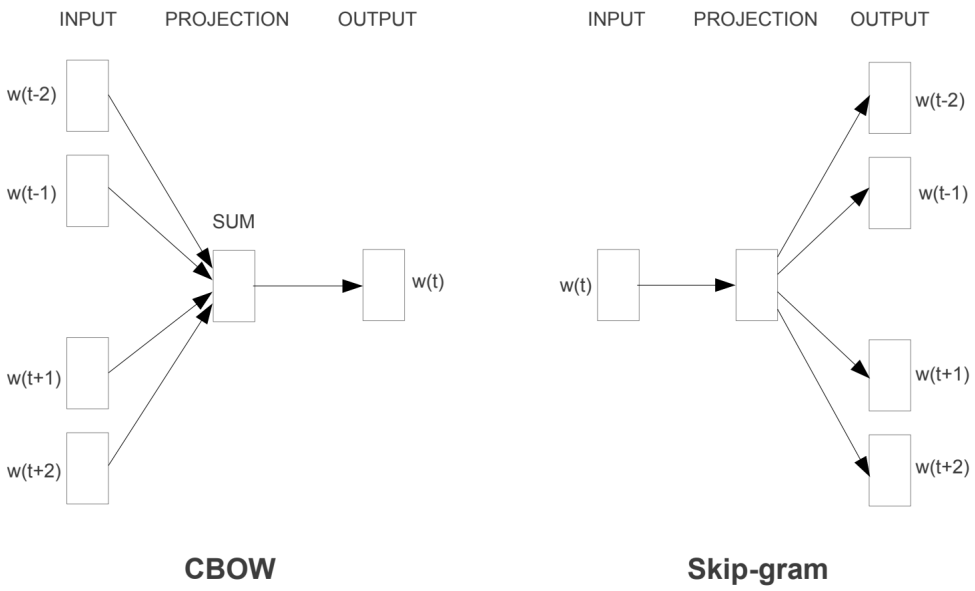

Figure 11.1: Word2Vec Training Models. Taken from *Efficient Estimation of Word Representations in Vector Space*, 2013

Both models are focused on learning about words given their local usage context, where the context is defined by a window of neighboring words. This window is a configurable parameter of the model.

> The size of the sliding window has a strong effect on the resulting vector similarities. Large windows tend to produce more topical similarities [...], while smaller windows tend to produce more functional and syntactic similarities.
>
> — Page 128, *Neural Network Methods in Natural Language Processing*, 2017.

The key benefit of the approach is that high-quality word embeddings can be learned efficiently (low space and time complexity), allowing larger embeddings to be learned (more dimensions) from much larger corpora of text (billions of words).

11.3.3 GloVe

The Global Vectors for Word Representation, or GloVe, algorithm is an extension to the Word2Vec method for efficiently learning word vectors, developed by Pennington, et al. at Stanford. Classical vector space model representations of words were developed using matrix

factorization techniques such as Latent Semantic Analysis (LSA) that do a good job of using global text statistics but are not as good as the learned methods like Word2Vec at capturing meaning and demonstrating it on tasks like calculating analogies (e.g. the King and Queen example above).

GloVe is an approach to marry both the global statistics of matrix factorization techniques like LSA with the local context-based learning in Word2Vec. Rather than using a window to define local context, GloVe constructs an explicit word-context or word co-occurrence matrix using statistics across the whole text corpus. The result is a learning model that may result in generally better word embeddings.

> GloVe, is a new global log-bilinear regression model for the unsupervised learning of word representations that outperforms other models on word analogy, word similarity, and named entity recognition tasks.
>
> — *GloVe: Global Vectors for Word Representation*, 2014.

11.4 Using Word Embeddings

You have some options when it comes time to using word embeddings on your natural language processing project. This section outlines those options.

11.4.1 Learn an Embedding

You may choose to learn a word embedding for your problem. This will require a large amount of text data to ensure that useful embeddings are learned, such as millions or billions of words. You have two main options when training your word embedding:

- **Learn it Standalone**, where a model is trained to learn the embedding, which is saved and used as a part of another model for your task later. This is a good approach if you would like to use the same embedding in multiple models.
- **Learn Jointly**, where the embedding is learned as part of a large task-specific model. This is a good approach if you only intend to use the embedding on one task.

11.4.2 Reuse an Embedding

It is common for researchers to make pre-trained word embeddings available for free, often under a permissive license so that you can use them on your own academic or commercial projects. For example, both Word2Vec and GloVe word embeddings are available for free download. These can be used on your project instead of training your own embeddings from scratch. You have two main options when it comes to using pre-trained embeddings:

- **Static**, where the embedding is kept static and is used as a component of your model. This is a suitable approach if the embedding is a good fit for your problem and gives good results.
- **Updated**, where the pre-trained embedding is used to seed the model, but the embedding is updated jointly during the training of the model. This may be a good option if you are looking to get the most out of the model and embedding on your task.

11.4.3 Which Option Should You Use?

Explore the different options, and if possible, test to see which gives the best results on your problem. Perhaps start with fast methods, like using a pre-trained embedding, and only use a new embedding if it results in better performance on your problem.

11.5 Further Reading

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

11.5.1 Books

- Neural Network Methods in Natural Language Processing, 2017.
  http://amzn.to/2wycQKA

11.5.2 Articles

- Word embedding on Wikipedia.
  https://en.wikipedia.org/wiki/Word_embedding
- Word2Vec on Wikipedia.
  https://en.wikipedia.org/wiki/Word2vec
- GloVe on Wikipedia
  https://en.wikipedia.org/wiki/GloVe_(machine_learning)
- *An overview of word embeddings and their connection to distributional semantic models*, 2016.
  http://blog.aylien.com/overview-word-embeddings-history-word2vec-cbow-glove/
- *Deep Learning, NLP, and Representations*, 2014.
  http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/

11.5.3 Papers

- *Distributional structure*, 1956.
  http://www.tandfonline.com/doi/pdf/10.1080/00437956.1954.11659520
- *A Neural Probabilistic Language Model*, 2003.
  http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf
- *A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning*, 2008.
  https://ronan.collobert.com/pub/matos/2008_nlp_icml.pdf
- *Continuous space language models*, 2007.
  https://pdfs.semanticscholar.org/0fcc/184b3b90405ec3ceafd6a4007c749df7c363.pdf

- *Efficient Estimation of Word Representations in Vector Space*, 2013.
  https://arxiv.org/pdf/1301.3781.pdf
- *Distributed Representations of Words and Phrases and their Compositionality*, 2013.
  https://arxiv.org/pdf/1310.4546.pdf
- *GloVe: Global Vectors for Word Representation*, 2014.
  https://nlp.stanford.edu/pubs/glove.pdf

11.5.4 Projects

- Word2Vec on Google Code.
  https://code.google.com/archive/p/word2vec/
- GloVe: Global Vectors for Word Representation.
  https://nlp.stanford.edu/projects/glove/

11.6 Summary

In this chapter, you discovered Word Embeddings as a representation method for text in deep learning applications. Specifically, you learned:

- What the word embedding approach for representation text is and how it differs from other feature extraction methods.
- That there are 3 main algorithms for learning a word embedding from text data.
- That you can either train a new embedding or use a pre-trained embedding on your natural language processing task.

11.6.1 Next

In the next chapter, you will discover how you can train and manipulate word embeddings using the Gensim Python library.

Chapter 12

How to Develop Word Embeddings with Gensim

Word embeddings are a modern approach for representing text in natural language processing. Embedding algorithms like Word2Vec and GloVe are key to the state-of-the-art results achieved by neural network models on natural language processing problems like machine translation. In this tutorial, you will discover how to train and load word embedding models for natural language processing applications in Python using Gensim. After completing this tutorial, you will know:

- How to train your own Word2Vec word embedding model on text data.
- How to visualize a trained word embedding model using Principal Component Analysis.
- How to load pre-trained Word2Vec and GloVe word embedding models from Google and Stanford.

Let's get started.

12.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Word Embeddings
2. Gensim Library
3. Develop Word2Vec Embedding
4. Visualize Word Embedding
5. Load Google's Word2Vec Embedding
6. Load Stanford's GloVe Embedding

12.2 Word Embeddings

A word embedding is an approach to provide a dense vector representation of words that capture something about their meaning. Word embeddings are an improvement over simpler bag-of-word model word encoding schemes like word counts and frequencies that result in large and sparse vectors (mostly 0 values) that describe documents but not the meaning of the words.

Word embeddings work by using an algorithm to train a set of fixed-length dense and continuous-valued vectors based on a large corpus of text. Each word is represented by a point in the embedding space and these points are learned and moved around based on the words that surround the target word. It is defining a word by the company that it keeps that allows the word embedding to learn something about the meaning of words. The vector space representation of the words provides a projection where words with similar meanings are locally clustered within the space.

The use of word embeddings over other text representations is one of the key methods that has led to breakthrough performance with deep neural networks on problems like machine translation. In this tutorial, we are going to look at how to use two different word embedding methods called Word2Vec by researchers at Google and GloVe by researchers at Stanford.

12.3 Gensim Python Library

Gensim is an open source Python library for natural language processing, with a focus on topic modeling. It is billed as "*topic modeling for humans*". Gensim was developed and is maintained by the Czech natural language processing researcher Radim Rehurek and his company RaRe Technologies. It is not an everything-including-the-kitchen-sink NLP research library (like NLTK); instead, Gensim is a mature, focused, and efficient suite of NLP tools for topic modeling. Most notably for this tutorial, it supports an implementation of the Word2Vec word embedding for learning new word vectors from text.

It also provides tools for loading pre-trained word embeddings in a few formats and for making use and querying a loaded embedding. We will use the Gensim library in this tutorial. Gensim can be installed easily using `pip` or `easy_install`. For example, you can install Gensim with `pip` by typing the following on your command line:

```
sudo pip install -U gensim
```

Listing 12.1: Install the Gensim library with `pip`.

If you need help installing Gensim on your system, you can see the Gensim Installation Instructions (linked at the end of the chapter).

12.4 Develop Word2Vec Embedding

Word2Vec is one algorithm for learning a word embedding from a text corpus. There are two main training algorithms that can be used to learn the embedding from text; they are Continuous Bag-of-Words (CBOW) and skip grams. We will not get into the algorithms other than to say that they generally look at a window of words for each target word to provide context and in turn meaning for words. The approach was developed by Tomas Mikolov, formerly at Google and currently at Facebook.

Word2Vec models require a lot of text, e.g. the entire Wikipedia corpus. Nevertheless, we will demonstrate the principles using a small in-memory example of text. Gensim provides the `Word2Vec` class for working with a Word2Vec model. Learning a word embedding from text involves loading and organizing the text into sentences and providing them to the constructor of a new `Word2Vec()` instance. For example:

```
sentences = ...
model = Word2Vec(sentences)
```

Listing 12.2: Example of creating a Word2Vec model.

Specifically, each sentence must be tokenized, meaning divided into words and prepared (e.g. perhaps pre-filtered and perhaps converted to a preferred case). The sentences could be text loaded into memory, or an iterator that progressively loads text, required for very large text corpora. There are many parameters on this constructor; a few noteworthy arguments you may wish to configure are:

- `size`: (default 100) The number of dimensions of the embedding, e.g. the length of the dense vector to represent each token (word).
- `window`: (default 5) The maximum distance between a target word and words around the target word.
- `min_count`: (default 5) The minimum count of words to consider when training the model; words with an occurrence less than this count will be ignored.
- `workers`: (default 3) The number of threads to use while training.
- `sg`: (default 0 or CBOW) The training algorithm, either CBOW (0) or skip gram (1).

The defaults are often good enough when just getting started. If you have a lot of cores, as most modern computers do, I strongly encourage you to increase workers to match the number of cores (e.g. 8). After the model is trained, it is accessible via the `wv` attribute. This is the actual word vector model in which queries can be made. For example, you can print the learned vocabulary of tokens (words) as follows:

```
words = list(model.wv.vocab)
print(words)
```

Listing 12.3: Example of summarizing the words in the model vocabulary.

You can review the embedded vector for a specific token as follows:

```
print(model['word'])
```

Listing 12.4: Example of printing the embedding for a specific word.

Finally, a trained model can then be saved to file by calling the `save_word2vec_format()` function on the word vector model. By default, the model is saved in a binary format to save space. For example:

```
model.wv.save_word2vec_format('model.bin')
```

Listing 12.5: Example of saving a word embedding.

When getting started, you can save the learned model in ASCII format and review the contents. You can do this by setting `binary=False` when calling the `save_word2vec_format()` function, for example:

```
model.wv.save_word2vec_format('model.txt', binary=False)
```

Listing 12.6: Example of saving a word embedding in ASCII format.

The saved model can then be loaded again by calling the `Word2Vec.load()` function. For example:

```
model = Word2Vec.load('model.bin')
```

Listing 12.7: Example of loading a saved word embedding.

We can tie all of this together with a worked example. Rather than loading a large text document or corpus from file, we will work with a small, in-memory list of pre-tokenized sentences. The model is trained and the minimum count for words is set to 1 so that no words are ignored. After the model is learned, we summarize, print the vocabulary, then print a single vector for the word "*sentence*". Finally, the model is saved to a file in binary format, loaded, and then summarized.

```
from gensim.models import Word2Vec
# define training data
sentences = [['this', 'is', 'the', 'first', 'sentence', 'for', 'word2vec'],
      ['this', 'is', 'the', 'second', 'sentence'],
      ['yet', 'another', 'sentence'],
      ['one', 'more', 'sentence'],
      ['and', 'the', 'final', 'sentence']]
# train model
model = Word2Vec(sentences, min_count=1)
# summarize the loaded model
print(model)
# summarize vocabulary
words = list(model.wv.vocab)
print(words)
# access vector for one word
print(model['sentence'])
# save model
model.save('model.bin')
# load model
new_model = Word2Vec.load('model.bin')
print(new_model)
```

Listing 12.8: Example demonstrating the Word2Vec model in Gensim.

Running the example prints the following output.

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

```
Word2Vec(vocab=14, size=100, alpha=0.025)
['second', 'sentence', 'and', 'this', 'final', 'word2vec', 'for', 'another', 'one',
    'first', 'more', 'the', 'yet', 'is']
[ -4.61881841e-03 -4.88735968e-03 -3.19508743e-03 4.08568839e-03
  -3.38211656e-03 1.93076557e-03 3.90265253e-03 -1.04349572e-03
```

```
  4.14286414e-03  1.55219622e-03  3.85653134e-03  2.22428422e-03
 -3.52565176e-03  2.82056746e-03 -2.11121864e-03 -1.38054823e-03
 -1.12888147e-03 -2.87318649e-03 -7.99703528e-04  3.67874932e-03
  2.68940022e-03  6.31021452e-04 -4.36326629e-03  2.38655557e-04
 -1.94210222e-03  4.87691024e-03 -4.04118607e-03 -3.17813386e-03
  4.94802603e-03  3.43150692e-03 -1.44031656e-03  4.25637932e-03
 -1.15106850e-04 -3.73274647e-03  2.50349124e-03  4.28692997e-03
 -3.57313151e-03 -7.24728088e-05 -3.46099050e-03 -3.39612062e-03
  3.54845310e-03  1.56780297e-03  4.58260969e-04  2.52689526e-04
  3.06256465e-03  2.37558200e-03  4.06933809e-03  2.94650183e-03
 -2.96231941e-03 -4.47433954e-03  2.89590308e-03 -2.16034567e-03
 -2.58548348e-03 -2.06163677e-04  1.72605237e-03 -2.27384618e-04
 -3.70194600e-03  2.11557443e-03  2.03793868e-03  3.09839356e-03
 -4.71800892e-03  2.32995977e-03 -6.70911541e-05  1.39375112e-03
 -3.84263694e-03 -1.03898917e-03  4.13251948e-03  1.06330717e-03
  1.38514000e-03 -1.18144893e-03 -2.60811858e-03  1.54952740e-03
  2.49916781e-03 -1.95435272e-03  8.86975031e-05  1.89820060e-03
 -3.41996481e-03 -4.08187555e-03  5.88635216e-04  4.13103355e-03
 -3.25899688e-03  1.02130906e-03 -3.61028523e-03  4.17646067e-03
  4.65870230e-03  3.64110398e-04  4.95479070e-03 -1.29743712e-03
 -5.03367570e-04 -2.52546836e-03  3.31060472e-03 -3.12870182e-03
 -1.14580349e-03 -4.34387522e-03 -4.62882593e-03  3.19007039e-03
  2.88707414e-03  1.62976081e-04 -6.05802808e-04 -1.06368808e-03]
Word2Vec(vocab=14, size=100, alpha=0.025)
```

Listing 12.9: Example output of the Word2Vec model in Gensim.

You can see that with a little work to prepare your text document, you can create your own word embedding very easily with Gensim.

12.5 Visualize Word Embedding

After you learn word embedding for your text data, it can be nice to explore it with visualization. You can use classical projection methods to reduce the high-dimensional word vectors to two-dimensional plots and plot them on a graph. The visualizations can provide a qualitative diagnostic for your learned model. We can retrieve all of the vectors from a trained model as follows:

```
X = model[model.wv.vocab]
```

Listing 12.10: Access the model vocabulary.

We can then train a projection method on the vectors, such as those methods offered in scikit-learn, then use Matplotlib to plot the projection as a scatter plot. Let's look at an example with Principal Component Analysis or PCA.

12.5.1 Plot Word Vectors Using PCA

We can create a 2-dimensional PCA model of the word vectors using the scikit-learn `PCA` class as follows.

```
pca = PCA(n_components=2)
result = pca.fit_transform(X)
```

Listing 12.11: Example of fitting a 2D PCA model to the word vectors.

The resulting projection can be plotted using Matplotlib as follows, pulling out the two dimensions as `x` and `y` coordinates.

```
pyplot.scatter(result[:, 0], result[:, 1])
```

Listing 12.12: Example of plotting a scatter plot of the PCA vectors.

We can go one step further and annotate the points on the graph with the words themselves. A crude version without any nice offsets looks as follows.

```
words = list(model.wv.vocab)
for i, word in enumerate(words):
  pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
```

Listing 12.13: Example of plotting words on the scatter plot.

Putting this all together with the model from the previous section, the complete example is listed below.

```
from gensim.models import Word2Vec
from sklearn.decomposition import PCA
from matplotlib import pyplot
# define training data
sentences = [['this', 'is', 'the', 'first', 'sentence', 'for', 'word2vec'],
      ['this', 'is', 'the', 'second', 'sentence'],
      ['yet', 'another', 'sentence'],
      ['one', 'more', 'sentence'],
      ['and', 'the', 'final', 'sentence']]
# train model
model = Word2Vec(sentences, min_count=1)
# fit a 2d PCA model to the vectors
X = model[model.wv.vocab]
pca = PCA(n_components=2)
result = pca.fit_transform(X)
# create a scatter plot of the projection
pyplot.scatter(result[:, 0], result[:, 1])
words = list(model.wv.vocab)
for i, word in enumerate(words):
  pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
pyplot.show()
```

Listing 12.14: Example demonstrating how to plot word vectors.

Running the example creates a scatter plot with the dots annotated with the words. It is hard to pull much meaning out of the graph given such a tiny corpus was used to fit the model.

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

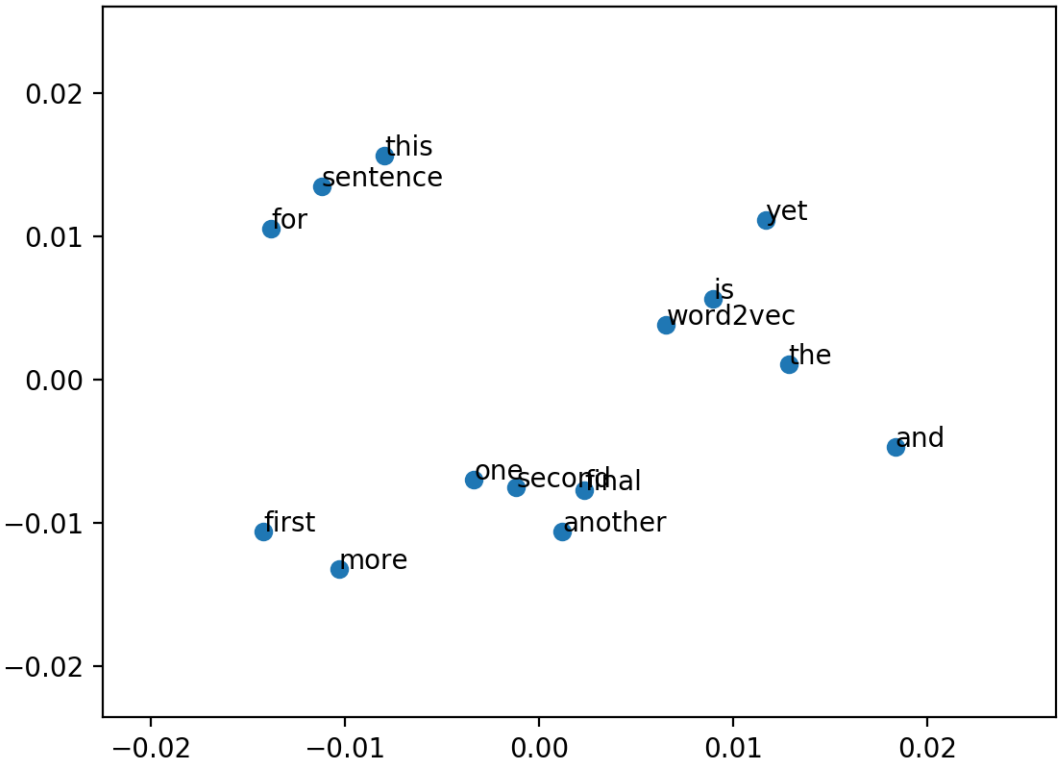

Figure 12.1: Scatter Plot of PCA Projection of Word2Vec Model

12.6 Load Google's Word2Vec Embedding

Training your own word vectors may be the best approach for a given NLP problem. But it can take a long time, a fast computer with a lot of RAM and disk space, and perhaps some expertise in finessing the input data and training algorithm. An alternative is to simply use an existing pre-trained word embedding. Along with the paper and code for Word2Vec, Google also published a pre-trained Word2Vec model on the Word2Vec Google Code Project.

A pre-trained model is nothing more than a file containing tokens and their associated word vectors. The pre-trained Google Word2Vec model was trained on Google news data (about 100 billion words); it contains 3 million words and phrases and was fit using 300-dimensional word vectors. It is a 1.53 Gigabyte file. You can download it from here:

- `GoogleNews-vectors-negative300.bin.gz`.
  https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing

Unzipped, the binary file (`GoogleNews-vectors-negative300.bin`) is 3.4 Gigabytes. The Gensim library provides tools to load this file. Specifically, you can call the `KeyedVectors.load_word2vec_format()` function to load this model into memory, for example:

```
from gensim.models import KeyedVectors
```

```
filename = 'GoogleNews-vectors-negative300.bin'
model = KeyedVectors.load_word2vec_format(filename, binary=True)
```

Listing 12.15: Example of loading the Google word vectors in Gensim.

Note, this example may require a workstation with 8 or more Gigabytes of RAM to execute. On my modern workstation, it takes about 43 seconds to load. Another interesting thing that you can do is do a little linear algebra arithmetic with words. For example, a popular example described in lectures and introduction papers is:

```
queen = (king - man) + woman
```

Listing 12.16: Example of arithmetic of word vectors.

That is the word queen is the closest word given the subtraction of the notion of man from king and adding the word woman. The *man-ness* in king is replaced with *woman-ness* to give us queen. A very cool concept. Gensim provides an interface for performing these types of operations in the `most_similar()` function on the trained or loaded model. For example:

```
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result)
```

Listing 12.17: Example of arithmetic of word vectors in Gensim.

We can put all of this together as follows.

```
from gensim.models import KeyedVectors
# load the google word2vec model
filename = 'GoogleNews-vectors-negative300.bin'
model = KeyedVectors.load_word2vec_format(filename, binary=True)
# calculate: (king - man) + woman = ?
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result)
```

Listing 12.18: Example demonstrating arithmetic with Google word vectors.

Running the example loads the Google pre-trained Word2Vec model and then calculates the `(king - man) + woman = ?` operation on the word vectors for those words. The answer, as we would expect, is `queen`.

```
[('queen', 0.7118192315101624)]
```

Listing 12.19: Output of arithmetic with Google word vectors.

See some of the articles in the further reading section for more interesting arithmetic examples that you can explore.

12.7 Load Stanford's GloVe Embedding

Stanford researchers also have their own word embedding algorithm like Word2Vec called Global Vectors for Word Representation, or GloVe for short. I won't get into the details of the differences between Word2Vec and GloVe here, but generally, NLP practitioners seem to prefer GloVe at the moment based on results.

Like Word2Vec, the GloVe researchers also provide pre-trained word vectors, in this case, a great selection to choose from. You can download the GloVe pre-trained word vectors and load

them easily with Gensim. The first step is to convert the GloVe file format to the Word2Vec file format. The only difference is the addition of a small header line. This can be done by calling the `glove2word2vec()` function. For example (note, this example is just a demonstration with a mock input filename):

```
from gensim.scripts.glove2word2vec import glove2word2vec
glove_input_file = 'glove.txt'
word2vec_output_file = 'word2vec.txt'
glove2word2vec(glove_input_file, word2vec_output_file)
```

Listing 12.20: Example of converting a file from GloVe to Word2Vec format.

Once converted, the file can be loaded just like Word2Vec file above. Let's make this concrete with an example. You can download the smallest GloVe pre-trained model from the GloVe website. It an 822 Megabyte zip file with 4 different models (50, 100, 200 and 300-dimensional vectors) trained on Wikipedia data with 6 billion tokens and a 400,000 word vocabulary. The direct download link is here:

- `glove.6B.zip`.
  http://nlp.stanford.edu/data/glove.6B.zip

Working with the 100-dimensional version of the model, we can convert the file to Word2Vec format as follows:

```
from gensim.scripts.glove2word2vec import glove2word2vec
glove_input_file = 'glove.6B.100d.txt'
word2vec_output_file = 'glove.6B.100d.txt.word2vec'
glove2word2vec(glove_input_file, word2vec_output_file)
```

Listing 12.21: Example of converting a specific GloVe file to Word2Vec format.

You now have a copy of the GloVe model in Word2Vec format with the filename `glove.6B.100d.txt.word2vec`. Now we can load it and perform the same `(king - man) + woman = ?` test as in the previous section. The complete code listing is provided below. Note that the converted file is ASCII format, not binary, so we set `binary=False` when loading.

```
from gensim.models import KeyedVectors
# load the Stanford GloVe model
filename = 'glove.6B.100d.txt.word2vec'
model = KeyedVectors.load_word2vec_format(filename, binary=False)
# calculate: (king - man) + woman = ?
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result)
```

Listing 12.22: Example of arithmetic with converted GloVe word vectors.

Pulling all of this together, the complete example is listed below.

```
from gensim.models import KeyedVectors
from gensim.scripts.glove2word2vec import glove2word2vec

# convert glove to word2vec format
glove_input_file = 'glove.6B.100d.txt'
word2vec_output_file = 'glove.6B.100d.txt.word2vec'
glove2word2vec(glove_input_file, word2vec_output_file)
```

```
# load the converted model
filename = 'glove.6B.100d.txt.word2vec'
model = KeyedVectors.load_word2vec_format(filename, binary=False)
# calculate: (king - man) + woman = ?
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result)
```

Listing 12.23: Example demonstrating how to load and use GloVe word embeddings.

Running the example prints the same result of `queen`.

```
[('queen', 0.7698540687561035)]
```

Listing 12.24: Example output of arithmetic with converted GloVe word vectors.

12.8 Further Reading

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

12.8.1 Word Embeddings

- Word Embedding on Wikipedia.
  https://en.wikipedia.org/wiki/Word2vec
- Word2Vec on Wikipedia.
  https://en.wikipedia.org/wiki/Word2vec
- Google Word2Vec project.
  https://code.google.com/archive/p/word2vec/
- Stanford GloVe project.
  https://nlp.stanford.edu/projects/glove/

12.8.2 Gensim

- Gensim Python Library.
  https://radimrehurek.com/gensim/index.html
- Gensim Installation Instructions.
  https://radimrehurek.com/gensim/install.html
- models.word2vec Gensim API.
  https://radimrehurek.com/gensim/models/keyedvectors.html
- models.keyedvectors Gensim API.
  https://radimrehurek.com/gensim/models/keyedvectors.html
- scripts.glove2word2vec Gensim API.
  https://radimrehurek.com/gensim/scripts/glove2word2vec.html

12.8.3 Articles

- *Messing Around With Word2Vec*, 2016.
  https://quomodocumque.wordpress.com/2016/01/15/messing-around-with-word2vec/
- *Vector Space Models for the Digital Humanities*, 2015.
  http://bookworm.benschmidt.org/posts/2015-10-25-Word-Embeddings.html
- *Gensim Word2Vec Tutorial*, 2014.
  https://rare-technologies.com/word2vec-tutorial/

12.9 Summary

In this tutorial, you discovered how to develop and load word embedding layers in Python using Gensim. Specifically, you learned:

- How to train your own Word2Vec word embedding model on text data.
- How to visualize a trained word embedding model using Principal Component Analysis.
- How to load pre-trained Word2Vec and GloVe word embedding models from Google and Stanford.

12.9.1 Next

In the next chapter, you will discover how you can develop a neural network model with a learned word embedding.

Chapter 13

How to Learn and Load Word Embeddings in Keras

Word embeddings provide a dense representation of words and their relative meanings. They are an improvement over sparse representations used in simpler bag of word model representations. Word embeddings can be learned from text data and reused among projects. They can also be learned as part of fitting a neural network on text data. In this tutorial, you will discover how to use word embeddings for deep learning in Python with Keras. After completing this tutorial, you will know:

- About word embeddings and that Keras supports word embeddings via the `Embedding` layer.
- How to learn a word embedding while fitting a neural network.
- How to use a pre-trained word embedding in a neural network.

Let's get started.

13.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Word Embedding
2. Keras Embedding Layer
3. Example of Learning an Embedding
4. Example of Using Pre-Trained GloVe Embedding
5. Tips for Cleaning Text for Word Embedding

13.2 Word Embedding

A word embedding is a class of approaches for representing words and documents using a dense vector representation. It is an improvement over more the traditional bag-of-word model encoding schemes where large sparse vectors were used to represent each word or to score each word within a vector to represent an entire vocabulary. These representations were sparse because the vocabularies were vast and a given word or document would be represented by a large vector comprised mostly of zero values.

Instead, in an embedding, words are represented by dense vectors where a vector represents the projection of the word into a continuous vector space. The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. The position of a word in the learned vector space is referred to as its embedding. Two popular examples of methods of learning word embeddings from text include:

- Word2Vec.
- GloVe.

In addition to these carefully designed methods, a word embedding can be learned as part of a deep learning model. This can be a slower approach, but tailors the model to a specific training dataset.

13.3 Keras Embedding Layer

Keras offers an `Embedding` layer that can be used for neural networks on text data. It requires that the input data be integer encoded, so that each word is represented by a unique integer. This data preparation step can be performed using the `Tokenizer` API also provided with Keras.

The `Embedding` layer is initialized with random weights and will learn an embedding for all of the words in the training dataset. It is a flexible layer that can be used in a variety of ways, such as:

- It can be used alone to learn a word embedding that can be saved and used in another model later.
- It can be used as part of a deep learning model where the embedding is learned along with the model itself.
- It can be used to load a pre-trained word embedding model, a type of transfer learning.

The `Embedding` layer is defined as the first hidden layer of a network. It must specify 3 arguments:

- `input_dim`: This is the size of the vocabulary in the text data. For example, if your data is integer encoded to values between 0-10, then the size of the vocabulary would be 11 words.

- `output_dim`: This is the size of the vector space in which words will be embedded. It defines the size of the output vectors from this layer for each word. For example, it could be 32 or 100 or even larger. Test different values for your problem.
- `input_length`: This is the length of input sequences, as you would define for any input layer of a Keras model. For example, if all of your input documents are comprised of 1000 words, this would be 1000.

For example, below we define an `Embedding` layer with a vocabulary of 200 (e.g. integer encoded words from 0 to 199, inclusive), a vector space of 32 dimensions in which words will be embedded, and input documents that have 50 words each.

```
e = Embedding(200, 32, input_length=50)
```

Listing 13.1: Example of creating a word embedding layer.

The `Embedding` layer has weights that are learned. If you save your model to file, this will include weights for the `Embedding` layer. The output of the `Embedding` layer is a 2D vector with one embedding for each word in the input sequence of words (input document). If you wish to connect a `Dense` layer directly to an `Embedding` layer, you must first flatten the 2D output matrix to a 1D vector using the `Flatten` layer. Now, let's see how we can use an `Embedding` layer in practice.

13.4 Example of Learning an Embedding

In this section, we will look at how we can learn a word embedding while fitting a neural network on a text classification problem. We will define a small problem where we have 10 text documents, each with a comment about a piece of work a student submitted. Each text document is classified as positive `1` or negative `0`. This is a simple sentiment analysis problem. First, we will define the documents and their class labels.

```
# define documents
docs = ['Well done!',
    'Good work',
    'Great effort',
    'nice work',
    'Excellent!',
    'Weak',
    'Poor effort!',
    'not good',
    'poor work',
    'Could have done better.']
# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
```

Listing 13.2: Example of a small contrived classification problem.

Next, we can integer encode each document. This means that as input the `Embedding` layer will have sequences of integers. We could experiment with other more sophisticated bag of word model encoding like counts or TF-IDF. Keras provides the `one_hot()` function that creates a hash of each word as an efficient integer encoding. We will estimate the vocabulary size of 50, which is much larger than needed to reduce the probability of collisions from the hash function.

```
# integer encode the documents
vocab_size = 50
encoded_docs = [one_hot(d, vocab_size) for d in docs]
print(encoded_docs)
```

Listing 13.3: Integer encode the text.

The sequences have different lengths and Keras prefers inputs to be vectorized and all inputs to have the same length. We will pad all input sequences to have the length of 4. Again, we can do this with a built in Keras function, in this case the `pad_sequences()` function.

```
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
```

Listing 13.4: Pad the encoded text.

We are now ready to define our `Embedding` layer as part of our neural network model. The `Embedding` layer has a vocabulary of 50 and an input length of 4. We will choose a small embedding space of 8 dimensions. The model is a simple binary classification model. Importantly, the output from the `Embedding` layer will be 4 vectors of 8 dimensions each, one for each word. We flatten this to a one 32-element vector to pass on to the `Dense` output layer.

```
# define the model
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# summarize the model
model.summary()
```

Listing 13.5: Define a simple model with an `Embedding` input.

Finally, we can fit and evaluate the classification model.

```
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

Listing 13.6: Fit the defined model and print model accuracy.

The complete code listing is provided below.

```
from keras.preprocessing.text import one_hot
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
# define documents
docs = ['Well done!',
    'Good work',
```

```
    'Great effort',
    'nice work',
    'Excellent!',
    'Weak',
    'Poor effort!',
    'not good',
    'poor work',
    'Could have done better.']
# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
# integer encode the documents
vocab_size = 50
encoded_docs = [one_hot(d, vocab_size) for d in docs]
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
# define the model
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# summarize the model
model.summary()
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

Listing 13.7: Example of fitting and evaluating a Keras model with an `Embedding` input layer.

Running the example first prints the integer encoded documents.

```
[[6, 16], [42, 24], [2, 17], [42, 24], [18], [17], [22, 17], [27, 42], [22, 24], [49, 46,
    16, 34]]
```

Listing 13.8: Example output of the encoded documents.

Then the padded versions of each document are printed, making them all uniform length.

```
[[ 6 16  0  0]
 [42 24  0  0]
 [ 2 17  0  0]
 [42 24  0  0]
 [18  0  0  0]
 [17  0  0  0]
 [22 17  0  0]
 [27 42  0  0]
 [22 24  0  0]
 [49 46 16 34]]
```

Listing 13.9: Example output of the padded encoded documents.

After the network is defined, a summary of the layers is printed. We can see that as expected, the output of the `Embedding` layer is a 4 x 8 matrix and this is squashed to a 32-element vector by the `Flatten` layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 4, 8)              400
_________________________________________________________________
flatten_1 (Flatten)          (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 433
Trainable params: 433
Non-trainable params: 0
_________________________________________________________________
```

Listing 13.10: Example output of the model summary.

Finally, the accuracy of the trained model is printed, showing that it learned the training dataset perfectly (which is not surprising).

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

```
Accuracy: 100.000000
```

Listing 13.11: Example output of the model accuracy.

You could save the learned weights from the `Embedding` layer to file for later use in other models. You could also use this model generally to classify other documents that have the same kind vocabulary seen in the test dataset. Next, let's look at loading a pre-trained word embedding in Keras.

13.5 Example of Using Pre-Trained GloVe Embedding

The Keras `Embedding` layer can also use a word embedding learned elsewhere. It is common in the field of Natural Language Processing to learn, save, and make freely available word embeddings. For example, the researchers behind GloVe method provide a suite of pre-trained word embeddings on their website released under a public domain license.

The smallest package of embeddings is 822 Megabytes, called `glove.6B.zip`. It was trained on a dataset of one billion tokens (words) with a vocabulary of 400 thousand words. There are a few different embedding vector sizes, including 50, 100, 200 and 300 dimensions. You can download this collection of embeddings and we can seed the Keras `Embedding` layer with weights from the pre-trained embedding for the words in your training dataset.

This example is inspired by an example in the Keras project: `pretrained_word_embeddings.py`. After downloading and unzipping, you will see a few files, one of which is `glove.6B.100d.txt`, which contains a 100-dimensional version of the embedding. If you peek inside the file, you will see a token (word) followed by the weights (100 numbers) on each line. For example, below are the first line of the embedding ASCII text file showing the embedding for *the*.

```
the -0.038194 -0.24487 0.72812 -0.39961 0.083172 0.043953 -0.39141 0.3344 -0.57545 0.087459
    0.28787 -0.06731 0.30906 -0.26384 -0.13231 -0.20757 0.33395 -0.33848 -0.31743 -0.48336
    0.1464 -0.37304 0.34577 0.052041 0.44946 -0.46971 0.02628 -0.54155 -0.15518 -0.14107
    -0.039722 0.28277 0.14393 0.23464 -0.31021 0.086173 0.20397 0.52624 0.17164 -0.082378
    -0.71787 -0.41531 0.20335 -0.12763 0.41367 0.55187 0.57908 -0.33477 -0.36559 -0.54857
    -0.062892 0.26584 0.30205 0.99775 -0.80481 -3.0243 0.01254 -0.36942 2.2167 0.72201
    -0.24978 0.92136 0.034514 0.46745 1.1079 -0.19358 -0.074575 0.23353 -0.052062 -0.22044
    0.057162 -0.15806 -0.30798 -0.41625 0.37972 0.15006 -0.53212 -0.2055 -1.2526 0.071624
    0.70565 0.49744 -0.42063 0.26148 -1.538 -0.30223 -0.073438 -0.28312 0.37104 -0.25217
    0.016215 -0.017099 -0.38984 0.87424 -0.72569 -0.51058 -0.52028 -0.1459 0.8278 0.27062
```

Listing 13.12: Example GloVe word vector for the word 'the'.

As in the previous section, the first step is to define the examples, encode them as integers, then pad the sequences to be the same length. In this case, we need to be able to map words to integers as well as integers to words. Keras provides a `Tokenizer` class that can be fit on the training data, can convert text to sequences consistently by calling the `texts_to_sequences()` method on the `Tokenizer` class, and provides access to the dictionary mapping of words to integers in a `word_index` attribute.

```
# define documents
docs = ['Well done!',
    'Good work',
    'Great effort',
    'nice work',
    'Excellent!',
    'Weak',
    'Poor effort!',
    'not good',
    'poor work',
    'Could have done better.']
# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
# prepare tokenizer
t = Tokenizer()
t.fit_on_texts(docs)
vocab_size = len(t.word_index) + 1
# integer encode the documents
encoded_docs = t.texts_to_sequences(docs)
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
```

Listing 13.13: Define encode and pad sample documents.

Next, we need to load the entire GloVe word embedding file into memory as a dictionary of word to embedding array.

```
# load the whole embedding into memory
embeddings_index = dict()
f = open('glove.6B.100d.txt')
for line in f:
  values = line.split()
  word = values[0]
```

```
  coefs = asarray(values[1:], dtype='float32')
  embeddings_index[word] = coefs
f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
```

Listing 13.14: Load the GloVe word embedding into memory.

This is pretty slow. It might be better to filter the embedding for the unique words in your training data. Next, we need to create a matrix of one embedding for each word in the training dataset. We can do that by enumerating all unique words in the `Tokenizer.word_index` and locating the embedding weight vector from the loaded GloVe embedding. The result is a matrix of weights only for words we will see during training.

```
# create a weight matrix for words in training docs
embedding_matrix = zeros((vocab_size, 100))
for word, i in t.word_index.items():
  embedding_vector = embeddings_index.get(word)
  if embedding_vector is not None:
    embedding_matrix[i] = embedding_vector
```

Listing 13.15: Covert the word embedding into a weight matrix.

Now we can define our model, fit, and evaluate it as before. The key difference is that the `Embedding` layer can be seeded with the GloVe word embedding weights. We chose the 100-dimensional version, therefore the `Embedding` layer must be defined with `output_dim` set to 100. Finally, we do not want to update the learned word weights in this model, therefore we will set the trainable attribute for the model to be `False`.

```
e = Embedding(vocab_size, 100, weights=[embedding_matrix], input_length=4, trainable=False)
```

Listing 13.16: Create an `Embedding` layer with the pre-loaded weights.

The complete worked example is listed below.

```
from numpy import asarray
from numpy import zeros
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Embedding
# define documents
docs = ['Well done!',
    'Good work',
    'Great effort',
    'nice work',
    'Excellent!',
    'Weak',
    'Poor effort!',
    'not good',
    'poor work',
    'Could have done better.']
# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
# prepare tokenizer
t = Tokenizer()
```

```
t.fit_on_texts(docs)
vocab_size = len(t.word_index) + 1
# integer encode the documents
encoded_docs = t.texts_to_sequences(docs)
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
# load the whole embedding into memory
embeddings_index = dict()
f = open('glove.6B.100d.txt', mode='rt', encoding='utf-8')
for line in f:
  values = line.split()
  word = values[0]
  coefs = asarray(values[1:], dtype='float32')
  embeddings_index[word] = coefs
f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
# create a weight matrix for words in training docs
embedding_matrix = zeros((vocab_size, 100))
for word, i in t.word_index.items():
  embedding_vector = embeddings_index.get(word)
  if embedding_vector is not None:
    embedding_matrix[i] = embedding_vector
# define model
model = Sequential()
e = Embedding(vocab_size, 100, weights=[embedding_matrix], input_length=4, trainable=False)
model.add(e)
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# summarize the model
model.summary()
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

Listing 13.17: Example loading pre-trained GloVe weights into an `Embedding` input layer.

Running the example may take a bit longer, but then demonstrates that it is just as capable of fitting this simple problem.

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

```
...
Accuracy: 100.000000
```

Listing 13.18: Example output of loading pre-trained GloVe weights into an `Embedding` input layer.

In practice, I would encourage you to experiment with learning a word embedding using a pre-trained embedding that is fixed and trying to perform learning on top of a pre-trained embedding. See what works best for your specific problem.

13.6 Tips for Cleaning Text for Word Embedding

Recently, the field of natural language processing has been moving away from bag-of-word models and word encoding toward word embeddings. The benefit of word embeddings is that they encode each word into a dense vector that captures something about its relative meaning within the training text. This means that variations of words like case, spelling, punctuation, and so on will automatically be learned to be similar in the embedding space. In turn, this can mean that the amount of cleaning required from your text may be less and perhaps quite different to classical text cleaning. For example, it may no-longer make sense to stem words or remove punctuation for contractions.

Tomas Mikolov is one of the developers of Word2Vec, a popular word embedding method. He suggests only very minimal text cleaning is required when learning a word embedding model. Below is his response when pressed with the question about how to best prepare text data for Word2Vec.

> There is no universal answer. It all depends on what you plan to use the vectors for. In my experience, it is usually good to disconnect (or remove) punctuation from words, and sometimes also convert all characters to lowercase. One can also replace all numbers (possibly greater than some constant) with some single token such as .
>
> All these pre-processing steps aim to reduce the vocabulary size without removing any important content (which in some cases may not be true when you lowercase certain words, ie. 'Bush' is different than 'bush', while 'Another' has usually the same sense as 'another'). The smaller the vocabulary is, the lower is the memory complexity, and the more robustly are the parameters for the words estimated. You also have to pre-process the test data in the same way.
>
> [...]
>
> In short, you will understand all this much better if you will run experiments.
>
> — Tomas Mikolov, *word2vec-toolkit: google groups thread.*, 2015.
> https://goo.gl/KtDGst

13.7 Further Reading

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

- Word Embedding on Wikipedia.
  https://en.wikipedia.org/wiki/Word_embedding
- Keras Embedding Layer API.
  https://keras.io/layers/embeddings/#embedding

- *Using pre-trained word embeddings in a Keras model*, 2016.
  https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html
- Example of using a pre-trained GloVe Embedding in Keras.
  https://github.com/fchollet/keras/blob/master/examples/pretrained_word_embeddings.py
- GloVe Embedding.
  https://nlp.stanford.edu/projects/glove/
- *An overview of word embeddings and their connection to distributional semantic models*, 2016.
  http://blog.aylien.com/overview-word-embeddings-history-word2vec-cbow-glove/
- *Deep Learning, NLP, and Representations*, 2014.
  http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/

13.8 Summary

In this tutorial, you discovered how to use word embeddings for deep learning in Python with Keras. Specifically, you learned:

- About word embeddings and that Keras supports word embeddings via the `Embedding` layer.
- How to learn a word embedding while fitting a neural network.
- How to use a pre-trained word embedding in a neural network.

13.8.1 Next

This is the end of the part on word embeddings. In the next part you will discover neural text classification.

Part VI

Text Classification

Chapter 14

Neural Models for Document Classification

Text classification describes a general class of problems such as predicting the sentiment of tweets and movie reviews, as well as classifying email as spam or not. Deep learning methods are proving very good at text classification, achieving state-of-the-art results on a suite of standard academic benchmark problems. In this chapter, you will discover some best practices to consider when developing deep learning models for text classification. After reading this chapter, you will know:

- The general combination of deep learning methods to consider when starting your text classification problems.
- The first architecture to try with specific advice on how to configure hyperparameters.
- That deeper networks may be the future of the field in terms of flexibility and capability.

Let's get started.

14.1 Overview

This tutorial is divided into the following parts:

1. Word Embeddings + CNN = Text Classification
2. Use a Single Layer CNN Architecture
3. Dial in CNN Hyperparameters
4. Consider Character-Level CNNs
5. Consider Deeper CNNs for Classification

14.2 Word Embeddings + CNN = Text Classification

The modus operandi for text classification involves the use of a word embedding for representing words and a Convolutional Neural Network (CNN) for learning how to discriminate documents on classification problems. Yoav Goldberg, in his primer on deep learning for natural language processing, comments that neural networks in general offer better performance than classical linear classifiers, especially when used with pre-trained word embeddings.

> The non-linearity of the network, as well as the ability to easily integrate pre-trained word embeddings, often lead to superior classification accuracy.
>
> — *A Primer on Neural Network Models for Natural Language Processing*, 2015.

He also comments that convolutional neural networks are effective at document classification, namely because they are able to pick out salient features (e.g. tokens or sequences of tokens) in a way that is invariant to their position within the input sequences.

> Networks with convolutional and pooling layers are useful for classification tasks in which we expect to find strong local clues regarding class membership, but these clues can appear in different places in the input. [...] We would like to learn that certain sequences of words are good indicators of the topic, and do not necessarily care where they appear in the document. Convolutional and pooling layers allow the model to learn to find such local indicators, regardless of their position.
>
> — *A Primer on Neural Network Models for Natural Language Processing*, 2015.

The architecture is therefore comprised of three key pieces:

- **Word Embedding**: A distributed representation of words where different words that have a similar meaning (based on their usage) also have a similar representation.
- **Convolutional Model**: A feature extraction model that learns to extract salient features from documents represented using a word embedding.
- **Fully Connected Model**: The interpretation of extracted features in terms of a predictive output.

Yoav Goldberg highlights the CNNs role as a feature extractor model in his book:

> ... the CNN is in essence a feature-extracting architecture. It does not constitute a standalone, useful network on its own, but rather is meant to be integrated into a larger network, and to be trained to work in tandem with it in order to produce an end result. The CNNs layer's responsibility is to extract meaningful sub-structures that are useful for the overall prediction task at hand.
>
> — Page 152, *Neural Network Methods for Natural Language Processing*, 2017.

The tying together of these three elements is demonstrated in perhaps one of the most widely cited examples of the combination, described in the next section.

14.3 Use a Single Layer CNN Architecture

You can get good results for document classification with a single layer CNN, perhaps with differently sized kernels across the filters to allow grouping of word representations at different scales. Yoon Kim in his study of the use of pre-trained word vectors for classification tasks with Convolutional Neural Networks found that using pre-trained static word vectors does very well. He suggests that pre-trained word embeddings that were trained on very large text corpora, such as the freely available Word2Vec vectors trained on 100 billion tokens from Google news may offer good universal features for use in natural language processing.

> Despite little tuning of hyperparameters, a simple CNN with one layer of convolution performs remarkably well. Our results add to the well-established evidence that unsupervised pre-training of word vectors is an important ingredient in deep learning for NLP
>
> — *Convolutional Neural Networks for Sentence Classification*, 2014.

He also discovered that further task-specific tuning of the word vectors offer a small additional improvement in performance. Kim describes the general approach of using CNN for natural language processing. Sentences are mapped to embedding vectors and are available as a matrix input to the model. Convolutions are performed across the input word-wise using differently sized kernels, such as 2 or 3 words at a time. The resulting feature maps are then processed using a max pooling layer to condense or summarize the extracted features.

The architecture is based on the approach used by Ronan Collobert, et al. in their paper *Natural Language Processing (almost) from Scratch*, 2011. In it, they develop a single end-to-end neural network model with convolutional and pooling layers for use across a range of fundamental natural language processing problems. Kim provides a diagram that helps to see the sampling of the filters using differently sized kernels as different colors (red and yellow).

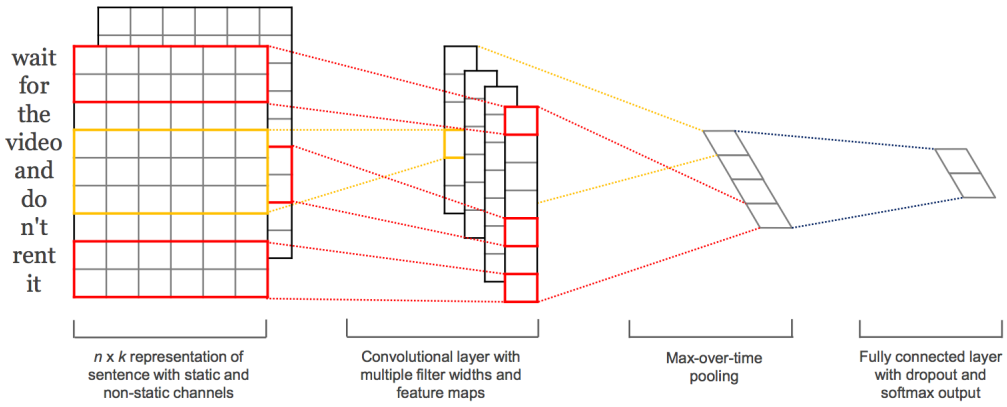

Figure 14.1: An example of a CNN Filter and Polling Architecture for Natural Language Processing. Taken from *Convolutional Neural Networks for Sentence Classification*.

Usefully, he reports his chosen model configuration, discovered via grid search and used across a suite of 7 text classification tasks, summarized as follows:

- Transfer function: rectified linear.
- Kernel sizes: 2, 4, 5.
- Number of filters: 100.
- Dropout rate: 0.5.
- Weight regularization (L2): 3.
- Batch Size: 50.
- Update Rule: Adadelta.

These configurations could be used to inspire a starting point for your own experiments.

14.4 Dial in CNN Hyperparameters

Some hyperparameters matter more than others when tuning a convolutional neural network on your document classification problem. Ye Zhang and Byron Wallace performed a sensitivity analysis into the hyperparameters needed to configure a single layer convolutional neural network for document classification. The study is motivated by their claim that the models are sensitive to their configuration.

> Unfortunately, a downside to CNN-based models - even simple ones - is that they require practitioners to specify the exact model architecture to be used and to set the accompanying hyperparameters. To the uninitiated, making such decisions can seem like something of a black art because there are many free parameters in the model.

— *A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification*, 2015.

Their aim was to provide general configurations that can be used for configuring CNNs on new text classification tasks. They provide a nice depiction of the model architecture and the decision points for configuring the model, reproduced below.

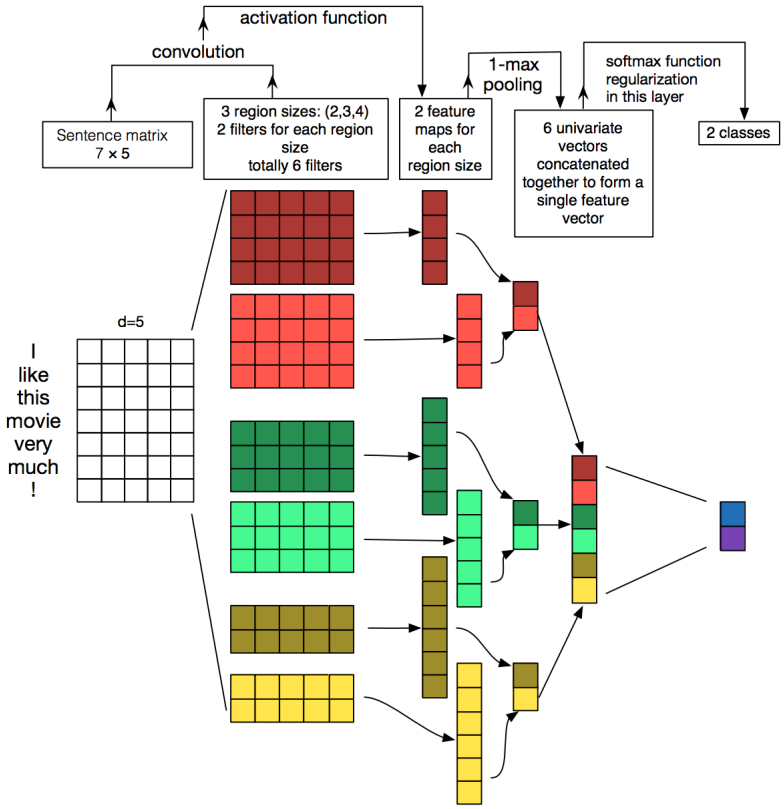

Figure 14.2: Convolutional Neural Network Architecture for Sentence Classification. Taken from *A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification*.

The study makes a number of useful findings that could be used as a starting point for configuring shallow CNN models for text classification. The general findings were as follows:

- The choice of pre-trained Word2Vec and GloVe embeddings differ from problem to problem, and both performed better than using one hot encoded word vectors.
- The size of the kernel is important and should be tuned for each problem.
- The number of feature maps is also important and should be tuned.
- The 1-max pooling generally outperformed other types of pooling.
- Dropout has little effect on the model performance.

They go on to provide more specific heuristics, as follows:

- Use Word2Vec or GloVe word embeddings as a starting point and tune them while fitting the model.
- Grid search across different kernel sizes to find the optimal configuration for your problem, in the range 1-10.

- Search the number of filters from 100-600 and explore a dropout of 0.0-0.5 as part of the same search.
- Explore using tanh, relu, and linear activation functions.

The key caveat is that the findings are based on empirical results on binary text classification problems using single sentences as input.

14.5 Consider Character-Level CNNs

Text documents can be modeled at the character level using convolutional neural networks that are capable of learning the relevant hierarchical structure of words, sentences, paragraphs, and more. Xiang Zhang, et al. use a character-based representation of text as input for a convolutional neural network. The promise of the approach is that all of the labor-intensive effort required to clean and prepare text could be overcome if a CNN can learn to abstract the salient details.

> ... deep ConvNets do not require the knowledge of words, in addition to the conclusion from previous research that ConvNets do not require the knowledge about the syntactic or semantic structure of a language. This simplification of engineering could be crucial for a single system that can work for different languages, since characters always constitute a necessary construct regardless of whether segmentation into words is possible. Working on only characters also has the advantage that abnormal character combinations such as misspellings and emoticons may be naturally learnt.
>
> — *Character-level Convolutional Networks for Text Classification*, 2015.

The model reads in one hot encoded characters in a fixed-sized alphabet. Encoded characters are read in blocks or sequences of 1,024 characters. A stack of 6 convolutional layers with pooling follows, with 3 fully connected layers at the output end of the network in order to make a prediction.

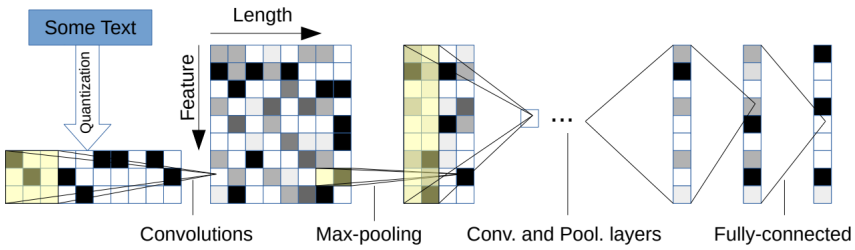

Figure 14.3: Character-based Convolutional Neural Network for Text Classification. Taken from *Character-level Convolutional Networks for Text Classification*.

The model achieves some success, performing better on problems that offer a larger corpus of text.

> ... analysis shows that character-level ConvNet is an effective method. [...] how well our model performs in comparisons depends on many factors, such as dataset size, whether the texts are curated and choice of alphabet.

— *Character-level Convolutional Networks for Text Classification*, 2015.

Results using an extended version of this approach were pushed to the state-of-the-art in a follow-up paper covered in the next section.

14.6 Consider Deeper CNNs for Classification

Better performance can be achieved with very deep convolutional neural networks, although standard and reusable architectures have not been adopted for classification tasks, yet. Alexis Conneau, et al. comment on the relatively shallow networks used for natural language processing and the success of much deeper networks used for computer vision applications. For example, Kim (above) restricted the model to a single convolutional layer.

Other architectures used for natural language reviewed in the paper are limited to 5 and 6 layers. These are contrasted with successful architectures used in computer vision with 19 or even up to 152 layers. They suggest and demonstrate that there are benefits for hierarchical feature learning with very deep convolutional neural network model, called VDCNN.

> ... we propose to use deep architectures of many convolutional layers to approach this goal, using up to 29 layers. The design of our architecture is inspired by recent progress in computer vision [...] The proposed deep convolutional network shows significantly better results than previous ConvNets approach.

— *Very Deep Convolutional Networks for Text Classification*, 2016.

Key to their approach is an embedding of individual characters, rather than a word embedding.

> We present a new architecture (VDCNN) for text processing which operates directly at the character level and uses only small convolutions and pooling operations.

— *Very Deep Convolutional Networks for Text Classification*, 2016.

Results on a suite of 8 large text classification tasks show better performance than more shallow networks. Specifically, state-of-the-art results on all but two of the datasets tested, at the time of writing. Generally, they make some key findings from exploring the deeper architectural approach:

- The very deep architecture worked well on small and large datasets.
- Deeper networks decrease classification error.
- Max-pooling achieves better results than other, more sophisticated types of pooling.
- Generally going deeper degrades accuracy; the shortcut connections used in the architecture are important.

> ... this is the first time that the "benefit of depths" was shown for convolutional neural networks in NLP.

— *Very Deep Convolutional Networks for Text Classification*, 2016.

14.7 Further Reading

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

- *A Primer on Neural Network Models for Natural Language Processing*, 2015.
  https://arxiv.org/abs/1510.00726
- *Convolutional Neural Networks for Sentence Classification*, 2014.
  https://arxiv.org/abs/1103.0398
- *Natural Language Processing (almost) from Scratch*, 2011.
  https://arxiv.org/abs/1103.0398
- *Very Deep Convolutional Networks for Text Classification*, 2016.
  https://arxiv.org/abs/1606.01781
- *Character-level Convolutional Networks for Text Classification*, 2015.
  https://arxiv.org/abs/1509.01626
- *A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification*, 2015.
  https://arxiv.org/abs/1510.03820

14.8 Summary

In this chapter, you discovered some best practices for developing deep learning models for document classification. Specifically, you learned:

- That a key approach is to use word embeddings and convolutional neural networks for text classification.
- That a single layer model can do well on moderate-sized problems, and ideas on how to configure it.
- That deeper models that operate directly on text may be the future of natural language processing.

14.8.1 Next

In the next chapter, you will discover how you can develop a neural text classification model with word embeddings and a convolutional neural network.

Chapter 15

Project: Develop an Embedding + CNN Model for Sentiment Analysis

Word embeddings are a technique for representing text where different words with similar meaning have a similar real-valued vector representation. They are a key breakthrough that has led to great performance of neural network models on a suite of challenging natural language processing problems. In this tutorial, you will discover how to develop word embedding models with convolutional neural networks to classify movie reviews. After completing this tutorial, you will know:

- How to prepare movie review text data for classification with deep learning methods.
- How to develop a neural classification model with word embedding and convolutional layers.
- How to evaluate the developed a neural classification model.

Let's get started.

15.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Movie Review Dataset
2. Data Preparation
3. Train CNN With Embedding Layer
4. Evaluate Model

15.2 Movie Review Dataset

In this tutorial, we will use the Movie Review Dataset. This dataset designed for sentiment analysis was described previously in Chapter 9. You can download the dataset from here:

- Movie Review Polarity Dataset (`review_polarity.tar.gz`, 3MB).
  http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

After unzipping the file, you will have a directory called `txt_sentoken` with two sub-directories containing the text *neg* and *pos* for negative and positive reviews. Reviews are stored one per file with a naming convention *cv000* to *cv999* for each of *neg* and *pos*.

15.3 Data Preparation

Note: The preparation of the movie review dataset was first described in Chapter 9. In this section, we will look at 3 things:

1. Separation of data into training and test sets.
2. Loading and cleaning the data to remove punctuation and numbers.
3. Defining a vocabulary of preferred words.

15.3.1 Split into Train and Test Sets

We are pretending that we are developing a system that can predict the sentiment of a textual movie review as either positive or negative. This means that after the model is developed, we will need to make predictions on new textual reviews. This will require all of the same data preparation to be performed on those new reviews as is performed on the training data for the model. We will ensure that this constraint is built into the evaluation of our models by splitting the training and test datasets prior to any data preparation. This means that any knowledge in the data in the test set that could help us better prepare the data (e.g. the words used) are unavailable in the preparation of data used for training the model.

That being said, we will use the last 100 positive reviews and the last 100 negative reviews as a test set (100 reviews) and the remaining 1,800 reviews as the training dataset. This is a 90% train, 10% split of the data. The split can be imposed easily by using the filenames of the reviews where reviews named 000 to 899 are for training data and reviews named 900 onwards are for test.

15.3.2 Loading and Cleaning Reviews

The text data is already pretty clean; not much preparation is required. Without getting bogged down too much in the details, we will prepare the data using the following way:

- Split tokens on white space.
- Remove all punctuation from words.
- Remove all words that are not purely comprised of alphabetical characters.
- Remove all words that are known stop words.
- Remove all words that have a length ≤ 1 character.

We can put all of these steps into a function called `clean_doc()` that takes as an argument the raw text loaded from a file and returns a list of cleaned tokens. We can also define a function `load_doc()` that loads a document from file ready for use with the `clean_doc()` function. An example of cleaning the first positive review is listed below.

```
from nltk.corpus import stopwords
import string
import re

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load the document
filename = 'txt_sentoken/pos/cv000_29590.txt'
text = load_doc(filename)
tokens = clean_doc(text)
print(tokens)
```

Listing 15.1: Example of cleaning a movie review.

Running the example prints a long list of clean tokens. There are many more cleaning steps we may want to explore and I leave them as further exercises.

```
...
'creepy', 'place', 'even', 'acting', 'hell', 'solid', 'dreamy', 'depp', 'turning',
    'typically', 'strong', 'performance', 'deftly', 'handling', 'british', 'accent',
    'ians', 'holm', 'joe', 'goulds', 'secret', 'richardson', 'dalmatians', 'log', 'great',
    'supporting', 'roles', 'big', 'surprise', 'graham', 'cringed', 'first', 'time',
    'opened', 'mouth', 'imagining', 'attempt', 'irish', 'accent', 'actually', 'wasnt',
    'half', 'bad', 'film', 'however', 'good', 'strong', 'violencegore', 'sexuality',
    'language', 'drug', 'content']
```

Listing 15.2: Example output of cleaning a movie review.

15.3.3 Define a Vocabulary

It is important to define a vocabulary of known words when using a text model. The more words, the larger the representation of documents, therefore it is important to constrain the words to only those believed to be predictive. This is difficult to know beforehand and often it is important to test different hypotheses about how to construct a useful vocabulary. We have already seen how we can remove punctuation and numbers from the vocabulary in the previous section. We can repeat this for all documents and build a set of all known words.

We can develop a vocabulary as a `Counter`, which is a dictionary mapping of words and their count that allows us to easily update and query. Each document can be added to the counter (a new function called `add_doc_to_vocab()`) and we can step over all of the reviews in the negative directory and then the positive directory (a new function called `process_docs()`). The complete example is listed below.

```
import string
import re
from os import listdir
from collections import Counter
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
  # load doc
  doc = load_doc(filename)
  # clean doc
  tokens = clean_doc(doc)
  # update counts
```

```
  vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # add doc to vocab
    add_doc_to_vocab(path, vocab)

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/pos', vocab)
process_docs('txt_sentoken/neg', vocab)
# print the size of the vocab
print(len(vocab))
# print the top words in the vocab
print(vocab.most_common(50))
```

Listing 15.3: Example of selecting a vocabulary for the dataset.

Running the example shows that we have a vocabulary of 44,276 words. We also can see a sample of the top 50 most used words in the movie reviews. Note that this vocabulary was constructed based on only those reviews in the training dataset.

```
44276
[('film', 7983), ('one', 4946), ('movie', 4826), ('like', 3201), ('even', 2262), ('good',
    2080), ('time', 2041), ('story', 1907), ('films', 1873), ('would', 1844), ('much',
    1824), ('also', 1757), ('characters', 1735), ('get', 1724), ('character', 1703),
    ('two', 1643), ('first', 1588), ('see', 1557), ('way', 1515), ('well', 1511), ('make',
    1418), ('really', 1407), ('little', 1351), ('life', 1334), ('plot', 1288), ('people',
    1269), ('could', 1248), ('bad', 1248), ('scene', 1241), ('movies', 1238), ('never',
    1201), ('best', 1179), ('new', 1140), ('scenes', 1135), ('man', 1131), ('many', 1130),
    ('doesnt', 1118), ('know', 1092), ('dont', 1086), ('hes', 1024), ('great', 1014),
    ('another', 992), ('action', 985), ('love', 977), ('us', 967), ('go', 952),
    ('director', 948), ('end', 946), ('something', 945), ('still', 936)]
```

Listing 15.4: Example output of selecting a vocabulary for the dataset.

We can step through the vocabulary and remove all words that have a low occurrence, such as only being used once or twice in all reviews. For example, the following snippet will retrieve only the tokens that appear 2 or more times in all reviews.

```
# keep tokens with a min occurrence
min_occurane = 2
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
```

Listing 15.5: Example of filtering the vocabulary by occurrence.

Finally, the vocabulary can be saved to a new file called `vocab.txt` that we can later load and use to filter movie reviews prior to encoding them for modeling. We define a new function

called `save_list()` that saves the vocabulary to file, with one word per line. For example:

```
# save list to file
def save_list(lines, filename):
  # convert lines to a single blob of text
  data = '\n'.join(lines)
  # open file
  file = open(filename, 'w')
  # write text
  file.write(data)
  # close file
  file.close()

# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')
```

Listing 15.6: Example of saving the filtered vocabulary.

Pulling all of this together, the complete example is listed below.

```
import string
import re
from os import listdir
from collections import Counter
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
  # load doc
  doc = load_doc(filename)
```

```
  # clean doc
  tokens = clean_doc(doc)
  # update counts
  vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # add doc to vocab
    add_doc_to_vocab(path, vocab)

# save list to file
def save_list(lines, filename):
  # convert lines to a single blob of text
  data = '\n'.join(lines)
  # open file
  file = open(filename, 'w')
  # write text
  file.write(data)
  # close file
  file.close()

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/pos', vocab)
process_docs('txt_sentoken/neg', vocab)
# print the size of the vocab
print(len(vocab))
# keep tokens with a min occurrence
min_occurane = 2
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')
```

Listing 15.7: Example of filtering the vocabulary for the dataset.

Running the above example with this addition shows that the vocabulary size drops by a little more than half its size, from 44,276 to 25,767 words.

```
25767
```

Listing 15.8: Example output of filtering the vocabulary by min occurrence.

Running the min occurrence filter on the vocabulary and saving it to file, you should now have a new file called `vocab.txt` with only the words we are interested in. The order of words in your file will differ, but should look something like the following:

```
aberdeen
dupe
```

```
burt
libido
hamlet
arlene
available
corners
web
columbia
...
```

Listing 15.9: Sample of the vocabulary file `vocab.txt`.

We are now ready to look at extracting features from the reviews ready for modeling.

15.4 Train CNN With Embedding Layer

In this section, we will learn a word embedding while training a convolutional neural network on the classification problem. A word embedding is a way of representing text where each word in the vocabulary is represented by a real valued vector in a high-dimensional space. The vectors are learned in such a way that words that have similar meanings will have similar representation in the vector space (close in the vector space). This is a more expressive representation for text than more classical methods like bag-of-words, where relationships between words or tokens are ignored, or forced in bigram and trigram approaches.

The real valued vector representation for words can be learned while training the neural network. We can do this in the Keras deep learning library using the `Embedding` layer. The first step is to load the vocabulary. We will use it to filter out words from movie reviews that we are not interested in. If you have worked through the previous section, you should have a local file called `vocab.txt` with one word per line. We can load that file and build a vocabulary as a set for checking the validity of tokens.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
```

Listing 15.10: Load vocabulary.

Next, we need to load all of the training data movie reviews. For that we can adapt the `process_docs()` from the previous section to load the documents, clean them, and return them as a list of strings, with one document per string. We want each document to be a string for easy encoding as a sequence of integers later. Cleaning the document involves splitting each review based on white space, removing punctuation, and then filtering out all tokens not in the vocabulary. The updated `clean_doc()` function is listed below.

```
# turn a doc into clean tokens
def clean_doc(doc, vocab):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # filter out tokens not in vocab
  tokens = [w for w in tokens if w in vocab]
  tokens = ' '.join(tokens)
  return tokens
```

Listing 15.11: Function to load and filter a loaded review.

The updated `process_docs()` can then call the `clean_doc()` for each document in a given directory.

```
# load all docs in a directory
def process_docs(directory, vocab, is_train):
  documents = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if is_train and filename.startswith('cv9'):
      continue
    if not is_train and not filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load the doc
    doc = load_doc(path)
    # clean doc
    tokens = clean_doc(doc, vocab)
    # add to list
    documents.append(tokens)
  return documents
```

Listing 15.12: Example to clean all movie reviews.

We can call the `process_docs` function for both the `neg` and `pos` directories and combine the reviews into a single train or test dataset. We also can define the class labels for the dataset. The `load_clean_dataset()` function below will load all reviews and prepare class labels for the training or test dataset.

```
# load and clean a dataset
def load_clean_dataset(vocab, is_train):
  # load documents
  neg = process_docs('txt_sentoken/neg', vocab, is_train)
  pos = process_docs('txt_sentoken/pos', vocab, is_train)
  docs = neg + pos
  # prepare labels
  labels = array([0 for _ in range(len(neg))] + [1 for _ in range(len(pos))])
  return docs, labels
```

Listing 15.13: Function to load and clean all train or test movie reviews.

The next step is to encode each document as a sequence of integers. The Keras `Embedding` layer requires integer inputs where each integer maps to a single token that has a specific real-valued vector representation within the embedding. These vectors are random at the beginning of training, but during training become meaningful to the network. We can encode the training documents as sequences of integers using the `Tokenizer` class in the Keras API. First, we must construct an instance of the class then train it on all documents in the training dataset. In this case, it develops a vocabulary of all tokens in the training dataset and develops a consistent mapping from words in the vocabulary to unique integers. We could just as easily develop this mapping ourselves using our vocabulary file. The `create_tokenizer()` function below will prepare a `Tokenizer` from the training data.

```
# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer
```

Listing 15.14: Function to create a `Tokenizer` from training.

Now that the mapping of words to integers has been prepared, we can use it to encode the reviews in the training dataset. We can do that by calling the `texts_to_sequences()` function on the `Tokenizer`. We also need to ensure that all documents have the same length. This is a requirement of Keras for efficient computation. We could truncate reviews to the smallest size or zero-pad (pad with the value `0`) reviews to the maximum length, or some hybrid. In this case, we will pad all reviews to the length of the longest review in the training dataset. First, we can find the longest review using the `max()` function on the training dataset and take its length. We can then call the Keras function `pad_sequences()` to pad the sequences to the maximum length by adding 0 values on the end.

```
max_length = max([len(s.split()) for s in train_docs])
print('Maximum length: %d' % max_length)
```

Listing 15.15: Calculate the maximum movie review length.

We can then use the maximum length as a parameter to a function to integer encode and pad the sequences.

```
# integer encode and pad documents
def encode_docs(tokenizer, max_length, docs):
  # integer encode
  encoded = tokenizer.texts_to_sequences(docs)
  # pad sequences
  padded = pad_sequences(encoded, maxlen=max_length, padding='post')
  return padded
```

Listing 15.16: Function to integer encode and pad movie reviews.

We are now ready to define our neural network model. The model will use an `Embedding` layer as the first hidden layer. The `Embedding` layer requires the specification of the vocabulary size, the size of the real-valued vector space, and the maximum length of input documents. The vocabulary size is the total number of words in our vocabulary, plus one for unknown words. This could be the vocab set length or the size of the vocab within the tokenizer used to integer encode the documents, for example:

```
# define vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary size: %d' % vocab_size)
```

Listing 15.17: Calculate the size of the vocabulary for the `Embedding` layer.

We will use a 100-dimensional vector space, but you could try other values, such as 50 or 150. Finally, the maximum document length was calculated above in the `max_length` variable used during padding. The complete model definition is listed below including the `Embedding` layer. We use a Convolutional Neural Network (CNN) as they have proven to be successful at document classification problems. A conservative CNN configuration is used with 32 filters (parallel fields for processing words) and a kernel size of 8 with a rectified linear (`relu`) activation function. This is followed by a pooling layer that reduces the output of the convolutional layer by half.

Next, the 2D output from the CNN part of the model is flattened to one long 2D vector to represent the *features* extracted by the CNN. The back-end of the model is a standard Multilayer Perceptron layers to interpret the CNN features. The output layer uses a sigmoid activation function to output a value between 0 and 1 for the negative and positive sentiment in the review.

```
# define the model
def define_model(vocab_size, max_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 100, input_length=max_length))
  model.add(Conv1D(filters=32, kernel_size=8, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(10, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  # compile network
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model
```

Listing 15.18: Define a CNN model with the Embedding Layer.

Running just this piece provides a summary of the defined network. We can see that the `Embedding` layer expects documents with a length of 1,317 words as input and encodes each word in the document as a 100 element vector.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 1317, 100)         2576800
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 1310, 32)          25632
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 655, 32)           0
_________________________________________________________________
flatten_1 (Flatten)          (None, 20960)             0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                209610
_________________________________________________________________
```

```
dense_2 (Dense)              (None, 1)                 11
=================================================================
Total params: 2,812,053
Trainable params: 2,812,053
Non-trainable params: 0
_________________________________________________________________
```

Listing 15.19: Summary of the defined model.

A plot the defined model is then saved to file with the name `model.png`.

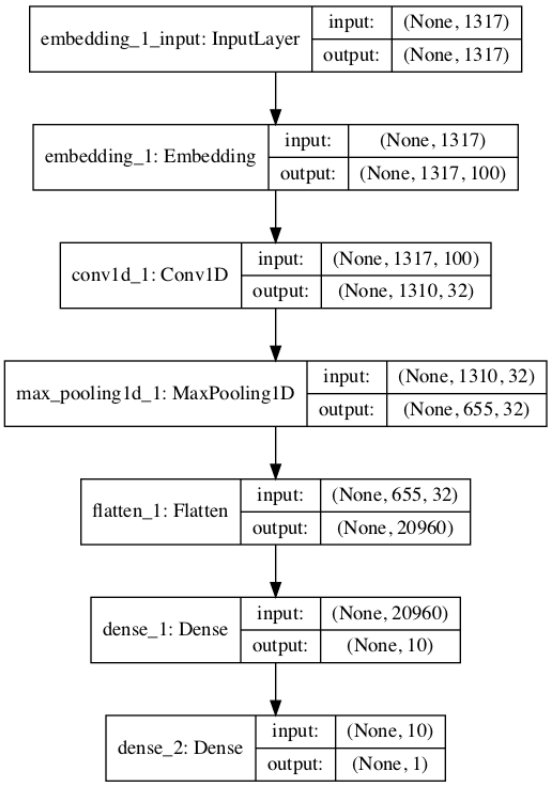

Figure 15.1: Plot of the defined CNN classification model.

Next, we fit the network on the training data. We use a binary cross entropy loss function because the problem we are learning is a binary classification problem. The efficient Adam implementation of stochastic gradient descent is used and we keep track of accuracy in addition to loss during training. The model is trained for 10 epochs, or 10 passes through the training data. The network configuration and training schedule were found with a little trial and error, but are by no means optimal for this problem. If you can get better results with a different configuration, let me know.

```
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
```

Listing 15.20: Train the defined classification model.

After the model is fit, it is saved to a file named `model.h5` for later evaluation.

```
# save the model
model.save('model.h5')
```

Listing 15.21: Save the fit model to file.

We can tie all of this together. The complete code listing is provided below.

```
import string
import re
from os import listdir
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc, vocab):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # filter out tokens not in vocab
  tokens = [w for w in tokens if w in vocab]
  tokens = ' '.join(tokens)
  return tokens

# load all docs in a directory
def process_docs(directory, vocab, is_train):
  documents = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if is_train and filename.startswith('cv9'):
      continue
    if not is_train and not filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load the doc
    doc = load_doc(path)
```

```
    # clean doc
    tokens = clean_doc(doc, vocab)
    # add to list
    documents.append(tokens)
  return documents

# load and clean a dataset
def load_clean_dataset(vocab, is_train):
  # load documents
  neg = process_docs('txt_sentoken/neg', vocab, is_train)
  pos = process_docs('txt_sentoken/pos', vocab, is_train)
  docs = neg + pos
  # prepare labels
  labels = array([0 for _ in range(len(neg))] + [1 for _ in range(len(pos))])
  return docs, labels

# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer

# integer encode and pad documents
def encode_docs(tokenizer, max_length, docs):
  # integer encode
  encoded = tokenizer.texts_to_sequences(docs)
  # pad sequences
  padded = pad_sequences(encoded, maxlen=max_length, padding='post')
  return padded

# define the model
def define_model(vocab_size, max_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 100, input_length=max_length))
  model.add(Conv1D(filters=32, kernel_size=8, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(10, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  # compile network
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load training data
train_docs, ytrain = load_clean_dataset(vocab, True)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# define vocabulary size
vocab_size = len(tokenizer.word_index) + 1
```

```
print('Vocabulary size: %d' % vocab_size)
# calculate the maximum sequence length
max_length = max([len(s.split()) for s in train_docs])
print('Maximum length: %d' % max_length)
# encode data
Xtrain = encode_docs(tokenizer, max_length, train_docs)
# define model
model = define_model(vocab_size, max_length)
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
# save the model
model.save('model.h5')
```

Listing 15.22: Complete example of fitting a CNN model with an `Embedding` input layer.

Running the example will first provide a summary of the training dataset vocabulary (25,768) and maximum input sequence length in words (1,317). The example should run in a few minutes and the fit model will be saved to file.

```
...
Vocabulary size: 25768
Maximum length: 1317
Epoch 1/10
8s - loss: 0.6927 - acc: 0.4800
Epoch 2/10
7s - loss: 0.6610 - acc: 0.5922
Epoch 3/10
7s - loss: 0.3461 - acc: 0.8844
Epoch 4/10
7s - loss: 0.0441 - acc: 0.9889
Epoch 5/10
7s - loss: 0.0058 - acc: 1.0000
Epoch 6/10
7s - loss: 0.0024 - acc: 1.0000
Epoch 7/10
7s - loss: 0.0015 - acc: 1.0000
Epoch 8/10
7s - loss: 0.0011 - acc: 1.0000
Epoch 9/10
7s - loss: 8.0111e-04 - acc: 1.0000
Epoch 10/10
7s - loss: 5.4109e-04 - acc: 1.0000
```

Listing 15.23: Example output from fitting the model.

15.5 Evaluate Model

In this section, we will evaluate the trained model and use it to make predictions on new data. First, we can use the built-in `evaluate()` function to estimate the skill of the model on both the training and test dataset. This requires that we load and encode both the training and test datasets.

```
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
```

```
test_docs, ytest = load_clean_dataset(vocab, False)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# define vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary size: %d' % vocab_size)
# calculate the maximum sequence length
max_length = max([len(s.split()) for s in train_docs])
print('Maximum length: %d' % max_length)
# encode data
Xtrain = encode_docs(tokenizer, max_length, train_docs)
Xtest = encode_docs(tokenizer, max_length, test_docs)
```

Listing 15.24: Load and encode both training and test datasets.

We can then load the model and evaluate it on both datasets and print the accuracy.

```
# load the model
model = load_model('model.h5')
# evaluate model on training dataset
_, acc = model.evaluate(Xtrain, ytrain, verbose=0)
print('Train Accuracy: %f' % (acc*100))
# evaluate model on test dataset
_, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %f' % (acc*100))
```

Listing 15.25: Load and evaluate model on both train and test datasets.

New data must then be prepared using the same text encoding and encoding schemes as was used on the training dataset. Once prepared, a prediction can be made by calling the `predict()` function on the model. The function below named `predict_sentiment()` will encode and pad a given movie review text and return a prediction in terms of both the percentage and a label.

```
# classify a review as negative or positive
def predict_sentiment(review, vocab, tokenizer, max_length, model):
  # clean review
  line = clean_doc(review, vocab)
  # encode and pad review
  padded = encode_docs(tokenizer, max_length, [line])
  # predict sentiment
  yhat = model.predict(padded, verbose=0)
  # retrieve predicted percentage and label
  percent_pos = yhat[0,0]
  if round(percent_pos) == 0:
    return (1-percent_pos), 'NEGATIVE'
  return percent_pos, 'POSITIVE'
```

Listing 15.26: Function to predict the sentiment for an ad hoc movie review.

We can test out this model with two ad hoc movie reviews. The complete example is listed below.

```
import string
import re
from os import listdir
from numpy import array
from keras.preprocessing.text import Tokenizer
```

```
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc, vocab):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # filter out tokens not in vocab
  tokens = [w for w in tokens if w in vocab]
  tokens = ' '.join(tokens)
  return tokens

# load all docs in a directory
def process_docs(directory, vocab, is_train):
  documents = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if is_train and filename.startswith('cv9'):
      continue
    if not is_train and not filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load the doc
    doc = load_doc(path)
    # clean doc
    tokens = clean_doc(doc, vocab)
    # add to list
    documents.append(tokens)
  return documents

# load and clean a dataset
def load_clean_dataset(vocab, is_train):
  # load documents
  neg = process_docs('txt_sentoken/neg', vocab, is_train)
  pos = process_docs('txt_sentoken/pos', vocab, is_train)
  docs = neg + pos
  # prepare labels
  labels = array([0 for _ in range(len(neg))] + [1 for _ in range(len(pos))])
  return docs, labels
```

```
# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer

# integer encode and pad documents
def encode_docs(tokenizer, max_length, docs):
  # integer encode
  encoded = tokenizer.texts_to_sequences(docs)
  # pad sequences
  padded = pad_sequences(encoded, maxlen=max_length, padding='post')
  return padded

# classify a review as negative or positive
def predict_sentiment(review, vocab, tokenizer, max_length, model):
  # clean review
  line = clean_doc(review, vocab)
  # encode and pad review
  padded = encode_docs(tokenizer, max_length, [line])
  # predict sentiment
  yhat = model.predict(padded, verbose=0)
  # retrieve predicted percentage and label
  percent_pos = yhat[0,0]
  if round(percent_pos) == 0:
    return (1-percent_pos), 'NEGATIVE'
  return percent_pos, 'POSITIVE'

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# define vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary size: %d' % vocab_size)
# calculate the maximum sequence length
max_length = max([len(s.split()) for s in train_docs])
print('Maximum length: %d' % max_length)
# encode data
Xtrain = encode_docs(tokenizer, max_length, train_docs)
Xtest = encode_docs(tokenizer, max_length, test_docs)
# load the model
model = load_model('model.h5')
# evaluate model on training dataset
_, acc = model.evaluate(Xtrain, ytrain, verbose=0)
print('Train Accuracy: %.2f' % (acc*100))
# evaluate model on test dataset
_, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %.2f' % (acc*100))

# test positive text
```

```
text = 'Everyone will enjoy this film. I love it, recommended!'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, max_length, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))
# test negative text
text = 'This is a bad movie. Do not watch it. It sucks.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, max_length, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))
```

Listing 15.27: Complete example of making a prediction on new text data.

Running the example first prints the skill of the model on the training and test dataset. We can see that the model achieves 100% accuracy on the training dataset and 87.5% on the test dataset, an impressive score.

Next, we can see that the model makes the correct prediction on two contrived movie reviews. We can see that the percentage or confidence of the prediction is close to 50% for both, this may be because the two contrived reviews are very short and the model is expecting sequences of 1,000 or more words.

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

```
Train Accuracy: 100.00
Test Accuracy: 87.50
Review: [Everyone will enjoy this film. I love it, recommended!]
Sentiment: POSITIVE (55.431%)
Review: [This is a bad movie. Do not watch it. It sucks.]
Sentiment: NEGATIVE (54.746%)
```

Listing 15.28: Example output from making a prediction on new reviews.

15.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Cleaning**. Explore better data cleaning, perhaps leaving some punctuation in tact or normalizing contractions.
- **Truncated Sequences**. Padding all sequences to the length of the longest sequence might be extreme if the longest sequence is very different to all other reviews. Study the distribution of review lengths and truncate reviews to a mean length.
- **Truncated Vocabulary**. We removed infrequently occurring words, but still had a large vocabulary of more than 25,000 words. Explore further reducing the size of the vocabulary and the effect on model skill.
- **Filters and Kernel Size**. The number of filters and kernel size are important to model skill and were not tuned. Explore tuning these two CNN parameters.
- **Epochs and Batch Size**. The model appears to fit the training dataset quickly. Explore alternate configurations of the number of training epochs and batch size and use the test dataset as a validation set to pick a better stopping point for training the model.

- **Deeper Network**. Explore whether a deeper network results in better skill, either in terms of CNN layers, MLP layers and both.
- **Pre-Train an Embedding**. Explore pre-training a Word2Vec word embedding in the model and the impact on model skill with and without further fine tuning during training.
- **Use GloVe Embedding**. Explore loading the pre-trained GloVe embedding and the impact on model skill with and without further fine tuning during training.
- **Longer Test Reviews**. Explore whether the skill of model predictions is dependent on the length of movie reviews as suspected in the final section on evaluating the model.
- **Train Final Model**. Train a final model on all available data and use it make predictions on real ad hoc movie reviews from the internet.

If you explore any of these extensions, I'd love to know.

15.7 Further Reading

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

15.7.1 Dataset

- Movie Review Data.
  http://www.cs.cornell.edu/people/pabo/movie-review-data/
- *A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts*, 2004.
  http://xxx.lanl.gov/abs/cs/0409058
- Movie Review Polarity Dataset.
  http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

15.7.2 APIs

- `collections` API - Container datatypes.
  https://docs.python.org/3/library/collections.html
- `Tokenizer` Keras API.
  https://keras.io/preprocessing/text/#tokenizer
- `Embedding` Keras API.
  https://keras.io/layers/embeddings/

15.8 Summary

In this tutorial, you discovered how to develop word embeddings for the classification of movie reviews. Specifically, you learned:

- How to prepare movie review text data for classification with deep learning methods.
- How to develop a neural classification model with word embedding and convolutional layers.
- How to evaluate the developed a neural classification model.

15.8.1 Next

In the next chapter, you will discover how you can develop an n-gram multichannel convolutional neural network for text classification.

Chapter 16

Project: Develop an n-gram CNN Model for Sentiment Analysis

A standard deep learning model for text classification and sentiment analysis uses a word embedding layer and one-dimensional convolutional neural network. The model can be expanded by using multiple parallel convolutional neural networks that read the source document using different kernel sizes. This, in effect, creates a multichannel convolutional neural network for text that reads text with different n-gram sizes (groups of words). In this tutorial, you will discover how to develop a multichannel convolutional neural network for sentiment prediction on text movie review data. After completing this tutorial, you will know:

- How to prepare movie review text data for modeling.
- How to develop a multichannel convolutional neural network for text in Keras.
- How to evaluate a fit model on unseen movie review data.

Let's get started.

16.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Movie Review Dataset.
2. Data Preparation.
3. Develop Multichannel Model.
4. Evaluate Model.

16.2 Movie Review Dataset

In this tutorial, we will use the Movie Review Dataset. This dataset designed for sentiment analysis was described previously in Chapter 9. You can download the dataset from here:

- Movie Review Polarity Dataset (`review_polarity.tar.gz`, 3MB).
  http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

After unzipping the file, you will have a directory called `txt_sentoken` with two sub-directories containing the text *neg* and *pos* for negative and positive reviews. Reviews are stored one per file with a naming convention *cv000* to *cv999* for each of *neg* and *pos*.

16.3 Data Preparation

Note: The preparation of the movie review dataset was first described in Chapter 9. In this section, we will look at 3 things:

1. Separation of data into training and test sets.
2. Loading and cleaning the data to remove punctuation and numbers.
3. Clean All Reviews and Save.

16.3.1 Split into Train and Test Sets

We are pretending that we are developing a system that can predict the sentiment of a textual movie review as either positive or negative. This means that after the model is developed, we will need to make predictions on new textual reviews. This will require all of the same data preparation to be performed on those new reviews as is performed on the training data for the model. We will ensure that this constraint is built into the evaluation of our models by splitting the training and test datasets prior to any data preparation. This means that any knowledge in the data in the test set that could help us better prepare the data (e.g. the words used) are unavailable in the preparation of data used for training the model.

That being said, we will use the last 100 positive reviews and the last 100 negative reviews as a test set (100 reviews) and the remaining 1,800 reviews as the training dataset. This is a 90% train, 10% split of the data. The split can be imposed easily by using the filenames of the reviews where reviews named 000 to 899 are for training data and reviews named 900 onwards are for test.

16.3.2 Loading and Cleaning Reviews

The text data is already pretty clean; not much preparation is required. Without getting bogged down too much in the details, we will prepare the data using the following way:

- Split tokens on white space.
- Remove all punctuation from words.
- Remove all words that are not purely comprised of alphabetical characters.
- Remove all words that are known stop words.
- Remove all words that have a length ≤ 1 character.

We can put all of these steps into a function called `clean_doc()` that takes as an argument the raw text loaded from a file and returns a list of cleaned tokens. We can also define a function `load_doc()` that loads a document from file ready for use with the `clean_doc()` function. An example of cleaning the first positive review is listed below.

```
from nltk.corpus import stopwords
import string
import re

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  return tokens

# load the document
filename = 'txt_sentoken/pos/cv000_29590.txt'
text = load_doc(filename)
tokens = clean_doc(text)
print(tokens)
```

Listing 16.1: Example of cleaning a movie review.

Running the example prints a long list of clean tokens. There are many more cleaning steps we may want to explore and I leave them as further exercises.

```
...
'creepy', 'place', 'even', 'acting', 'hell', 'solid', 'dreamy', 'depp', 'turning',
    'typically', 'strong', 'performance', 'deftly', 'handling', 'british', 'accent',
    'ians', 'holm', 'joe', 'goulds', 'secret', 'richardson', 'dalmatians', 'log', 'great',
    'supporting', 'roles', 'big', 'surprise', 'graham', 'cringed', 'first', 'time',
    'opened', 'mouth', 'imagining', 'attempt', 'irish', 'accent', 'actually', 'wasnt',
    'half', 'bad', 'film', 'however', 'good', 'strong', 'violencegore', 'sexuality',
    'language', 'drug', 'content']
```

Listing 16.2: Example output of cleaning a movie review.

16.3.3 Clean All Reviews and Save

We can now use the function to clean reviews and apply it to all reviews. To do this, we will develop a new function named `process_docs()` below that will walk through all reviews in a directory, clean them, and return them as a list. We will also add an argument to the function to indicate whether the function is processing train or test reviews, that way the filenames can be filtered (as described above) and only those train or test reviews requested will be cleaned and returned. The full function is listed below.

```
# load all docs in a directory
def process_docs(directory, is_train):
  documents = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if is_train and filename.startswith('cv9'):
      continue
    if not is_train and not filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load the doc
    doc = load_doc(path)
    # clean doc
    tokens = clean_doc(doc)
    # add to list
    documents.append(tokens)
  return documents
```

Listing 16.3: Function for cleaning multiple review documents.

We can call this function with negative training reviews. We also need labels for the train and test documents. We know that we have 900 training documents and 100 test documents. We can use a Python list comprehension to create the labels for the negative (0) and positive (1) reviews for both train and test sets. The function below named `load_clean_dataset()` will load and clean the movie review text and also create the labels for the reviews.

```
# load and clean a dataset
def load_clean_dataset(is_train):
  # load documents
  neg = process_docs('txt_sentoken/neg', is_train)
  pos = process_docs('txt_sentoken/pos', is_train)
  docs = neg + pos
  # prepare labels
  labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
  return docs, labels
```

Listing 16.4: Function to prepare reviews and labels for a dataset.

Finally, we want to save the prepared train and test sets to file so that we can load them later for modeling and model evaluation. The function below-named `save_dataset()` will save a given prepared dataset (`X` and `y` elements) to a file using the pickle API (this is the standard API for saving objects in Python).

```
# save a dataset to file
def save_dataset(dataset, filename):
```

```
  dump(dataset, open(filename, 'wb'))
  print('Saved: %s' % filename)
```

Listing 16.5: Function for saving clean documents to file.

16.3.4 Complete Example

We can tie all of these data preparation steps together. The complete example is listed below.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords
from pickle import dump

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # filter out stop words
  stop_words = set(stopwords.words('english'))
  tokens = [w for w in tokens if not w in stop_words]
  # filter out short tokens
  tokens = [word for word in tokens if len(word) > 1]
  tokens = ' '.join(tokens)
  return tokens

# load all docs in a directory
def process_docs(directory, is_train):
  documents = list()
  # walk through all files in the folder
  for filename in listdir(directory):
    # skip any reviews in the test set
    if is_train and filename.startswith('cv9'):
      continue
    if not is_train and not filename.startswith('cv9'):
      continue
    # create the full path of the file to open
    path = directory + '/' + filename
```

```
    # load the doc
    doc = load_doc(path)
    # clean doc
    tokens = clean_doc(doc)
    # add to list
    documents.append(tokens)
  return documents

# load and clean a dataset
def load_clean_dataset(is_train):
  # load documents
  neg = process_docs('txt_sentoken/neg', is_train)
  pos = process_docs('txt_sentoken/pos', is_train)
  docs = neg + pos
  # prepare labels
  labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
  return docs, labels

# save a dataset to file
def save_dataset(dataset, filename):
  dump(dataset, open(filename, 'wb'))
  print('Saved: %s' % filename)

# load and clean all reviews
train_docs, ytrain = load_clean_dataset(True)
test_docs, ytest = load_clean_dataset(False)
# save training datasets
save_dataset([train_docs, ytrain], 'train.pkl')
save_dataset([test_docs, ytest], 'test.pkl')
```

Listing 16.6: Complete example of cleaning and saving all movie reviews.

Running the example cleans the text movie review documents, creates labels, and saves the prepared data for both train and test datasets in `train.pkl` and `test.pkl` respectively. Now we are ready to develop our model.

16.4 Develop Multichannel Model

In this section, we will develop a multichannel convolutional neural network for the sentiment analysis prediction problem. This section is divided into 3 parts:

1. Encode Data
2. Define Model.
3. Complete Example.

16.4.1 Encode Data

The first step is to load the cleaned training dataset. The function below-named `load_dataset()` can be called to load the pickled training dataset.

```
# load a clean dataset
def load_dataset(filename):
  return load(open(filename, 'rb'))

trainLines, trainLabels = load_dataset('train.pkl')
```

Listing 16.7: Example of loading the cleaned and saved reviews.

Next, we must fit a Keras `Tokenizer` on the training dataset. We will use this tokenizer to both define the vocabulary for the `Embedding` layer and encode the review documents as integers. The function `create_tokenizer()` below will create a `Tokenizer` given a list of documents.

```
# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer
```

Listing 16.8: Function for creating a `Tokenizer`.

We also need to know the maximum length of input sequences as input for the model and to pad all sequences to the fixed length. The function `max_length()` below will calculate the maximum length (number of words) for all reviews in the training dataset.

```
# calculate the maximum document length
def max_length(lines):
  return max([len(s.split()) for s in lines])
```

Listing 16.9: Function to calculate the maximum movie review length.

We also need to know the size of the vocabulary for the `Embedding` layer. This can be calculated from the prepared `Tokenizer`, as follows:

```
# calculate vocabulary size
vocab_size = len(tokenizer.word_index) + 1
```

Listing 16.10: Calculate the size of the vocabulary.

Finally, we can integer encode and pad the clean movie review text. The function below named `encode_text()` will both encode and pad text data to the maximum review length.

```
# encode a list of lines
def encode_text(tokenizer, lines, length):
  # integer encode
  encoded = tokenizer.texts_to_sequences(lines)
  # pad encoded sequences
  padded = pad_sequences(encoded, maxlen=length, padding='post')
  return padded
```

Listing 16.11: Function to encode and pad movie review text.

16.4.2 Define Model

A standard model for document classification is to use an `Embedding` layer as input, followed by a one-dimensional convolutional neural network, pooling layer, and then a prediction output layer. The kernel size in the convolutional layer defines the number of words to consider as

the convolution is passed across the input text document, providing a grouping parameter. A multi-channel convolutional neural network for document classification involves using multiple versions of the standard model with different sized kernels. This allows the document to be processed at different resolutions or different n-grams (groups of words) at a time, whilst the model learns how to best integrate these interpretations.

This approach was first described by Yoon Kim in his 2014 paper titled *Convolutional Neural Networks for Sentence Classification*. In the paper, Kim experimented with static and dynamic (updated) embedding layers, we can simplify the approach and instead focus only on the use of different kernel sizes. This approach is best understood with a diagram taken from Kim's paper, see Chapter 14.

In Keras, a multiple-input model can be defined using the functional API. We will define a model with three input channels for processing 4-grams, 6-grams, and 8-grams of movie review text. Each channel is comprised of the following elements:

- `Input` layer that defines the length of input sequences.
- `Embedding` layer set to the size of the vocabulary and 100-dimensional real-valued representations.
- `Conv1D` layer with 32 filters and a kernel size set to the number of words to read at once.
- `MaxPooling1D` layer to consolidate the output from the convolutional layer.
- `Flatten` layer to reduce the three-dimensional output to two dimensional for concatenation.

The output from the three channels are concatenated into a single vector and process by a `Dense` layer and an output layer. The function below defines and returns the model. As part of defining the model, a summary of the defined model is printed and a plot of the model graph is created and saved to file.

```
# define the model
def define_model(length, vocab_size):
  # channel 1
  inputs1 = Input(shape=(length,))
  embedding1 = Embedding(vocab_size, 100)(inputs1)
  conv1 = Conv1D(filters=32, kernel_size=4, activation='relu')(embedding1)
  drop1 = Dropout(0.5)(conv1)
  pool1 = MaxPooling1D(pool_size=2)(drop1)
  flat1 = Flatten()(pool1)
  # channel 2
  inputs2 = Input(shape=(length,))
  embedding2 = Embedding(vocab_size, 100)(inputs2)
  conv2 = Conv1D(filters=32, kernel_size=6, activation='relu')(embedding2)
  drop2 = Dropout(0.5)(conv2)
  pool2 = MaxPooling1D(pool_size=2)(drop2)
  flat2 = Flatten()(pool2)
  # channel 3
  inputs3 = Input(shape=(length,))
  embedding3 = Embedding(vocab_size, 100)(inputs3)
  conv3 = Conv1D(filters=32, kernel_size=8, activation='relu')(embedding3)
  drop3 = Dropout(0.5)(conv3)
  pool3 = MaxPooling1D(pool_size=2)(drop3)
  flat3 = Flatten()(pool3)
```

```
  # merge
  merged = concatenate([flat1, flat2, flat3])
  # interpretation
  dense1 = Dense(10, activation='relu')(merged)
  outputs = Dense(1, activation='sigmoid')(dense1)
  model = Model(inputs=[inputs1, inputs2, inputs3], outputs=outputs)
  # compile
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize
  model.summary()
  plot_model(model, show_shapes=True, to_file='multichannel.png')
  return model
```

Listing 16.12: Function for defining the classification model.

16.4.3 Complete Example

Pulling all of this together, the complete example is listed below.

```
from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils.vis_utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.merge import concatenate

# load a clean dataset
def load_dataset(filename):
  return load(open(filename, 'rb'))

# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer

# calculate the maximum document length
def max_length(lines):
  return max([len(s.split()) for s in lines])

# encode a list of lines
def encode_text(tokenizer, lines, length):
  # integer encode
  encoded = tokenizer.texts_to_sequences(lines)
  # pad encoded sequences
  padded = pad_sequences(encoded, maxlen=length, padding='post')
  return padded
```

```
# define the model
def define_model(length, vocab_size):
  # channel 1
  inputs1 = Input(shape=(length,))
  embedding1 = Embedding(vocab_size, 100)(inputs1)
  conv1 = Conv1D(filters=32, kernel_size=4, activation='relu')(embedding1)
  drop1 = Dropout(0.5)(conv1)
  pool1 = MaxPooling1D(pool_size=2)(drop1)
  flat1 = Flatten()(pool1)
  # channel 2
  inputs2 = Input(shape=(length,))
  embedding2 = Embedding(vocab_size, 100)(inputs2)
  conv2 = Conv1D(filters=32, kernel_size=6, activation='relu')(embedding2)
  drop2 = Dropout(0.5)(conv2)
  pool2 = MaxPooling1D(pool_size=2)(drop2)
  flat2 = Flatten()(pool2)
  # channel 3
  inputs3 = Input(shape=(length,))
  embedding3 = Embedding(vocab_size, 100)(inputs3)
  conv3 = Conv1D(filters=32, kernel_size=8, activation='relu')(embedding3)
  drop3 = Dropout(0.5)(conv3)
  pool3 = MaxPooling1D(pool_size=2)(drop3)
  flat3 = Flatten()(pool3)
  # merge
  merged = concatenate([flat1, flat2, flat3])
  # interpretation
  dense1 = Dense(10, activation='relu')(merged)
  outputs = Dense(1, activation='sigmoid')(dense1)
  model = Model(inputs=[inputs1, inputs2, inputs3], outputs=outputs)
  # compile
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize
  model.summary()
  plot_model(model, show_shapes=True, to_file='model.png')
  return model

# load training dataset
trainLines, trainLabels = load_dataset('train.pkl')
# create tokenizer
tokenizer = create_tokenizer(trainLines)
# calculate max document length
length = max_length(trainLines)
print('Max document length: %d' % length)
# calculate vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary size: %d' % vocab_size)
# encode data
trainX = encode_text(tokenizer, trainLines, length)
# define model
model = define_model(length, vocab_size)
# fit model
model.fit([trainX,trainX,trainX], trainLabels, epochs=7, batch_size=16)
# save the model
model.save('model.h5')
```

Listing 16.13: Complete example of fitting the n-gram CNN model.

Running the example first prints a summary of the prepared training dataset.

```
Max document length: 1380
Vocabulary size: 44277
```

Listing 16.14: Example output from preparing the training data.

The model is fit relatively quickly and appears to show good skill on the training dataset.

```
...
Epoch 3/7
1800/1800 [==============================] - 29s - loss: 0.0460 - acc: 0.9894
Epoch 4/7
1800/1800 [==============================] - 30s - loss: 0.0041 - acc: 1.0000
Epoch 5/7
1800/1800 [==============================] - 31s - loss: 0.0010 - acc: 1.0000
Epoch 6/7
1800/1800 [==============================] - 30s - loss: 3.0271e-04 - acc: 1.0000
Epoch 7/7
1800/1800 [==============================] - 28s - loss: 1.3875e-04 - acc: 1.0000
```

Listing 16.15: Example output from fitting the model.

A plot of the defined model is saved to file, clearly showing the three input channels for the model.

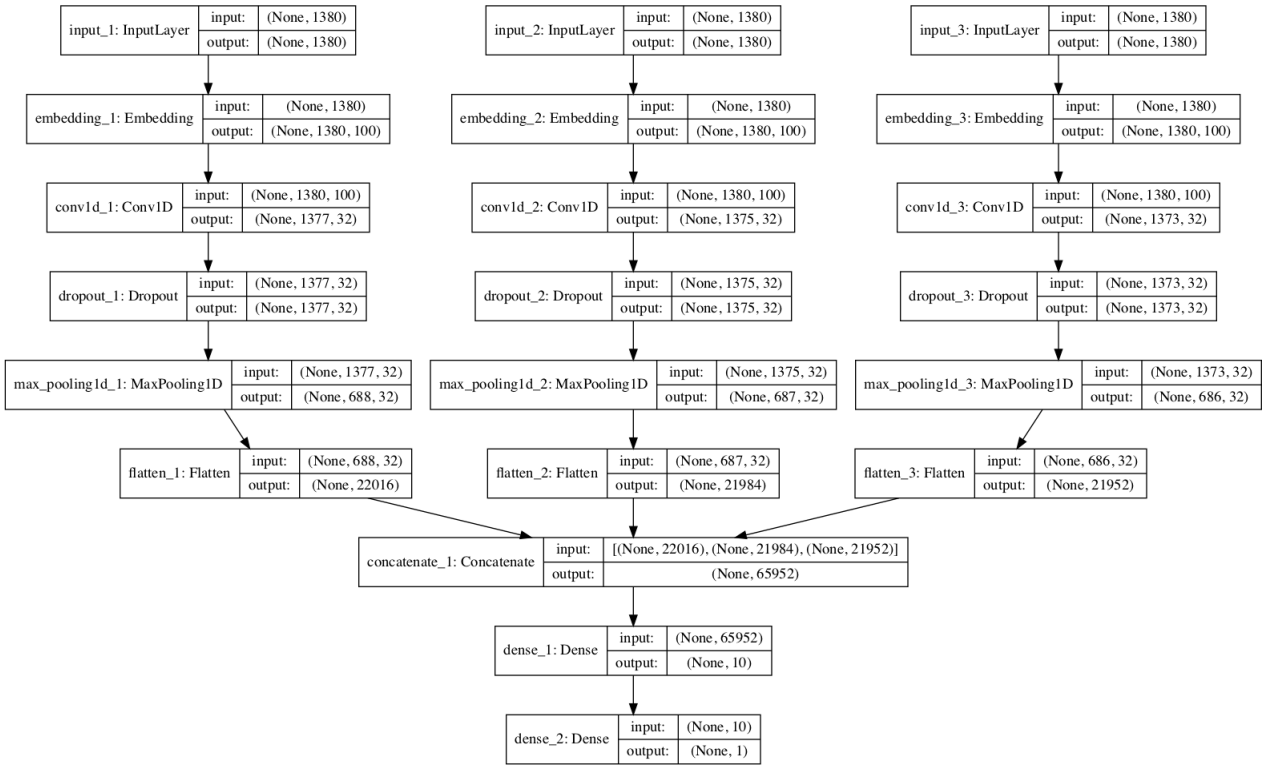

Figure 16.1: Plot of the Multichannel Convolutional Neural Network For Text.

The model is fit for a number of epochs and saved to the file `model.h5` for later evaluation.

16.5 Evaluate Model

In this section, we can evaluate the fit model by predicting the sentiment on all reviews in the unseen test dataset. Using the data loading functions developed in the previous section, we can load and encode both the training and test datasets.

```
# load datasets
trainLines, trainLabels = load_dataset('train.pkl')
testLines, testLabels = load_dataset('test.pkl')

# create tokenizer
tokenizer = create_tokenizer(trainLines)
# calculate max document length
length = max_length(trainLines)
# calculate vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Max document length: %d' % length)
print('Vocabulary size: %d' % vocab_size)
# encode data
trainX = encode_text(tokenizer, trainLines, length)
testX = encode_text(tokenizer, testLines, length)
print(trainX.shape, testX.shape)
```

Listing 16.16: Prepare train and test data for evaluating the model.

We can load the saved model and evaluate it on both the training and test datasets. The complete example is listed below.

```
from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model

# load a clean dataset
def load_dataset(filename):
  return load(open(filename, 'rb'))

# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer

# calculate the maximum document length
def max_length(lines):
  return max([len(s.split()) for s in lines])

# encode a list of lines
def encode_text(tokenizer, lines, length):
  # integer encode
  encoded = tokenizer.texts_to_sequences(lines)
  # pad encoded sequences
  padded = pad_sequences(encoded, maxlen=length, padding='post')
  return padded

# load datasets
```

```
trainLines, trainLabels = load_dataset('train.pkl')
testLines, testLabels = load_dataset('test.pkl')
# create tokenizer
tokenizer = create_tokenizer(trainLines)
# calculate max document length
length = max_length(trainLines)
print('Max document length: %d' % length)
# calculate vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary size: %d' % vocab_size)
# encode data
trainX = encode_text(tokenizer, trainLines, length)
testX = encode_text(tokenizer, testLines, length)
# load the model
model = load_model('model.h5')
# evaluate model on training dataset
_, acc = model.evaluate([trainX,trainX,trainX], trainLabels, verbose=0)
print('Train Accuracy: %.2f' % (acc*100))
# evaluate model on test dataset dataset
_, acc = model.evaluate([testX,testX,testX], testLabels, verbose=0)
print('Test Accuracy: %.2f' % (acc*100))
```

Listing 16.17: Complete example of evaluating the fit model.

Running the example prints the skill of the model on both the training and test datasets. We can see that, as expected, the skill on the training dataset is excellent, here at 100% accuracy. We can also see that the skill of the model on the unseen test dataset is also very impressive, achieving 88.5%, which is above the skill of the model reported in the 2014 paper (although not a direct apples-to-apples comparison).

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

```
Train Accuracy: 100.00
Test Accuracy: 88.50
```

Listing 16.18: Example output from evaluating the fit model.

16.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Different n-grams**. Explore the model by changing the kernel size (number of n-grams) used by the channels in the model to see how it impacts model skill.
- **More or Fewer Channels**. Explore using more or fewer channels in the model and see how it impacts model skill.
- **Shared Embedding**. Explore configurations where each channel shares the same word embedding and report on the impact on model skill.

- **Deeper Network**. Convolutional neural networks perform better in computer vision when they are deeper. Explore using deeper models here and see how it impacts model skill.
- **Truncated Sequences**. Padding all sequences to the length of the longest sequence might be extreme if the longest sequence is very different to all other reviews. Study the distribution of review lengths and truncate reviews to a mean length.
- **Truncated Vocabulary**. We removed infrequently occurring words, but still had a large vocabulary of more than 25,000 words. Explore further reducing the size of the vocabulary and the effect on model skill.
- **Epochs and Batch Size**. The model appears to fit the training dataset quickly. Explore alternate configurations of the number of training epochs and batch size and use the test dataset as a validation set to pick a better stopping point for training the model.
- **Pre-Train an Embedding**. Explore pre-training a Word2Vec word embedding in the model and the impact on model skill with and without further fine tuning during training.
- **Use GloVe Embedding**. Explore loading the pre-trained GloVe embedding and the impact on model skill with and without further fine tuning during training.
- **Train Final Model**. Train a final model on all available data and use it make predictions on real ad hoc movie reviews from the internet.

If you explore any of these extensions, I'd love to know.

16.7 Further Reading

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

- *Convolutional Neural Networks for Sentence Classification*, 2014.
  https://arxiv.org/abs/1408.5882
- Convolutional Neural Networks for Sentence Classification (code).
  https://github.com/yoonkim/CNN_sentence
- Keras Functional API.
  https://keras.io/getting-started/functional-api-guide/

16.8 Summary

In this tutorial, you discovered how to develop a multichannel convolutional neural network for sentiment prediction on text movie review data. Specifically, you learned:

- How to prepare movie review text data for modeling.
- How to develop a multichannel convolutional neural network for text in Keras.
- How to evaluate a fit model on unseen movie review data.

16.8.1 Next

This chapter is the last in the text classification part. In the next part, you will discover how to develop neural language models.

Part VII

Language Modeling

Chapter 17

Neural Language Modeling

Language modeling is central to many important natural language processing tasks. Recently, neural-network-based language models have demonstrated better performance than classical methods both standalone and as part of more challenging natural language processing tasks. In this chapter, you will discover language modeling for natural language processing. After reading this chapter, you will know:

- Why language modeling is critical to addressing tasks in natural language processing.
- What a language model is and some examples of where they are used.
- How neural networks can be used for language modeling.

Let's get started.

17.1 Overview

This tutorial is divided into the following parts:

1. Problem of Modeling Language
2. Statistical Language Modeling
3. Neural Language Models

17.2 Problem of Modeling Language

Formal languages, like programming languages, can be fully specified. All the reserved words can be defined and the valid ways that they can be used can be precisely defined. We cannot do this with natural language. Natural languages are not designed; they emerge, and therefore there is no formal specification.

There may be formal rules and heuristics for parts of the language, but as soon as rules are defined, you will devise or encounter counter examples that contradict the rules. Natural languages involve vast numbers of terms that can be used in ways that introduce all kinds of ambiguities, yet can still be understood by other humans. Further, languages change, word

usages change: it is a moving target. Nevertheless, linguists try to specify the language with formal grammars and structures. It can be done, but it is very difficult and the results can be fragile. An alternative approach to specifying the model of the language is to learn it from examples.

17.3 Statistical Language Modeling

Statistical Language Modeling, or Language Modeling and LM for short, is the development of probabilistic models that are able to predict the next word in the sequence given the words that precede it.

> Language modeling is the task of assigning a probability to sentences in a language. [...] Besides assigning a probability to each sequence of words, the language models also assigns a probability for the likelihood of a given word (or a sequence of words) to follow a sequence of words
>
> — Page 105, *Neural Network Methods in Natural Language Processing*, 2017.

A language model learns the probability of word occurrence based on examples of text. Simpler models may look at a context of a short sequence of words, whereas larger models may work at the level of sentences or paragraphs. Most commonly, language models operate at the level of words.

> The notion of a language model is inherently probabilistic. A language model is a function that puts a probability measure over strings drawn from some vocabulary.
>
> — Page 238, *An Introduction to Information Retrieval*, 2008.

A language model can be developed and used standalone, such as to generate new sequences of text that appear to have come from the corpus. Language modeling is a root problem for a large range of natural language processing tasks. More practically, language models are used on the front-end or back-end of a more sophisticated model for a task that requires language understanding.

> ... language modeling is a crucial component in real-world applications such as machine-translation and automatic speech recognition, [...] For these reasons, language modeling plays a central role in natural-language processing, AI, and machine-learning research.
>
> — Page 105, *Neural Network Methods in Natural Language Processing*, 2017.

A good example is speech recognition, where audio data is used as an input to the model and the output requires a language model that interprets the input signal and recognizes each new word within the context of the words already recognized.

> Speech recognition is principally concerned with the problem of transcribing the speech signal as a sequence of words. [...] From this point of view, speech is assumed to be a generated by a language model which provides estimates of Pr(w) for all word strings w independently of the observed signal [...] The goal of speech recognition is to find the most likely word sequence given the observed acoustic signal.

— Pages 205-206, *The Oxford Handbook of Computational Linguistics*, 2005

Similarly, language models are used to generate text in many similar natural language processing tasks, for example:

- Optical Character Recognition
- Handwriting Recognition.
- Machine Translation.
- Spelling Correction.
- Image Captioning.
- Text Summarization
- And much more.

> Language modeling is the art of determining the probability of a sequence of words. This is useful in a large variety of areas including speech recognition, optical character recognition, handwriting recognition, machine translation, and spelling correction

— *A Bit of Progress in Language Modeling*, 2001.

Developing better language models often results in models that perform better on their intended natural language processing task. This is the motivation for developing better and more accurate language models.

> [language models] have played a key role in traditional NLP tasks such as speech recognition, machine translation, or text summarization. Often (although not always), training better language models improves the underlying metrics of the downstream task (such as word error rate for speech recognition, or BLEU score for translation), which makes the task of training better LMs valuable by itself.

— *Exploring the Limits of Language Modeling*, 2016.

17.4 Neural Language Models

Recently, the use of neural networks in the development of language models has become very popular, to the point that it may now be the preferred approach. The use of neural networks in language modeling is often called Neural Language Modeling, or NLM for short. Neural network approaches are achieving better results than classical methods both on standalone language models and when models are incorporated into larger models on challenging tasks like speech recognition and machine translation. A key reason for the leaps in improved performance may be the method's ability to generalize.

> Nonlinear neural network models solve some of the shortcomings of traditional language models: they allow conditioning on increasingly large context sizes with only a linear increase in the number of parameters, they alleviate the need for manually designing backoff orders, and they support generalization across different contexts.
>
> — Page 109, *Neural Network Methods in Natural Language Processing*, 2017.

Specifically, a word embedding is adopted that uses a real-valued vector to represent each word in a projected vector space. This learned representation of words based on their usage allows words with a similar meaning to have a similar representation.

> Neural Language Models (NLM) address the n-gram data sparsity issue through parameterization of words as vectors (word embeddings) and using them as inputs to a neural network. The parameters are learned as part of the training process. Word embeddings obtained through NLMs exhibit the property whereby semantically close words are likewise close in the induced vector space.
>
> — *Character-Aware Neural Language Model*, 2015.

This generalization is something that the representation used in classical statistical language models cannot easily achieve.

> "True generalization" is difficult to obtain in a discrete word indice space, since there is no obvious relation between the word indices.
>
> — *Connectionist language modeling for large vocabulary continuous speech recognition*, 2002.

Further, the distributed representation approach allows the embedding representation to scale better with the size of the vocabulary. Classical methods that have one discrete representation per word fight the curse of dimensionality with larger and larger vocabularies of words that result in longer and more sparse representations. The neural network approach to language modeling can be described using the three following model properties, taken from *A Neural Probabilistic Language Model*, 2003.

1. Associate each word in the vocabulary with a distributed word feature vector.
2. Express the joint probability function of word sequences in terms of the feature vectors of these words in the sequence.
3. Learn simultaneously the word feature vector and the parameters of the probability function.

This represents a relatively simple model where both the representation and probabilistic model are learned together directly from raw text data. Recently, the neural based approaches have started to outperform the classical statistical approaches.

> We provide ample empirical evidence to suggest that connectionist language models are superior to standard n-gram techniques, except their high computational (training) complexity.

— *Recurrent neural network based language model*, 2010.

Initially, feedforward neural network models were used to introduce the approach. More recently, recurrent neural networks and then networks with a long-term memory like the Long Short-Term Memory network, or LSTM, allow the models to learn the relevant context over much longer input sequences than the simpler feedforward networks.

> [an RNN language model] provides further generalization: instead of considering just several preceding words, neurons with input from recurrent connections are assumed to represent short term memory. The model learns itself from the data how to represent memory. While shallow feedforward neural networks (those with just one hidden layer) can only cluster similar words, recurrent neural network (which can be considered as a deep architecture) can perform clustering of similar histories. This allows for instance efficient representation of patterns with variable length.

— *Extensions of recurrent neural network language model*, 2011.

Recently, researchers have been seeking the limits of these language models. In the paper *Exploring the Limits of Language Modeling*, evaluating language models over large datasets, such as the corpus of one million words, the authors find that LSTM-based neural language models out-perform the classical methods.

> ... we have shown that RNN LMs can be trained on large amounts of data, and outperform competing models including carefully tuned N-grams.

— *Exploring the Limits of Language Modeling*, 2016.

Further, they propose some heuristics for developing high-performing neural language models in general:

- **Size matters**. The best models were the largest models, specifically number of memory units.
- **Regularization matters**. Use of regularization like dropout on input connections improves results.
- **CNNs vs Embeddings**. Character-level Convolutional Neural Network (CNN) models can be used on the front-end instead of word embeddings, achieving similar and sometimes better results.
- **Ensembles matter**. Combining the prediction from multiple models can offer large improvements in model performance.

17.5 Further Reading

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

17.5.1 Books

- *Neural Network Methods in Natural Language Processing*, 2017.
  http://amzn.to/2vStiIS
- *Natural Language Processing, Artificial Intelligence A Modern Approach*, 2009.
  http://amzn.to/2fDPfF3
- *Language models for information retrieval, An Introduction to Information Retrieval*, 2008.
  http://amzn.to/2vAavQd

17.5.2 Papers

- *A Neural Probabilistic Language Model*, NIPS, 2001.
  https://papers.nips.cc/paper/1839-a-neural-probabilistic-language-model.pdf
- *A Neural Probabilistic Language Model*, JMLR, 2003.
  http://www.jmlr.org/papers/v3/bengio03a.html
- *Connectionist language modeling for large vocabulary continuous speech recognition*, 2002.
  https://pdfs.semanticscholar.org/b4db/83366f925e9a1e1528ee9f6b41d7cd666f41.pdf
- *Recurrent neural network based language model*, 2010.
  http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf
- *Extensions of recurrent neural network language model*, 2011.
  http://ieeexplore.ieee.org/abstract/document/5947611/
- *Character-Aware Neural Language Model*, 2015.
  https://arxiv.org/abs/1508.06615
- *LSTM Neural Networks for Language Modeling*, 2012.
  https://pdfs.semanticscholar.org/f9a1/b3850dfd837793743565a8af95973d395a4e.pdf
- *Exploring the Limits of Language Modeling*, 2016.
  https://arxiv.org/abs/1602.02410

17.5.3 Articles

- Language Model, Wikipedia.
  https://en.wikipedia.org/wiki/Language_model
- Neural net language models, Scholarpedia.
  http://www.scholarpedia.org/article/Neural_net_language_models

17.6 Summary

In this chapter, you discovered language modeling for natural language processing tasks. Specifically, you learned:

- That natural language is not formally specified and requires the use of statistical models to learn from examples.
- That statistical language models are central to many challenging natural language processing tasks.
- That state-of-the-art results are achieved using neural language models, specifically those with word embeddings and recurrent neural network algorithms.

17.6.1 Next

In the next chapter, you will discover how you can develop a character-based neural language model.

Chapter 18

How to Develop a Character-Based Neural Language Model

A language model predicts the next word in the sequence based on the specific words that have come before it in the sequence. It is also possible to develop language models at the character level using neural networks. The benefit of character-based language models is their small vocabulary and flexibility in handling any words, punctuation, and other document structure. This comes at the cost of requiring larger models that are slower to train. Nevertheless, in the field of neural language models, character-based models offer a lot of promise for a general, flexible and powerful approach to language modeling. In this tutorial, you will discover how to develop a character-based neural language model. After completing this tutorial, you will know:

- How to prepare text for character-based language modeling.
- How to develop a character-based language model using LSTMs.
- How to use a trained character-based language model to generate text.

Let's get started.

18.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Sing a Song of Sixpence
2. Data Preparation
3. Train Language Model
4. Generate Text

18.2 Sing a Song of Sixpence

The nursery rhyme *Sing a Song of Sixpence* is well known in the west. The first verse is common, but there is also a 4 verse version that we will use to develop our character-based language model. It is short, so fitting the model will be fast, but not so short that we won't see anything interesting. The complete 4 verse version we will use as source text is listed below.

```
Sing a song of sixpence,
A pocket full of rye.
Four and twenty blackbirds,
Baked in a pie.

When the pie was opened
The birds began to sing;
Wasn't that a dainty dish,
To set before the king.

The king was in his counting house,
Counting out his money;
The queen was in the parlour,
Eating bread and honey.

The maid was in the garden,
Hanging out the clothes,
When down came a blackbird
And pecked off her nose.
```

Listing 18.1: *Sing a Song of Sixpence* nursery rhyme.

Copy the text and save it in a new file in your current working directory with the file name `rhyme.txt`.

18.3 Data Preparation

The first step is to prepare the text data. We will start by defining the type of language model.

18.3.1 Language Model Design

A language model must be trained on the text, and in the case of a character-based language model, the input and output sequences must be characters. The number of characters used as input will also define the number of characters that will need to be provided to the model in order to elicit the first predicted character. After the first character has been generated, it can be appended to the input sequence and used as input for the model to generate the next character.

Longer sequences offer more context for the model to learn what character to output next but take longer to train and impose more burden on seeding the model when generating text. We will use an arbitrary length of 10 characters for this model. There is not a lot of text, and 10 characters is a few words. We can now transform the raw text into a form that our model can learn; specifically, input and output sequences of characters.

18.3.2 Load Text

We must load the text into memory so that we can work with it. Below is a function named `load_doc()` that will load a text file given a filename and return the loaded text.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text
```

Listing 18.2: Function to load a document into memory.

We can call this function with the filename of the nursery rhyme `rhyme.txt` to load the text into memory. The contents of the file are then printed to screen as a sanity check.

```
# load text
raw_text = load_doc('rhyme.txt')
print(raw_text)
```

Listing 18.3: Load the document into memory.

18.3.3 Clean Text

Next, we need to clean the loaded text. We will not do much to it on this example. Specifically, we will strip all of the new line characters so that we have one long sequence of characters separated only by white space.

```
# clean
tokens = raw_text.split()
raw_text = ' '.join(tokens)
```

Listing 18.4: Tokenize the loaded document.

You may want to explore other methods for data cleaning, such as normalizing the case to lowercase or removing punctuation in an effort to reduce the final vocabulary size and develop a smaller and leaner model.

18.3.4 Create Sequences

Now that we have a long list of characters, we can create our input-output sequences used to train the model. Each input sequence will be 10 characters with one output character, making each sequence 11 characters long. We can create the sequences by enumerating the characters in the text, starting at the 11th character at index 10.

```
# organize into sequences of characters
length = 10
sequences = list()
for i in range(length, len(raw_text)):
  # select sequence of tokens
  seq = raw_text[i-length:i+1]
```

```
  # store
  sequences.append(seq)
print('Total Sequences: %d' % len(sequences))
```

Listing 18.5: Convert text into fixed-length sequences.

Running this snippet, we can see that we end up with just under 400 sequences of characters for training our language model.

```
Total Sequences: 399
```

Listing 18.6: Example output of converting text into fixed-length sequences.

18.3.5 Save Sequences

Finally, we can save the prepared data to file so that we can load it later when we develop our model. Below is a function `save_doc()` that, given a list of strings and a filename, will save the strings to file, one per line.

```
# save tokens to file, one dialog per line
def save_doc(lines, filename):
  data = '\n'.join(lines)
  file = open(filename, 'w')
  file.write(data)
  file.close()
```

Listing 18.7: Function to save sequences to file.

We can call this function and save our prepared sequences to the filename `char_sequences.txt` in our current working directory.

```
# save sequences to file
out_filename = 'char_sequences.txt'
save_doc(sequences, out_filename)
```

Listing 18.8: Call function to save sequences to file.

18.3.6 Complete Example

Tying all of this together, the complete code listing is provided below.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# save tokens to file, one dialog per line
def save_doc(lines, filename):
  data = '\n'.join(lines)
  file = open(filename, 'w')
```

```
  file.write(data)
  file.close()

# load text
raw_text = load_doc('rhyme.txt')
print(raw_text)
# clean
tokens = raw_text.split()
raw_text = ' '.join(tokens)
# organize into sequences of characters
length = 10
sequences = list()
for i in range(length, len(raw_text)):
  # select sequence of tokens
  seq = raw_text[i-length:i+1]
  # store
  sequences.append(seq)
print('Total Sequences: %d' % len(sequences))
# save sequences to file
out_filename = 'char_sequences.txt'
save_doc(sequences, out_filename)
```

Listing 18.9: Complete example of preparing the text data.

Run the example to create the `char_sequences.txt` file. Take a look inside you should see something like the following:

```
Sing a song
ing a song
ng a song o
g a song of
 a song of
a song of s
 song of si
song of six
ong of sixp
ng of sixpe
...
```

Listing 18.10: Sample of the output file.

We are now ready to train our character-based neural language model.

18.4 Train Language Model

In this section, we will develop a neural language model for the prepared sequence data. The model will read encoded characters and predict the next character in the sequence. A Long Short-Term Memory recurrent neural network hidden layer will be used to learn the context from the input sequence in order to make the predictions.

18.4.1 Load Data

The first step is to load the prepared character sequence data from `char_sequences.txt`. We can use the same `load_doc()` function developed in the previous section. Once loaded, we split

the text by new line to give a list of sequences ready to be encoded.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# load
in_filename = 'char_sequences.txt'
raw_text = load_doc(in_filename)
lines = raw_text.split('\n')
```

Listing 18.11: Load the prepared text data.

18.4.2 Encode Sequences

The sequences of characters must be encoded as integers. This means that each unique character will be assigned a specific integer value and each sequence of characters will be encoded as a sequence of integers. We can create the mapping given a sorted set of unique characters in the raw input data. The mapping is a dictionary of character values to integer values.

```
chars = sorted(list(set(raw_text)))
mapping = dict((c, i) for i, c in enumerate(chars))
```

Listing 18.12: Create a mapping between chars and integers.

Next, we can process each sequence of characters one at a time and use the dictionary mapping to look up the integer value for each character.

```
sequences = list()
for line in lines:
  # integer encode line
  encoded_seq = [mapping[char] for char in line]
  # store
  sequences.append(encoded_seq)
```

Listing 18.13: Integer encode sequences of characters.

The result is a list of integer lists. We need to know the size of the vocabulary later. We can retrieve this as the size of the dictionary mapping.

```
# vocabulary size
vocab_size = len(mapping)
print('Vocabulary Size: %d' % vocab_size)
```

Listing 18.14: Summarize the size of the vocabulary.

Running this piece, we can see that there are 38 unique characters in the input sequence data.

```
Vocabulary Size: 38
```

Listing 18.15: Example output from summarizing the size of the vocabulary.

18.4.3 Split Inputs and Output

Now that the sequences have been integer encoded, we can separate the columns into input and output sequences of characters. We can do this using a simple array slice.

```
sequences = array(sequences)
X, y = sequences[:,:-1], sequences[:,-1]
```

Listing 18.16: Split sequences into input and output elements.

Next, we need to one hot encode each character. That is, each character becomes a vector as long as the vocabulary (38 elements) with a 1 marked for the specific character. This provides a more precise input representation for the network. It also provides a clear objective for the network to predict, where a probability distribution over characters can be output by the model and compared to the ideal case of all 0 values with a 1 for the actual next character. We can use the `to_categorical()` function in the Keras API to one hot encode the input and output sequences.

```
sequences = [to_categorical(x, num_classes=vocab_size) for x in X]
X = array(sequences)
y = to_categorical(y, num_classes=vocab_size)
```

Listing 18.17: Convert sequences into a format ready for training.

We are now ready to fit the model.

18.4.4 Fit Model

The model is defined with an input layer that takes sequences that have 10 time steps and 38 features for the one hot encoded input sequences. Rather than specify these numbers, we use the second and third dimensions on the X input data. This is so that if we change the length of the sequences or size of the vocabulary, we do not need to change the model definition. The model has a single LSTM hidden layer with 75 memory cells, chosen with a little trial and error. The model has a fully connected output layer that outputs one vector with a probability distribution across all characters in the vocabulary. A softmax activation function is used on the output layer to ensure the output has the properties of a probability distribution.

```
# define the model
def define_model(X):
  model = Sequential()
  model.add(LSTM(75, input_shape=(X.shape[1], X.shape[2])))
  model.add(Dense(vocab_size, activation='softmax'))
  # compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model
```

Listing 18.18: Define the language model.

The model is learning a multiclass classification problem, therefore we use the categorical log loss intended for this type of problem. The efficient Adam implementation of gradient descent is used to optimize the model and accuracy is reported at the end of each batch update. The

model is fit for 100 training epochs, again found with a little trial and error. Running this prints a summary of the defined network as a sanity check.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 75)                34200
_________________________________________________________________
dense_1 (Dense)              (None, 38)                2888
=================================================================
Total params: 37,088
Trainable params: 37,088
Non-trainable params: 0
_________________________________________________________________
```

Listing 18.19: Example output from summarizing the defined model.

A plot the defined model is then saved to file with the name `model.png`.

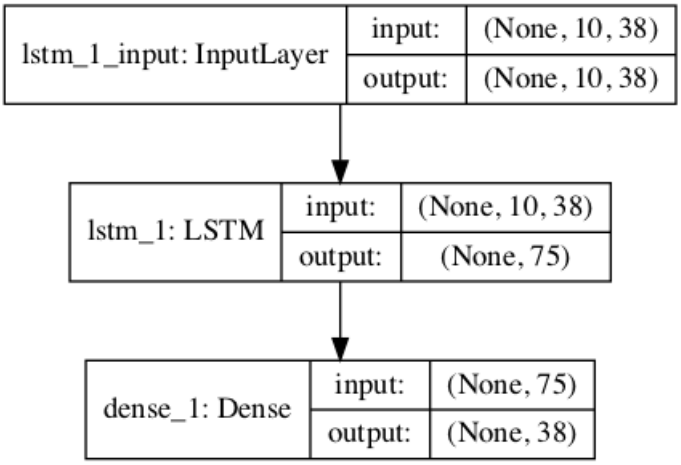

Figure 18.1: Plot of the defined character-based language model.

18.4.5 Save Model

After the model is fit, we save it to file for later use. The Keras model API provides the `save()` function that we can use to save the model to a single file, including weights and topology information.

```
# save the model to file
model.save('model.h5')
```

Listing 18.20: Save the fit model to file.

We also save the mapping from characters to integers that we will need to encode any input when using the model and decode any output from the model.

```
# save the mapping
dump(mapping, open('mapping.pkl', 'wb'))
```

Listing 18.21: Save the mapping of chars to integers to file.

18.4.6 Complete Example

Tying all of this together, the complete code listing for fitting the character-based neural language model is listed below.

```
from numpy import array
from pickle import dump
from keras.utils import to_categorical
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# define the model
def define_model(X):
  model = Sequential()
  model.add(LSTM(75, input_shape=(X.shape[1], X.shape[2])))
  model.add(Dense(vocab_size, activation='softmax'))
  # compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model

# load
in_filename = 'char_sequences.txt'
raw_text = load_doc(in_filename)
lines = raw_text.split('\n')
# integer encode sequences of characters
chars = sorted(list(set(raw_text)))
mapping = dict((c, i) for i, c in enumerate(chars))
sequences = list()
for line in lines:
  # integer encode line
  encoded_seq = [mapping[char] for char in line]
  # store
  sequences.append(encoded_seq)
# vocabulary size
vocab_size = len(mapping)
print('Vocabulary Size: %d' % vocab_size)
# separate into input and output
sequences = array(sequences)
X, y = sequences[:,:-1], sequences[:,-1]
sequences = [to_categorical(x, num_classes=vocab_size) for x in X]
X = array(sequences)
```

```
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(X)
# fit model
model.fit(X, y, epochs=100, verbose=2)
# save the model to file
model.save('model.h5')
# save the mapping
dump(mapping, open('mapping.pkl', 'wb'))
```

Listing 18.22: Complete example of training the language model.

Running the example might take one minute. You will see that the model learns the problem well, perhaps too well for generating surprising sequences of characters.

```
...
Epoch 96/100
0s - loss: 0.2193 - acc: 0.9950
Epoch 97/100
0s - loss: 0.2124 - acc: 0.9950
Epoch 98/100
0s - loss: 0.2054 - acc: 0.9950
Epoch 99/100
0s - loss: 0.1982 - acc: 0.9950
Epoch 100/100
0s - loss: 0.1910 - acc: 0.9950
```

Listing 18.23: Example output from training the language model.

At the end of the run, you will have two files saved to the current working directory, specifically `model.h5` and `mapping.pkl`. Next, we can look at using the learned model.

18.5 Generate Text

We will use the learned language model to generate new sequences of text that have the same statistical properties.

18.5.1 Load Model

The first step is to load the model saved to the file `model.h5`. We can use the `load_model()` function from the Keras API.

```
# load the model
model = load_model('model.h5')
```

Listing 18.24: Load the saved model.

We also need to load the pickled dictionary for mapping characters to integers from the file `mapping.pkl`. We will use the Pickle API to load the object.

```
# load the mapping
mapping = load(open('mapping.pkl', 'rb'))
```

Listing 18.25: Load the saved mapping from chars to integers.

We are now ready to use the loaded model.

18.5.2 Generate Characters

We must provide sequences of 10 characters as input to the model in order to start the generation process. We will pick these manually. A given input sequence will need to be prepared in the same way as preparing the training data for the model. First, the sequence of characters must be integer encoded using the loaded mapping.

```
# encode the characters as integers
encoded = [mapping[char] for char in in_text]
```

Listing 18.26: Encode input text to integers.

Next, the integers need to be one hot encoded using the `to_categorical()` Keras function. We also need to reshape the sequence to be 3-dimensional, as we only have one sequence and LSTMs require all input to be three dimensional (samples, time steps, features).

```
# one hot encode
encoded = to_categorical(encoded, num_classes=len(mapping))
encoded = encoded.reshape(1, encoded.shape[0], encoded.shape[1])
```

Listing 18.27: One hot encode the integer encoded text.

We can then use the model to predict the next character in the sequence. We use `predict_classes()` instead of `predict()` to directly select the integer for the character with the highest probability instead of getting the full probability distribution across the entire set of characters.

```
# predict character
yhat = model.predict_classes(encoded, verbose=0)
```

Listing 18.28: Predict the next character in the sequence.

We can then decode this integer by looking up the mapping to see the character to which it maps.

```
out_char = ''
for char, index in mapping.items():
  if index == yhat:
    out_char = char
    break
```

Listing 18.29: Map the predicted integer back to a character.

This character can then be added to the input sequence. We then need to make sure that the input sequence is 10 characters by truncating the first character from the input sequence text. We can use the `pad_sequences()` function from the Keras API that can perform this truncation operation. Putting all of this together, we can define a new function named `generate_seq()` for using the loaded model to generate new sequences of text.

```
# generate a sequence of characters with a language model
def generate_seq(model, mapping, seq_length, seed_text, n_chars):
  in_text = seed_text
  # generate a fixed number of characters
  for _ in range(n_chars):
    # encode the characters as integers
    encoded = [mapping[char] for char in in_text]
    # truncate sequences to a fixed length
```

```
    encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
    # one hot encode
    encoded = to_categorical(encoded, num_classes=len(mapping))
    encoded = encoded.reshape(1, encoded.shape[0], encoded.shape[1])
    # predict character
    yhat = model.predict_classes(encoded, verbose=0)
    # reverse map integer to character
    out_char = ''
    for char, index in mapping.items():
      if index == yhat:
        out_char = char
        break
    # append to input
    in_text += char
  return in_text
```

Listing 18.30: Function to predict a sequence of characters given seed text.

18.5.3 Complete Example

Tying all of this together, the complete example for generating text using the fit neural language model is listed below.

```
from pickle import load
from keras.models import load_model
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences

# generate a sequence of characters with a language model
def generate_seq(model, mapping, seq_length, seed_text, n_chars):
  in_text = seed_text
  # generate a fixed number of characters
  for _ in range(n_chars):
    # encode the characters as integers
    encoded = [mapping[char] for char in in_text]
    # truncate sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
    # one hot encode
    encoded = to_categorical(encoded, num_classes=len(mapping))
    encoded = encoded.reshape(1, encoded.shape[0], encoded.shape[1])
    # predict character
    yhat = model.predict_classes(encoded, verbose=0)
    # reverse map integer to character
    out_char = ''
    for char, index in mapping.items():
      if index == yhat:
        out_char = char
        break
    # append to input
    in_text += out_char
  return in_text

# load the model
model = load_model('model.h5')
# load the mapping
```

```
mapping = load(open('mapping.pkl', 'rb'))
# test start of rhyme
print(generate_seq(model, mapping, 10, 'Sing a son', 20))
# test mid-line
print(generate_seq(model, mapping, 10, 'king was i', 20))
# test not in original
print(generate_seq(model, mapping, 10, 'hello worl', 20))
```

Listing 18.31: Complete example of generating characters with the fit model.

Running the example generates three sequences of text. The first is a test to see how the model does at starting from the beginning of the rhyme. The second is a test to see how well it does at beginning in the middle of a line. The final example is a test to see how well it does with a sequence of characters never seen before.

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

```
Sing a song of sixpence, A poc
king was in his counting house
hello worls e pake wofey. The
```

Listing 18.32: Example output from generating sequences of characters.

We can see that the model did very well with the first two examples, as we would expect. We can also see that the model still generated something for the new text, but it is nonsense.

18.6 Further Reading

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

- Sing a Song of Sixpence on Wikipedia.
  https://en.wikipedia.org/wiki/Sing_a_Song_of_Sixpence
- Keras Utils API.
  https://keras.io/utils/
- Keras Sequence Processing API.
  https://keras.io/preprocessing/sequence/

18.7 Summary

In this tutorial, you discovered how to develop a character-based neural language model. Specifically, you learned:

- How to prepare text for character-based language modeling.
- How to develop a character-based language model using LSTMs.
- How to use a trained character-based language model to generate text.

18.7.1 Next

In the next chapter, you will discover how you can develop a word-based neural language model.

Chapter 19

How to Develop a Word-Based Neural Language Model

Language modeling involves predicting the next word in a sequence given the sequence of words already present. A language model is a key element in many natural language processing models such as machine translation and speech recognition. The choice of how the language model is framed must match how the language model is intended to be used. In this tutorial, you will discover how the framing of a language model affects the skill of the model when generating short sequences from a nursery rhyme. After completing this tutorial, you will know:

- The challenge of developing a good framing of a word-based language model for a given application.
- How to develop one-word, two-word, and line-based framings for word-based language models.
- How to generate sequences using a fit language model.

Let's get started.

19.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Framing Language Modeling
2. Jack and Jill Nursery Rhyme
3. Model 1: One-Word-In, One-Word-Out Sequences
4. Model 2: Line-by-Line Sequence
5. Model 3: Two-Words-In, One-Word-Out Sequence

19.2 Framing Language Modeling

A statistical language model is learned from raw text and predicts the probability of the next word in the sequence given the words already present in the sequence. Language models are a key component in larger models for challenging natural language processing problems, like machine translation and speech recognition. They can also be developed as standalone models and used for generating new sequences that have the same statistical properties as the source text.

Language models both learn and predict one word at a time. The training of the network involves providing sequences of words as input that are processed one at a time where a prediction can be made and learned for each input sequence. Similarly, when making predictions, the process can be seeded with one or a few words, then predicted words can be gathered and presented as input on subsequent predictions in order to build up a generated output sequence

Therefore, each model will involve splitting the source text into input and output sequences, such that the model can learn to predict words. There are many ways to frame the sequences from a source text for language modeling. In this tutorial, we will explore 3 different ways of developing word-based language models in the Keras deep learning library. There is no single best approach, just different framings that may suit different applications.

19.3 Jack and Jill Nursery Rhyme

Jack and Jill is a simple nursery rhyme. It is comprised of 4 lines, as follows:

```
Jack and Jill went up the hill
To fetch a pail of water
Jack fell down and broke his crown
And Jill came tumbling after
```

Listing 19.1: *Jack and Jill* nursery rhyme.

We will use this as our source text for exploring different framings of a word-based language model. We can define this text in Python as follows:

```
# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
    Jack fell down and broke his crown\n
    And Jill came tumbling after\n """
```

Listing 19.2: Sample text for this tutorial.

19.4 Model 1: One-Word-In, One-Word-Out Sequences

We can start with a very simple model. Given one word as input, the model will learn to predict the next word in the sequence. For example:

```
X,      y
Jack,   and
and,    Jill
Jill,   went
```

```
...
```

Listing 19.3: Example of input and output pairs.

The first step is to encode the text as integers. Each lowercase word in the source text is assigned a unique integer and we can convert the sequences of words to sequences of integers. Keras provides the `Tokenizer` class that can be used to perform this encoding. First, the `Tokenizer` is fit on the source text to develop the mapping from words to unique integers. Then sequences of text can be converted to sequences of integers by calling the `texts_to_sequences()` function.

```
# integer encode text
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
encoded = tokenizer.texts_to_sequences([data])[0]
```

Listing 19.4: Example of training a `Tokenizer` on the sample text.

We will need to know the size of the vocabulary later for both defining the word embedding layer in the model, and for encoding output words using a one hot encoding. The size of the vocabulary can be retrieved from the trained `Tokenizer` by accessing the `word_index` attribute.

```
# determine the vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
```

Listing 19.5: Summarize the size of the vocabulary.

Running this example, we can see that the size of the vocabulary is 21 words. We add one, because we will need to specify the integer for the largest encoded word as an array index, e.g. words encoded 1 to 21 with array indicies 0 to 21 or 22 positions. Next, we need to create sequences of words to fit the model with one word as input and one word as output.

```
# create word -> word sequences
sequences = list()
for i in range(1, len(encoded)):
  sequence = encoded[i-1:i+1]
  sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
```

Listing 19.6: Example of encoding the source text.

Running this piece shows that we have a total of 24 input-output pairs to train the network.

```
Total Sequences: 24
```

Listing 19.7: Example of output of summarizing the encoded text.

We can then split the sequences into input (`X`) and output elements (`y`). This is straightforward as we only have two columns in the data.

```
# split into X and y elements
sequences = array(sequences)
X, y = sequences[:,0],sequences[:,1]
```

Listing 19.8: Split the encoded text into input and output pairs.

We will fit our model to predict a probability distribution across all words in the vocabulary. That means that we need to turn the output element from a single integer into a one hot encoding with a 0 for every word in the vocabulary and a 1 for the actual word that the value. This gives the network a ground truth to aim for from which we can calculate error and update the model. Keras provides the `to_categorical()` function that we can use to convert the integer to a one hot encoding while specifying the number of classes as the vocabulary size.

```
# one hot encode outputs
y = to_categorical(y, num_classes=vocab_size)
```

Listing 19.9: One hot encode the output words.

We are now ready to define the neural network model. The model uses a learned word embedding in the input layer. This has one real-valued vector for each word in the vocabulary, where each word vector has a specified length. In this case we will use a 10-dimensional projection. The input sequence contains a single word, therefore the `input_length=1`. The model has a single hidden LSTM layer with 50 units. This is far more than is needed. The output layer is comprised of one neuron for each word in the vocabulary and uses a softmax activation function to ensure the output is normalized to look like a probability.

```
# define the model
def define_model(vocab_size):
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=1))
  model.add(LSTM(50))
  model.add(Dense(vocab_size, activation='softmax'))
  # compile network
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model
```

Listing 19.10: Define and compile the language model.

The structure of the network can be summarized as follows:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 1, 10)             220
_________________________________________________________________
lstm_1 (LSTM)                (None, 50)                12200
_________________________________________________________________
dense_1 (Dense)              (None, 22)                1122
=================================================================
Total params: 13,542
Trainable params: 13,542
Non-trainable params: 0
_________________________________________________________________
```

Listing 19.11: Example output summarizing the defined model.

A plot the defined model is then saved to file with the name `model.png`.

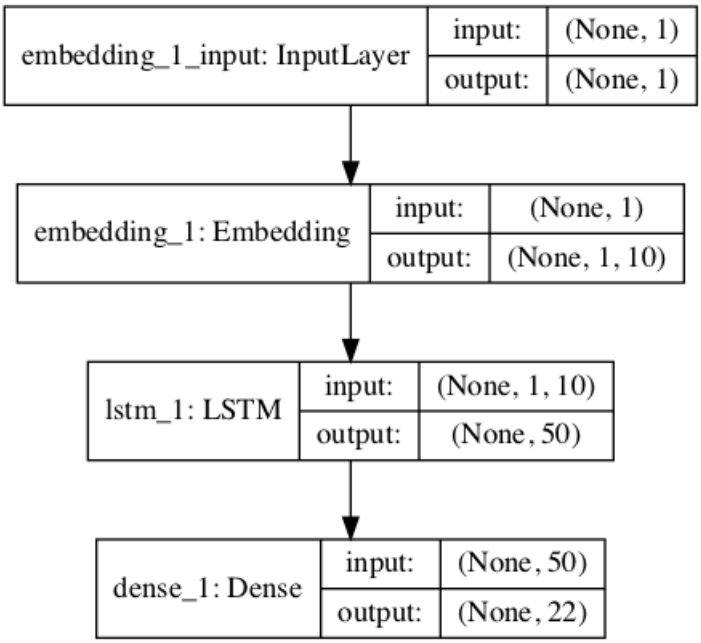

Figure 19.1: Plot of the defined word-based language model.

We will use this same general network structure for each example in this tutorial, with minor changes to the learned embedding layer. We can compile and fit the network on the encoded text data. Technically, we are modeling a multiclass classification problem (predict the word in the vocabulary), therefore using the categorical cross entropy loss function. We use the efficient Adam implementation of gradient descent and track accuracy at the end of each epoch. The model is fit for 500 training epochs, again, perhaps more than is needed. The network configuration was not tuned for this and later experiments; an over-prescribed configuration was chosen to ensure that we could focus on the framing of the language model.

After the model is fit, we test it by passing it a given word from the vocabulary and having the model predict the next word. Here we pass in '*Jack*' by encoding it and calling `model.predict_classes()` to get the integer output for the predicted word. This is then looked up in the vocabulary mapping to give the associated word.

```
# evaluate
in_text = 'Jack'
print(in_text)
encoded = tokenizer.texts_to_sequences([in_text])[0]
encoded = array(encoded)
yhat = model.predict_classes(encoded, verbose=0)
for word, index in tokenizer.word_index.items():
  if index == yhat:
    print(word)
```

Listing 19.12: Evaluate the fit language model.

This process could then be repeated a few times to build up a generated sequence of words. To make this easier, we wrap up the behavior in a function that we can call by passing in our model and the seed word.

```
# generate a sequence from the model
def generate_seq(model, tokenizer, seed_text, n_words):
```

```
  in_text, result = seed_text, seed_text
  # generate a fixed number of words
  for _ in range(n_words):
    # encode the text as integer
    encoded = tokenizer.texts_to_sequences([in_text])[0]
    encoded = array(encoded)
    # predict a word in the vocabulary
    yhat = model.predict_classes(encoded, verbose=0)
    # map predicted word index to word
    out_word = ''
    for word, index in tokenizer.word_index.items():
      if index == yhat:
        out_word = word
        break
    # append to input
    in_text, result = out_word, result + ' ' + out_word
  return result
```

Listing 19.13: Function to generate output sequences given a fit model.

We can tie all of this together. The complete code listing is provided below.

```
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.utils import to_categorical
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding

# generate a sequence from the model
def generate_seq(model, tokenizer, seed_text, n_words):
  in_text, result = seed_text, seed_text
  # generate a fixed number of words
  for _ in range(n_words):
    # encode the text as integer
    encoded = tokenizer.texts_to_sequences([in_text])[0]
    encoded = array(encoded)
    # predict a word in the vocabulary
    yhat = model.predict_classes(encoded, verbose=0)
    # map predicted word index to word
    out_word = ''
    for word, index in tokenizer.word_index.items():
      if index == yhat:
        out_word = word
        break
    # append to input
    in_text, result = out_word, result + ' ' + out_word
  return result

# define the model
def define_model(vocab_size):
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=1))
  model.add(LSTM(50))
  model.add(Dense(vocab_size, activation='softmax'))
```

```
  # compile network
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model

# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
    Jack fell down and broke his crown\n
    And Jill came tumbling after\n """
# integer encode text
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
encoded = tokenizer.texts_to_sequences([data])[0]
# determine the vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# create word -> word sequences
sequences = list()
for i in range(1, len(encoded)):
  sequence = encoded[i-1:i+1]
  sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
# split into X and y elements
sequences = array(sequences)
X, y = sequences[:,0],sequences[:,1]
# one hot encode outputs
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(vocab_size)
# fit network
model.fit(X, y, epochs=500, verbose=2)
# evaluate
print(generate_seq(model, tokenizer, 'Jack', 6))
```

Listing 19.14: Complete example of model1.

Running the example prints the loss and accuracy each training epoch.

```
...
Epoch 496/500
0s - loss: 0.2358 - acc: 0.8750
Epoch 497/500
0s - loss: 0.2355 - acc: 0.8750
Epoch 498/500
0s - loss: 0.2352 - acc: 0.8750
Epoch 499/500
0s - loss: 0.2349 - acc: 0.8750
Epoch 500/500
0s - loss: 0.2346 - acc: 0.8750
```

Listing 19.15: Example output of fitting the language model.

We can see that the model does not memorize the source sequences, likely because there is some ambiguity in the input sequences, for example:

```
jack => and
jack => fell
```

Listing 19.16: Example output of predicting the next word.

And so on. At the end of the run, *Jack* is passed in and a prediction or new sequence is generated. We get a reasonable sequence as output that has some elements of the source.

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

```
Jack and jill came tumbling after down
```

Listing 19.17: Example output of predicting a sequence of words.

This is a good first cut language model, but does not take full advantage of the LSTM's ability to handle sequences of input and disambiguate some of the ambiguous pairwise sequences by using a broader context.

19.5 Model 2: Line-by-Line Sequence

Another approach is to split up the source text line-by-line, then break each line down into a series of words that build up. For example:

```
X,                              y
_, _, _, _, _, Jack,            and
_, _, _, _, Jack, and,          Jill
_, _, _, Jack, and, Jill,       went
_, _, Jack, and, Jill, went,    up
_, Jack, and, Jill, went, up,   the
Jack, and, Jill, went, up, the, hill
```

Listing 19.18: Example framing of the problem as sequences of words.

This approach may allow the model to use the context of each line to help the model in those cases where a simple one-word-in-and-out model creates ambiguity. In this case, this comes at the cost of predicting words across lines, which might be fine for now if we are only interested in modeling and generating lines of text. Note that in this representation, we will require a padding of sequences to ensure they meet a fixed length input. This is a requirement when using Keras. First, we can create the sequences of integers, line-by-line by using the `Tokenizer` already fit on the source text.

```
# create line-based sequences
sequences = list()
for line in data.split('\n'):
  encoded = tokenizer.texts_to_sequences([line])[0]
  for i in range(1, len(encoded)):
    sequence = encoded[:i+1]
    sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
```

Listing 19.19: Example of preparing sequences of words.

Next, we can pad the prepared sequences. We can do this using the `pad_sequences()` function provided in Keras. This first involves finding the longest sequence, then using that as the length by which to pad-out all other sequences.

```
# pad input sequences
max_length = max([len(seq) for seq in sequences])
sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
print('Max Sequence Length: %d' % max_length)
```

Listing 19.20: Example of padding sequences of words.

Next, we can split the sequences into input and output elements, much like before.

```
# split into input and output elements
sequences = array(sequences)
X, y = sequences[:,:-1],sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
```

Listing 19.21: Example of preparing the input and output sequences.

The model can then be defined as before, except the input sequences are now longer than a single word. Specifically, they are `max_length-1` in length, -1 because when we calculated the maximum length of sequences, they included the input and output elements.

```
# define the model
def define_model(vocab_size, max_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=max_length-1))
  model.add(LSTM(50))
  model.add(Dense(vocab_size, activation='softmax'))
  # compile network
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model
```

Listing 19.22: Define and compile the language model.

We can use the model to generate new sequences as before. The `generate_seq()` function can be updated to build up an input sequence by adding predictions to the list of input words each iteration.

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, max_length, seed_text, n_words):
  in_text = seed_text
  # generate a fixed number of words
  for _ in range(n_words):
    # encode the text as integer
    encoded = tokenizer.texts_to_sequences([in_text])[0]
    # pre-pad sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=max_length, padding='pre')
    # predict probabilities for each word
    yhat = model.predict_classes(encoded, verbose=0)
    # map predicted word index to word
    out_word = ''
    for word, index in tokenizer.word_index.items():
      if index == yhat:
```

```
        out_word = word
        break
    # append to input
    in_text += ' ' + out_word
  return in_text
```

Listing 19.23: Function to generate sequences of words given input text.

Tying all of this together, the complete code example is provided below.

```
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding

# generate a sequence from a language model
def generate_seq(model, tokenizer, max_length, seed_text, n_words):
  in_text = seed_text
  # generate a fixed number of words
  for _ in range(n_words):
    # encode the text as integer
    encoded = tokenizer.texts_to_sequences([in_text])[0]
    # pre-pad sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=max_length, padding='pre')
    # predict probabilities for each word
    yhat = model.predict_classes(encoded, verbose=0)
    # map predicted word index to word
    out_word = ''
    for word, index in tokenizer.word_index.items():
      if index == yhat:
        out_word = word
        break
    # append to input
    in_text += ' ' + out_word
  return in_text

# define the model
def define_model(vocab_size, max_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=max_length-1))
  model.add(LSTM(50))
  model.add(Dense(vocab_size, activation='softmax'))
  # compile network
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model

# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
```

```
    Jack fell down and broke his crown\n
    And Jill came tumbling after\n """
# prepare the tokenizer on the source text
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
# determine the vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# create line-based sequences
sequences = list()
for line in data.split('\n'):
  encoded = tokenizer.texts_to_sequences([line])[0]
  for i in range(1, len(encoded)):
    sequence = encoded[:i+1]
    sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
# pad input sequences
max_length = max([len(seq) for seq in sequences])
sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
print('Max Sequence Length: %d' % max_length)
# split into input and output elements
sequences = array(sequences)
X, y = sequences[:,:-1],sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(vocab_size, max_length)
# fit network
model.fit(X, y, epochs=500, verbose=2)
# evaluate model
print(generate_seq(model, tokenizer, max_length-1, 'Jack', 4))
print(generate_seq(model, tokenizer, max_length-1, 'Jill', 4))
```

Listing 19.24: Complete example of model2.

Running the example achieves a better fit on the source data. The added context has allowed the model to disambiguate some of the examples. There are still two lines of text that start with "*Jack*" that may still be a problem for the network.

```
...
Epoch 496/500
0s - loss: 0.1039 - acc: 0.9524
Epoch 497/500
0s - loss: 0.1037 - acc: 0.9524
Epoch 498/500
0s - loss: 0.1035 - acc: 0.9524
Epoch 499/500
0s - loss: 0.1033 - acc: 0.9524
Epoch 500/500
0s - loss: 0.1032 - acc: 0.9524
```

Listing 19.25: Example output of fitting the language model.

At the end of the run, we generate two sequences with different seed words: *Jack* and *Jill*.

The first generated line looks good, directly matching the source text. The second is a bit strange. This makes sense, because the network only ever saw *Jill* within an input sequence, not at the beginning of the sequence, so it has forced an output to use the word *Jill*, i.e. the

last line of the rhyme.

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

```
Jack fell down and broke
Jill jill came tumbling after
```

Listing 19.26: Example output of generating sequences of words.

This was a good example of how the framing may result in better new lines, but not good partial lines of input.

19.6 Model 3: Two-Words-In, One-Word-Out Sequence

We can use an intermediate between the one-word-in and the whole-sentence-in approaches and pass in a sub-sequences of words as input. This will provide a trade-off between the two framings allowing new lines to be generated and for generation to be picked up mid line. We will use 3 words as input to predict one word as output. The preparation of the sequences is much like the first example, except with different offsets in the source sequence arrays, as follows:

```
# encode 2 words -> 1 word
sequences = list()
for i in range(2, len(encoded)):
  sequence = encoded[i-2:i+1]
  sequences.append(sequence)
```

Listing 19.27: Example of preparing constrained sequence data.

The complete example is listed below

```
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding

# generate a sequence from a language model
def generate_seq(model, tokenizer, max_length, seed_text, n_words):
  in_text = seed_text
  # generate a fixed number of words
  for _ in range(n_words):
    # encode the text as integer
    encoded = tokenizer.texts_to_sequences([in_text])[0]
    # pre-pad sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=max_length, padding='pre')
    # predict probabilities for each word
    yhat = model.predict_classes(encoded, verbose=0)
    # map predicted word index to word
    out_word = ''
```

```
    for word, index in tokenizer.word_index.items():
      if index == yhat:
        out_word = word
        break
    # append to input
    in_text += ' ' + out_word
  return in_text

# define the model
def define_model(vocab_size, max_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=max_length-1))
  model.add(LSTM(50))
  model.add(Dense(vocab_size, activation='softmax'))
  # compile network
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model

# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
    Jack fell down and broke his crown\n
    And Jill came tumbling after\n """
# integer encode sequences of words
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
encoded = tokenizer.texts_to_sequences([data])[0]
# retrieve vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# encode 2 words -> 1 word
sequences = list()
for i in range(2, len(encoded)):
  sequence = encoded[i-2:i+1]
  sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
# pad sequences
max_length = max([len(seq) for seq in sequences])
sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
print('Max Sequence Length: %d' % max_length)
# split into input and output elements
sequences = array(sequences)
X, y = sequences[:,:-1],sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(vocab_size, max_length)
# fit network
model.fit(X, y, epochs=500, verbose=2)
# evaluate model
print(generate_seq(model, tokenizer, max_length-1, 'Jack and', 5))
print(generate_seq(model, tokenizer, max_length-1, 'And Jill', 3))
print(generate_seq(model, tokenizer, max_length-1, 'fell down', 5))
print(generate_seq(model, tokenizer, max_length-1, 'pail of', 5))
```

Listing 19.28: Complete example of model3.

Running the example again gets a good fit on the source text at around 95% accuracy.

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

```
...
Epoch 496/500
0s - loss: 0.0685 - acc: 0.9565
Epoch 497/500
0s - loss: 0.0685 - acc: 0.9565
Epoch 498/500
0s - loss: 0.0684 - acc: 0.9565
Epoch 499/500
0s - loss: 0.0684 - acc: 0.9565
Epoch 500/500
0s - loss: 0.0684 - acc: 0.9565
```

Listing 19.29: Example output of fitting the language model.

We look at 4 generation examples, two start of line cases and two starting mid line.

```
Jack and jill went up the hill
And Jill went up the
fell down and broke his crown and
pail of water jack fell down and
```

Listing 19.30: Example output of generating sequences of words.

The first start of line case generated correctly, but the second did not. The second case was an example from the 4th line, which is ambiguous with content from the first line. Perhaps a further expansion to 3 input words would be better. The two mid-line generation examples were generated correctly, matching the source text.

We can see that the choice of how the language model is framed and the requirements on how the model will be used must be compatible. That careful design is required when using language models in general, perhaps followed-up by spot testing with sequence generation to confirm model requirements have been met.

19.7 Further Reading

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

- Jack and Jill on Wikipedia.
  https://en.wikipedia.org/wiki/Jack_and_Jill_(nursery_rhyme)
- Language Model on Wikpedia.
  https://en.wikipedia.org/wiki/Language_model
- Keras Embedding Layer API.
  https://keras.io/layers/embeddings/#embedding

- Keras Text Processing API.
  https://keras.io/preprocessing/text/
- Keras Sequence Processing API.
  https://keras.io/preprocessing/sequence/
- Keras Utils API.
  https://keras.io/utils/

19.8 Summary

In this tutorial, you discovered how to develop different word-based language models for a simple nursery rhyme. Specifically, you learned:

- The challenge of developing a good framing of a word-based language model for a given application.
- How to develop one-word, two-word, and line-based framings for word-based language models.
- How to generate sequences using a fit language model.

19.8.1 Next

In the next chapter, you will discover how you can develop a word-based neural language model on a large corpus of text.

Chapter 20

Project: Develop a Neural Language Model for Text Generation

A language model can predict the probability of the next word in the sequence, based on the words already observed in the sequence. Neural network models are a preferred method for developing statistical language models because they can use a distributed representation where different words with similar meanings have similar representation and because they can use a large context of recently observed words when making predictions. In this tutorial, you will discover how to develop a statistical language model using deep learning in Python. After completing this tutorial, you will know:

- How to prepare text for developing a word-based language model.
- How to design and fit a neural language model with a learned embedding and an LSTM hidden layer.
- How to use the learned language model to generate new text with similar statistical properties as the source text.

Let's get started.

20.1 Tutorial Overview

This tutorial is divided into the following parts:

1. The Republic by Plato
2. Data Preparation
3. Train Language Model
4. Use Language Model

20.2 The Republic by Plato

The Republic is the classical Greek philosopher Plato's most famous work. It is structured as a dialog (e.g. conversation) on the topic of order and justice within a city state The entire text is available for free in the public domain. It is available on the Project Gutenberg website in a number of formats. You can download the ASCII text version of the entire book (or books) here (you might need to open the URL twice):

- Download The Republic by Plato.
  http://www.gutenberg.org/cache/epub/1497/pg1497.txt

Download the book text and place it in your current working directly with the filename `republic.txt`. Open the file in a text editor and delete the front and back matter. This includes details about the book at the beginning, a long analysis, and license information at the end. The text should begin with:

> BOOK I.
>
> I went down yesterday to the Piraeus with Glaucon the son of Ariston, ...

And end with:

> ... And it shall be well with us both in this life and in the pilgrimage of a thousand years which we have been describing.

Save the cleaned version as `republic_clean.txt` in your current working directory. The file should be about 15,802 lines of text. Now we can develop a language model from this text.

20.3 Data Preparation

We will start by preparing the data for modeling. The first step is to look at the data.

20.3.1 Review the Text

Open the text in an editor and just look at the text data. For example, here is the first piece of dialog:

> BOOK I.
>
> I went down yesterday to the Piraeus with Glaucon the son of Ariston, that I might offer up my prayers to the goddess (Bendis, the Thracian Artemis.); and also because I wanted to see in what manner they would celebrate the festival, which was a new thing. I was delighted with the procession of the inhabitants; but that of the Thracians was equally, if not more, beautiful. When we had finished our prayers and viewed the spectacle, we turned in the direction of the city; and at that instant Polemarchus the son of Cephalus chanced to catch sight of us from a distance as we were starting on our way home, and told his servant to run and bid us wait for him. The servant took hold of me by the cloak behind, and said: Polemarchus desires you to wait.

> I turned round, and asked him where his master was.
>
> There he is, said the youth, coming after you, if you will only wait.
>
> Certainly we will, said Glaucon; and in a few minutes Polemarchus appeared, and with him Adeimantus, Glaucon's brother, Niceratus the son of Nicias, and several others who had been at the procession.
>
> Polemarchus said to me: I perceive, Socrates, that you and your companion are already on your way to the city.
>
> You are not far wrong, I said.
>
> ...

What do you see that we will need to handle in preparing the data? Here's what I see from a quick look:

- Book/Chapter headings (e.g. *BOOK I.*).
- Lots of punctuation (e.g. -, *;*-, *?*-, and more).
- Strange names (e.g. *Polemarchus*).
- Some long monologues that go on for hundreds of lines.
- Some quoted dialog (e.g. '...').

These observations, and more, suggest at ways that we may wish to prepare the text data. The specific way we prepare the data really depends on how we intend to model it, which in turn depends on how we intend to use it.

20.3.2 Language Model Design

In this tutorial, we will develop a model of the text that we can then use to generate new sequences of text. The language model will be statistical and will predict the probability of each word given an input sequence of text. The predicted word will be fed in as input to in turn generate the next word. A key design decision is how long the input sequences should be. They need to be long enough to allow the model to learn the context for the words to predict. This input length will also define the length of seed text used to generate new sequences when we use the model.

There is no correct answer. With enough time and resources, we could explore the ability of the model to learn with differently sized input sequences. Instead, we will pick a length of 50 words for the length of the input sequences, somewhat arbitrarily. We could process the data so that the model only ever deals with self-contained sentences and pad or truncate the text to meet this requirement for each input sequence. You could explore this as an extension to this tutorial.

Instead, to keep the example brief, we will let all of the text flow together and train the model to predict the next word across sentences, paragraphs, and even books or chapters in the text. Now that we have a model design, we can look at transforming the raw text into sequences of 100 input words to 1 output word, ready to fit a model.

20.3.3 Load Text

The first step is to load the text into memory. We can develop a small function to load the entire text file into memory and return it. The function is called `load_doc()` and is listed below. Given a filename, it returns a sequence of loaded text.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text
```

Listing 20.1: Function to load text into memory.

Using this function, we can load the cleaner version of the document in the file `republic_clean.txt` as follows:

```
# load document
in_filename = 'republic_clean.txt'
doc = load_doc(in_filename)
print(doc[:200])
```

Listing 20.2: Example of loading the text into memory.

Running this snippet loads the document and prints the first 200 characters as a sanity check.

```
BOOK I.

I went down yesterday to the Piraeus with Glaucon the son of Ariston,
that I might offer up my prayers to the goddess (Bendis, the Thracian
Artemis.); and also because I wanted to see in what
```

Listing 20.3: Example output of loading the text into memory.

So far, so good. Next, let's clean the text.

20.3.4 Clean Text

We need to transform the raw text into a sequence of tokens or words that we can use as a source to train the model. Based on reviewing the raw text (above), below are some specific operations we will perform to clean the text. You may want to explore more cleaning operations yourself as an extension.

- Replace '-' with a white space so we can split words better.
- Split words based on white space.
- Remove all punctuation from words to reduce the vocabulary size (e.g. 'What?' becomes 'What').
- Remove all words that are not alphabetic to remove standalone punctuation tokens.

- Normalize all words to lowercase to reduce the vocabulary size.

Vocabulary size is a big deal with language modeling. A smaller vocabulary results in a smaller model that trains faster. We can implement each of these cleaning operations in this order in a function. Below is the function `clean_doc()` that takes a loaded document as an argument and returns an array of clean tokens.

```
# turn a doc into clean tokens
def clean_doc(doc):
  # replace '--' with a space ' '
  doc = doc.replace('--', ' ')
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # make lower case
  tokens = [word.lower() for word in tokens]
  return tokens
```

Listing 20.4: Function to clean text.

We can run this cleaning operation on our loaded document and print out some of the tokens and statistics as a sanity check.

```
# clean document
tokens = clean_doc(doc)
print(tokens[:200])
print('Total Tokens: %d' % len(tokens))
print('Unique Tokens: %d' % len(set(tokens)))
```

Listing 20.5: Example of cleaning text.

First, we can see a nice list of tokens that look cleaner than the raw text. We could remove the 'Book I' chapter markers and more, but this is a good start.

```
['book', 'i', 'i', 'went', 'down', 'yesterday', 'to', 'the', 'piraeus', 'with', 'glaucon',
    'the', 'son', 'of', 'ariston', 'that', 'i', 'might', 'offer', 'up', 'my', 'prayers',
    'to', 'the', 'goddess', 'bendis', 'the', 'thracian', 'artemis', 'and', 'also',
    'because', 'i', 'wanted', 'to', 'see', 'in', 'what', 'manner', 'they', 'would',
    'celebrate', 'the', 'festival', 'which', 'was', 'a', 'new', 'thing', 'i', 'was',
    'delighted', 'with', 'the', 'procession', 'of', 'the', 'inhabitants', 'but', 'that',
    'of', 'the', 'thracians', 'was', 'equally', 'if', 'not', 'more', 'beautiful', 'when',
    'we', 'had', 'finished', 'our', 'prayers', 'and', 'viewed', 'the', 'spectacle', 'we',
    'turned', 'in', 'the', 'direction', 'of', 'the', 'city', 'and', 'at', 'that',
    'instant', 'polemarchus', 'the', 'son', 'of', 'cephalus', 'chanced', 'to', 'catch',
    'sight', 'of', 'us', 'from', 'a', 'distance', 'as', 'we', 'were', 'starting', 'on',
    'our', 'way', 'home', 'and', 'told', 'his', 'servant', 'to', 'run', 'and', 'bid', 'us',
    'wait', 'for', 'him', 'the', 'servant', 'took', 'hold', 'of', 'me', 'by', 'the',
    'cloak', 'behind', 'and', 'said', 'polemarchus', 'desires', 'you', 'to', 'wait', 'i',
    'turned', 'round', 'and', 'asked', 'him', 'where', 'his', 'master', 'was', 'there',
    'he', 'is', 'said', 'the', 'youth', 'coming', 'after', 'you', 'if', 'you', 'will',
    'only', 'wait', 'certainly', 'we', 'will', 'said', 'glaucon', 'and', 'in', 'a', 'few',
    'minutes', 'polemarchus', 'appeared', 'and', 'with', 'him', 'adeimantus', 'glaucons',
```

```
    'brother', 'niceratus', 'the', 'son', 'of', 'nicias', 'and', 'several', 'others',
    'who', 'had', 'been', 'at', 'the', 'procession', 'polemarchus', 'said']
```

Listing 20.6: Example output of tokenized and clean text.

We also get some statistics about the clean document. We can see that there are just under 120,000 words in the clean text and a vocabulary of just under 7,500 words. This is smallish and models fit on this data should be manageable on modest hardware.

```
Total Tokens: 118684
Unique Tokens: 7409
```

Listing 20.7: Example output summarizing properties of the clean text.

Next, we can look at shaping the tokens into sequences and saving them to file.

20.3.5 Save Clean Text

We can organize the long list of tokens into sequences of 50 input words and 1 output word. That is, sequences of 51 words. We can do this by iterating over the list of tokens from token 51 onwards and taking the prior 50 tokens as a sequence, then repeating this process to the end of the list of tokens. We will transform the tokens into space-separated strings for later storage in a file. The code to split the list of clean tokens into sequences with a length of 51 tokens is listed below.

```
# organize into sequences of tokens
length = 50 + 1
sequences = list()
for i in range(length, len(tokens)):
  # select sequence of tokens
  seq = tokens[i-length:i]
  # convert into a line
  line = ' '.join(seq)
  # store
  sequences.append(line)
print('Total Sequences: %d' % len(sequences))
```

Listing 20.8: Split document into sequences of text.

Running this piece creates a long list of lines. Printing statistics on the list, we can see that we will have exactly 118,633 training patterns to fit our model.

```
Total Sequences: 118633
```

Listing 20.9: Example output of splitting the document into sequences.

Next, we can save the sequences to a new file for later loading. We can define a new function for saving lines of text to a file. This new function is called `save_doc()` and is listed below. It takes as input a list of lines and a filename. The lines are written, one per line, in ASCII format.

```
# save tokens to file, one dialog per line
def save_doc(lines, filename):
  data = '\n'.join(lines)
  file = open(filename, 'w')
  file.write(data)
  file.close()
```

Listing 20.10: Function to save sequences of text to file.

We can call this function and save our training sequences to the file `republic_sequences.txt`.

```
# save sequences to file
out_filename = 'republic_sequences.txt'
save_doc(sequences, out_filename)
```

Listing 20.11: Example of saving sequences to file.

Take a look at the file with your text editor. You will see that each line is shifted along one word, with a new word at the end to be predicted; for example, here are the first 3 lines in truncated form:

```
book i i ... catch sight of
i i went ... sight of us
i went down ... of us from
...
```

Listing 20.12: Example contents of sequences saved to file.

20.3.6 Complete Example

Tying all of this together, the complete code listing is provided below.

```
import string
import re

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# turn a doc into clean tokens
def clean_doc(doc):
  # replace '--' with a space ' '
  doc = doc.replace('--', ' ')
  # split into tokens by white space
  tokens = doc.split()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  # remove punctuation from each word
  tokens = [re_punc.sub('', w) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # make lower case
  tokens = [word.lower() for word in tokens]
  return tokens

# save tokens to file, one dialog per line
```

```
def save_doc(lines, filename):
  data = '\n'.join(lines)
  file = open(filename, 'w')
  file.write(data)
  file.close()

# load document
in_filename = 'republic_clean.txt'
doc = load_doc(in_filename)
print(doc[:200])
# clean document
tokens = clean_doc(doc)
print(tokens[:200])
print('Total Tokens: %d' % len(tokens))
print('Unique Tokens: %d' % len(set(tokens)))
# organize into sequences of tokens
length = 50 + 1
sequences = list()
for i in range(length, len(tokens)):
  # select sequence of tokens
  seq = tokens[i-length:i]
  # convert into a line
  line = ' '.join(seq)
  # store
  sequences.append(line)
print('Total Sequences: %d' % len(sequences))
# save sequences to file
out_filename = 'republic_sequences.txt'
save_doc(sequences, out_filename)
```

Listing 20.13: Complete example preparing text data for modeling.

You should now have training data stored in the file `republic_sequences.txt` in your current working directory. Next, let's look at how to fit a language model to this data.

20.4 Train Language Model

We can now train a statistical language model from the prepared data. The model we will train is a neural language model. It has a few unique characteristics:

- It uses a distributed representation for words so that different words with similar meanings will have a similar representation.
- It learns the representation at the same time as learning the model.
- It learns to predict the probability for the next word using the context of the last 100 words.

Specifically, we will use an Embedding Layer to learn the representation of words, and a Long Short-Term Memory (LSTM) recurrent neural network to learn to predict words based on their context. Let's start by loading our training data.

20.4.1 Load Sequences

We can load our training data using the `load_doc()` function we developed in the previous section. Once loaded, we can split the data into separate training sequences by splitting based on new lines. The snippet below will load the `republic_sequences.txt` data file from the current working directory.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# load
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
```

Listing 20.14: Load the clean sequences from file.

Next, we can encode the training data.

20.4.2 Encode Sequences

The word embedding layer expects input sequences to be comprised of integers. We can map each word in our vocabulary to a unique integer and encode our input sequences. Later, when we make predictions, we can convert the prediction to numbers and look up their associated words in the same mapping. To do this encoding, we will use the `Tokenizer` class in the Keras API.

First, the `Tokenizer` must be trained on the entire training dataset, which means it finds all of the unique words in the data and assigns each a unique integer. We can then use the fit `Tokenizer` to encode all of the training sequences, converting each sequence from a list of words to a list of integers.

```
# integer encode sequences of words
tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
sequences = tokenizer.texts_to_sequences(lines)
```

Listing 20.15: Train a tokenizer on the loaded sequences.

We can access the mapping of words to integers as a dictionary attribute called `word_index` on the `Tokenizer` object. We need to know the size of the vocabulary for defining the embedding layer later. We can determine the vocabulary by calculating the size of the mapping dictionary.

Words are assigned values from 1 to the total number of words (e.g. 7,409). The `Embedding` layer needs to allocate a vector representation for each word in this vocabulary from index 1 to the largest index and because indexing of arrays is zero-offset, the index of the word at the end of the vocabulary will be 7,409; that means the array must be 7,409 + 1 in length. Therefore, when specifying the vocabulary size to the `Embedding` layer, we specify it as 1 larger than the actual vocabulary.

```
# vocabulary size
vocab_size = len(tokenizer.word_index) + 1
```

Listing 20.16: Calculate the size of the vocabulary.

20.4.3 Sequence Inputs and Output

Now that we have encoded the input sequences, we need to separate them into input (`X`) and output (`y`) elements. We can do this with array slicing. After separating, we need to one hot encode the output word. This means converting it from an integer to a vector of 0 values, one for each word in the vocabulary, with a 1 to indicate the specific word at the index of the words integer value.

This is so that the model learns to predict the probability distribution for the next word and the ground truth from which to learn from is 0 for all words except the actual word that comes next. Keras provides the `to_categorical()` that can be used to one hot encode the output words for each input-output sequence pair.

Finally, we need to specify to the `Embedding` layer how long input sequences are. We know that there are 50 words because we designed the model, but a good generic way to specify that is to use the second dimension (number of columns) of the input data's shape. That way, if you change the length of sequences when preparing data, you do not need to change this data loading code; it is generic.

```
# separate into input and output
sequences = array(sequences)
X, y = sequences[:,:-1], sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
seq_length = X.shape[1]
```

Listing 20.17: Split text data into input and output sequences.

20.4.4 Fit Model

We can now define and fit our language model on the training data. The learned embedding needs to know the size of the vocabulary and the length of input sequences as previously discussed. It also has a parameter to specify how many dimensions will be used to represent each word. That is, the size of the embedding vector space.

Common values are 50, 100, and 300. We will use 50 here, but consider testing smaller or larger values. We will use a two LSTM hidden layers with 100 memory cells each. More memory cells and a deeper network may achieve better results.

A dense fully connected layer with 100 neurons connects to the LSTM hidden layers to interpret the features extracted from the sequence. The output layer predicts the next word as a single vector the size of the vocabulary with a probability for each word in the vocabulary. A softmax activation function is used to ensure the outputs have the characteristics of normalized probabilities.

```
# define the model
def define_model(vocab_size, seq_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 50, input_length=seq_length))
```

```
model.add(LSTM(100, return_sequences=True))
model.add(LSTM(100))
model.add(Dense(100, activation='relu'))
model.add(Dense(vocab_size, activation='softmax'))
# compile network
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# summarize defined model
model.summary()
plot_model(model, to_file='model.png', show_shapes=True)
return model
```

Listing 20.18: Define the language model.

A summary of the defined network is printed as a sanity check to ensure we have constructed what we intended.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 50, 50)            370500
_________________________________________________________________
lstm_1 (LSTM)                (None, 50, 100)           60400
_________________________________________________________________
lstm_2 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense_1 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_2 (Dense)              (None, 7410)              748410
=================================================================
Total params: 1,269,810
Trainable params: 1,269,810
Non-trainable params: 0
_________________________________________________________________
```

Listing 20.19: Example output from summarizing the defined model.

A plot the defined model is then saved to file with the name `model.png`.

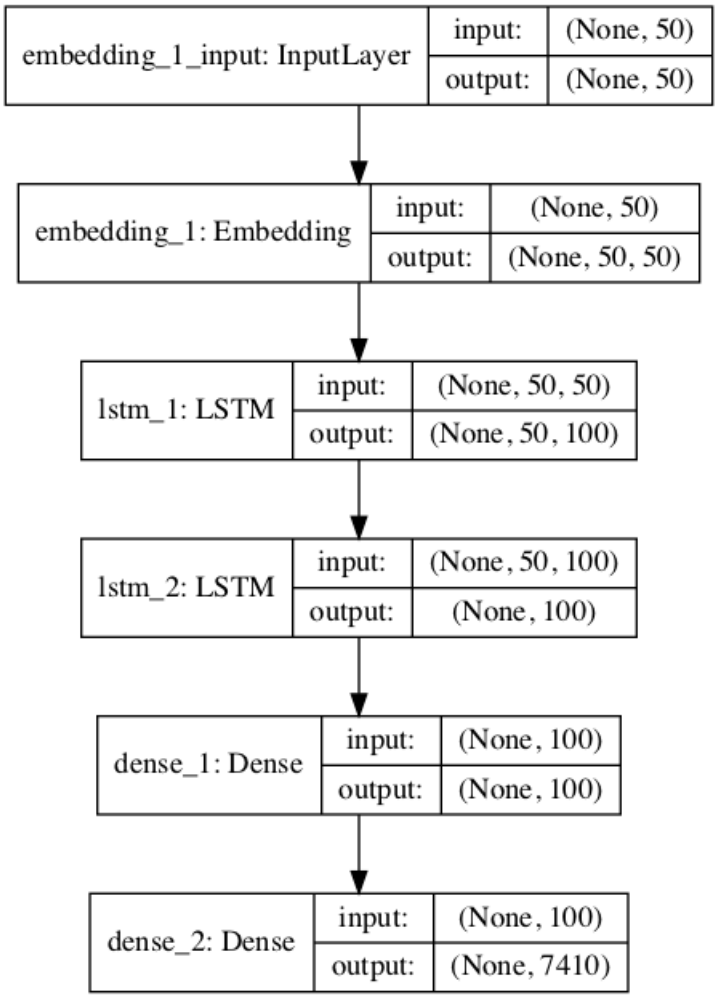

Figure 20.1: Plot of the defined word-based language model.

The model is compiled specifying the categorical cross entropy loss needed to fit the model. Technically, the model is learning a multiclass classification and this is the suitable loss function for this type of problem. The efficient Adam implementation to mini-batch gradient descent is used and accuracy is evaluated of the model. Finally, the model is fit on the data for 100 training epochs with a modest batch size of 128 to speed things up. Training may take a few hours on modern hardware without GPUs. You can speed it up with a larger batch size and/or fewer training epochs.

During training, you will see a summary of performance, including the loss and accuracy evaluated from the training data at the end of each batch update. You will get different results, but perhaps an accuracy of just over 50% of predicting the next word in the sequence, which is not bad. We are not aiming for 100% accuracy (e.g. a model that memorized the text), but rather a model that captures the essence of the text.

```
...
Epoch 96/100
118633/118633 [==============================] - 265s - loss: 2.0324 - acc: 0.5187
Epoch 97/100
118633/118633 [==============================] - 265s - loss: 2.0136 - acc: 0.5247
Epoch 98/100
```

```
118633/118633 [==============================] - 267s - loss: 1.9956 - acc: 0.5262
Epoch 99/100
118633/118633 [==============================] - 266s - loss: 1.9812 - acc: 0.5291
Epoch 100/100
118633/118633 [==============================] - 270s - loss: 1.9709 - acc: 0.5315
```

Listing 20.20: Example output from training the language model.

20.4.5 Save Model

At the end of the run, the trained model is saved to file. Here, we use the Keras model API to save the model to the file `model.h5` in the current working directory. Later, when we load the model to make predictions, we will also need the mapping of words to integers. This is in the `Tokenizer` object, and we can save that too using Pickle.

```
# save the model to file
model.save('model.h5')
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

Listing 20.21: Save the fit model and `Tokenizer` to file.

20.4.6 Complete Example

We can put all of this together; the complete example for fitting the language model is listed below.

```
from numpy import array
from pickle import dump
from keras.preprocessing.text import Tokenizer
from keras.utils.vis_utils import plot_model
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# define the model
def define_model(vocab_size, seq_length):
  model = Sequential()
  model.add(Embedding(vocab_size, 50, input_length=seq_length))
  model.add(LSTM(100, return_sequences=True))
  model.add(LSTM(100))
  model.add(Dense(100, activation='relu'))
```

```
  model.add(Dense(vocab_size, activation='softmax'))
  # compile network
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model

# load
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
# integer encode sequences of words
tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
sequences = tokenizer.texts_to_sequences(lines)
# vocabulary size
vocab_size = len(tokenizer.word_index) + 1
# separate into input and output
sequences = array(sequences)
X, y = sequences[:,:-1], sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
seq_length = X.shape[1]
# define model
model = define_model(vocab_size, seq_length)
# fit model
model.fit(X, y, batch_size=128, epochs=100)
# save the model to file
model.save('model.h5')
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

Listing 20.22: Complete example training the language model.

20.5 Use Language Model

Now that we have a trained language model, we can use it. In this case, we can use it to generate new sequences of text that have the same statistical properties as the source text. This is not practical, at least not for this example, but it gives a concrete example of what the language model has learned. We will start by loading the training sequences again.

20.5.1 Load Data

We can use the same code from the previous section to load the training data sequences of text. Specifically, the `load_doc()` function.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
```

```
  file.close()
  return text

# load cleaned text sequences
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
```

Listing 20.23: Load the clean sequences from file.

We need the text so that we can choose a source sequence as input to the model for generating a new sequence of text. The model will require 50 words as input. Later, we will need to specify the expected length of input. We can determine this from the input sequences by calculating the length of one line of the loaded data and subtracting 1 for the expected output word that is also on the same line.

```
seq_length = len(lines[0].split()) - 1
```

Listing 20.24: Calculate the expected input length.

20.5.2 Load Model

We can now load the model from file. Keras provides the `load_model()` function for loading the model, ready for use.

```
# load the model
model = load_model('model.h5')
```

Listing 20.25: Load the saved model from file.

We can also load the tokenizer from file using the Pickle API.

```
# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
```

Listing 20.26: Load the saved `Tokenizer` from file.

We are ready to use the loaded model.

20.5.3 Generate Text

The first step in generating text is preparing a seed input. We will select a random line of text from the input text for this purpose. Once selected, we will print it so that we have some idea of what was used.

```
# select a seed text
seed_text = lines[randint(0,len(lines))]
print(seed_text + '\n')
```

Listing 20.27: Select random examples as seed text.

Next, we can generate new words, one at a time. First, the seed text must be encoded to integers using the same tokenizer that we used when training the model.

```
encoded = tokenizer.texts_to_sequences([seed_text])[0]
```

Listing 20.28: Encode the selected seed text.

The model can predict the next word directly by calling `model.predict_classes()` that will return the index of the word with the highest probability.

```
# predict probabilities for each word
yhat = model.predict_classes(encoded, verbose=0)
```

Listing 20.29: Predict the next word in the sequence.

We can then look up the index in the `Tokenizer`'s mapping to get the associated word.

```
out_word = ''
for word, index in tokenizer.word_index.items():
  if index == yhat:
    out_word = word
    break
```

Listing 20.30: Map the predicted integer to a word in the known vocabulary.

We can then append this word to the seed text and repeat the process. Importantly, the input sequence is going to get too long. We can truncate it to the desired length after the input sequence has been encoded to integers. Keras provides the `pad_sequences()` function that we can use to perform this truncation.

```
encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
```

Listing 20.31: Pad the encoded sequence.

We can wrap all of this into a function called `generate_seq()` that takes as input the model, the tokenizer, input sequence length, the seed text, and the number of words to generate. It then returns a sequence of words generated by the model.

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
  result = list()
  in_text = seed_text
  # generate a fixed number of words
  for _ in range(n_words):
    # encode the text as integer
    encoded = tokenizer.texts_to_sequences([in_text])[0]
    # truncate sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
    # predict probabilities for each word
    yhat = model.predict_classes(encoded, verbose=0)
    # map predicted word index to word
    out_word = ''
    for word, index in tokenizer.word_index.items():
      if index == yhat:
        out_word = word
        break
    # append to input
    in_text += ' ' + out_word
    result.append(out_word)
  return ' '.join(result)
```

Listing 20.32: Function to generate a sequence of words given the model and seed text.

We are now ready to generate a sequence of new words given some seed text.

```
# generate new text
generated = generate_seq(model, tokenizer, seq_length, seed_text, 50)
print(generated)
```

Listing 20.33: Example of generating a sequence of text.

Putting this all together, the complete code listing for generating text from the learned-language model is listed below.

```
from random import randint
from pickle import load
from keras.models import load_model
from keras.preprocessing.sequence import pad_sequences

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# generate a sequence from a language model
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
  result = list()
  in_text = seed_text
  # generate a fixed number of words
  for _ in range(n_words):
    # encode the text as integer
    encoded = tokenizer.texts_to_sequences([in_text])[0]
    # truncate sequences to a fixed length
    encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
    # predict probabilities for each word
    yhat = model.predict_classes(encoded, verbose=0)
    # map predicted word index to word
    out_word = ''
    for word, index in tokenizer.word_index.items():
      if index == yhat:
        out_word = word
        break
    # append to input
    in_text += ' ' + out_word
    result.append(out_word)
  return ' '.join(result)

# load cleaned text sequences
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
seq_length = len(lines[0].split()) - 1
# load the model
model = load_model('model.h5')
# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
```

```
# select a seed text
seed_text = lines[randint(0,len(lines))]
print(seed_text + '\n')
# generate new text
generated = generate_seq(model, tokenizer, seq_length, seed_text, 50)
print(generated)
```

Listing 20.34: Complete example of generating sequences of text.

Running the example first prints the seed text.

```
when he said that a man when he grows old may learn many things for he can no more learn
    much than he can run much youth is the time for any extraordinary toil of course and
    therefore calculation and geometry and all the other elements of instruction which are a
```

Listing 20.35: Example output from selecting seed text.

Then 50 words of generated text are printed.

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

```
preparation for dialectic should be presented to the name of idle spendthrifts of whom the
    other is the manifold and the unjust and is the best and the other which delighted to
    be the opening of the soul of the soul and the embroiderer will have to be said at
```

Listing 20.36: Example output of generated text.

You can see that the text seems reasonable. In fact, the addition of concatenation would help in interpreting the seed and the generated text. Nevertheless, the generated text gets the right kind of words in the right kind of order. Try running the example a few times to see other examples of generated text.

20.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Contrived Seed Text**. Hand craft or select seed text and evaluate how the seed text impacts the generated text, specifically the initial words or sentences generated.
- **Simplify Vocabulary**. Explore a simpler vocabulary, perhaps with stemmed words or stop words removed.
- **Data Cleaning**. Consider using more or less cleaning of the text, perhaps leave in some punctuation or perhaps replacing all fancy names with one or a handful. Evaluate how these changes to the size of the vocabulary impact the generated text.
- **Tune Model**. Tune the model, such as the size of the embedding or number of memory cells in the hidden layer, to see if you can develop a better model.
- **Deeper Model**. Extend the model to have multiple LSTM hidden layers, perhaps with dropout to see if you can develop a better model.

- **Develop Pre-Trained Embedding**. Extend the model to use pre-trained Word2Vec vectors to see if it results in a better model.
- **Use GloVe Embedding**. Use the GloVe word embedding vectors with and without fine tuning by the network and evaluate how it impacts training and the generated words.
- **Sequence Length**. Explore training the model with different length input sequences, both shorter and longer, and evaluate how it impacts the quality of the generated text.
- **Reduce Scope**. Consider training the model on one book (chapter) or a subset of the original text and evaluate the impact on training, training speed and the resulting generated text.
- **Sentence-Wise Model**. Split the raw data based on sentences and pad each sentence to a fixed length (e.g. the longest sentence length).

If you explore any of these extensions, I'd love to know.

20.7 Further Reading

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

- Project Gutenberg.
  https://www.gutenberg.org/
- The Republic by Plato on Project Gutenberg.
  https://www.gutenberg.org/ebooks/1497
- Republic (Plato) on Wikipedia.
  https://en.wikipedia.org/wiki/Republic_(Plato)
- Language model on Wikipedia.
  https://en.wikipedia.org/wiki/Language_model

20.8 Summary

In this tutorial, you discovered how to develop a word-based language model using a word embedding and a recurrent neural network. Specifically, you learned:

- How to prepare text for developing a word-based language model.
- How to design and fit a neural language model with a learned embedding and an LSTM hidden layer.
- How to use the learned language model to generate new text with similar statistical properties as the source text.

20.8.1 Next

This is the final chapter in the language modeling part. In the next part you will discover how to develop automatic caption generation for photographs.