ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

LECTURE NOTES

COURSE: Deep Learning

COURSE CODE: BCS714A

SEMESTER: VII

COURSE COORDINATOR: Mrs. Harshitha M K

(ACADEMIC YEAR 2026-27)

MODULE-5

Interactive Applications of Deep Learning: Natural Language Processing: Preprocessing Natural Language Data: Tokenization, Converting All Characters to Lowercase, Removing Stop Words and Punctuation, Stemming, Handling *n*-grams, Preprocessing the Full Corpus, Creating Word Embeddings with word2vec: The Essential Theory Behind word2vec, Evaluating Word Vectors, Running word2vec, Plotting Word Vectors, The Area under the ROC Curve: The Confusion Matrix, Calculating the ROC AUC Metric, Natural Language Classification with Familiar Networks: Loading the IMDb Film Reviews, Examining the IMDb Data, Standardizing the Length of the Reviews, Dense Network, Convolutional Networks, Networks Designed for Sequential Data: Recurrent Neural Networks, Long Short-Term Memory Units, Bidirectional LSTMs, Stacked Recurrent Models, Seq2seq and Attention, Transfer Learning in NLP.

Preprocessing Natural Language Data:

- Natural Language Processing (NLP) involves enabling machines to understand and work with human language.
- Before building models, raw text must be transformed into a clean and structured format. This process is called **Natural Language Preprocessing**.

Well-preprocessed text improves:

- Model accuracy
- Training speed
- Generalization
- Vocabulary efficiency

There are steps you can take to preprocess natural language data such that the modelling you carry out downstream may be more accurate. Common natural language preprocessing options include:

■ *Tokenization*: This is the splitting of a document (e.g., a book) into a list of discrete elements of language (e.g., words), which we call *tokens*.

Example:

Text:
"Natural Language Processing is exciting!"

Word tokens:
["natural", "language", "processing", "is", "exciting"]

Sentence tokens:
["Natural Language Processing is exciting!"]

■ Converting all characters *to lowercase*: A capitalized word at the beginning of a sentence (e.g., *She*) has the same meaning as when it's used later in a sentence (*she*). By converting all characters in a corpus to lowercase, we disregard any use of capitalization.

Example:

"She LOVES Dogs" → "she loves dogs"

■ Removing *stop words*: These are frequently occurring words that tend to contain relatively little distinctive meaning, such as *the*, *at*, *which*, and *of*. There is no universal consensus on the precise list of stop words, but depending on your application it may be sensible to ensure that certain words are (or aren't!) considered to be stop words. For

example, in this chapter, we'll build a model to classify movie reviews as positive or negative. Some lists of stop words include negations like *didn't*, *isn't*, and wouldn't that might be critical for our model to identify the sentiment of a movie review, so these words probably shouldn't be removed.

■ Removing punctuation: Punctuation marks generally don't add much value to a natural language model and so are often removed.

■ Stemming:1 Stemming is the truncation of words down to their stem. For example, the words house and housing both have the stem house. With smaller datasets in particular, stemming can be productive because it pools words with similar meanings into a single token. There will be more examples of this stemmed token's context, enabling techniques like word2vec or GloVe to more accurately identify an appropriate location for the token in word-vector

■ Handling n-grams: Some words commonly co-occur in such a way that the combination of words is better suited to being considered a single concept than several separate concepts. As examples, New York is a bigram (an n-gram of length two), and New York City is a trigram (an n-gram of length three). When chained together, the words new, york, and city have a specific meaning that might be better captured by a single token (and therefore a single location in word-vector space) than three separate ones.

- Depending on the particular task that we've designed our model for, as well as the dataset that we're feeding into it, we may use all, some, or none of these data preprocessing steps. As you consider applying any preprocessing step to your particular problem, you can use your intuition to weigh whether it might ultimately be valuable to your downstream task. We've already mentioned some examples of this:

  ■ Stemming may be helpful for a small corpus but unhelpful for a large one.

  ■ Likewise, converting all characters to lowercase is likely to be helpful when you're working with a small corpus, but, in a larger corpus that has many more examples of individual uses of words, the distinction of, say, *general* (an adjective meaning "widespread") versus *General* (a noun meaning the commander of an army) may be valuable.

  ■ Removing punctuation would not be an advantage in all cases. Consider, for example, if you were building a question-answering algorithm, which could use question marks to help it identify questions.

  ■ Negations may be helpful as stop words for some classifiers but probably not for a sentiment classifier, for example. Which words you include in your list of stop

words could be crucial to your particular application, so be careful with this one. In many instances, it will be best to remove only a limited number of stop words.

- In deep learning, the utility of preprocessing steps can be empirically tested by incorporating them and observing their impact on model accuracy. Generally, as the corpus size grows, the benefits of extensive preprocessing reduce. For smaller corpora, pooling rare or out-of-vocabulary words into common tokens can help the model learn better, while for very large corpora, preserving individual rare words is better to capture subtle nuances and unique meanings effectively.
- Key points about preprocessing include:
  - For small datasets, grouping rare words into a shared token helps overcome scarcity and improves training efficiency by reducing vocabulary complexity.
  - For large datasets, such grouping is less helpful because the frequency of rare words is sufficient to model their specific meanings accurately.
  - Text preprocessing often involves token normalization, stopword removal, padding or truncation of sequences, and subword tokenization methods like Byte Pair Encoding (BPE) to reduce vocabulary size without losing meaningful distinctions.
  - Empirical evaluation of preprocessing steps by monitoring their impact on downstream model accuracy and resource usage is recommended, as preprocessing is not one-size-fits-all and depends on corpus size and domain.
  - Larger corpora benefit from fewer aggressive preprocessing steps, preserving more unique tokens to maintain semantic subtleties.
  - Thus, the decision to apply preprocessing steps such as rare word pooling or token normalization should be guided by the dataset size and verified experimentally by observing model accuracy changes after adding those steps

To provide practical examples of these preprocessing steps in action, we invite you to check out our Natural Language Preprocessing Jupyter notebook. It begins by loading a number of dependencies:

```
import nltk
from nltk import word_tokenize, sent_tokenize
from nltk.corpus import stopwords
from nltk.stem.porter import *
nltk.download('gutenberg')
nltk.download('punkt')
nltk.download('stopwords')

import string

import gensim
from gensim.models.phrases import Phraser, Phrases
from gensim.models.word2vec import Word2Vec

from sklearn.manifold import TSNE

import pandas as pd
from bokeh.io import output_notebook, output_file
from bokeh.plotting import show, figure
%matplotlib inline
```

Tokenization

The small corpus from nltk's Gutenberg dataset, including texts like Jane Austen's Emma and Shakespeare's plays, with around 2.6 million words, can be processed effectively on a laptop. To tokenize the corpus into sentences, nltk's sent_tokenize() method on the raw text form (e.g., gutenberg.raw()) is a common starting step.

Further common preprocessing steps for this Gutenberg corpus in nltk include:

- Tokenization into words using word_tokenize() for analyses that require word-level processing.
- Converting tokens to lowercase to normalize.
- Removing punctuation and stopwords to reduce noise.
- Optionally applying stemming or lemmatization for word-root normalization.
- Constructing a vocabulary from tokens for model input.
- Calculating token frequency distribution to manage rare and common words.

These NLP preprocessing steps prepare the text data to be structured and meaningful for downstream modeling tasks. They are straightforward to apply through nltk's functions an d fundamental for smaller corpora like this Gutenberg set to improve model training efficiency and accuracy.

Example code snippet to load and tokenize sentences:

python

```
from nltk.corpus import gutenberg
from nltk.tokenize import sent_tokenize
# Load raw text from the Gutenberg corpus
raw_text = gutenberg.raw()
```

```
# Tokenize into sentences
gberg_sent_tokens = sent_tokenize(raw_text)
print(f"Number of sentences: {len(gberg_sent_tokens)}")
```

You can further extend preprocessing based on your modeling needs with word tokenization, normalization, and filtering steps.

The tokenization process you described for the Project Gutenberg corpus using nltk works as follows:

- The corpus raw text can be split into sentences using sent_tokenize(). For example, gberg_sent_tokens = sent_tokenize(gutenberg.raw()). The first element will contain the title page and chapter markers blended with the first sentence, while subsequent elements typically contain standalone sentences.
- You can access the first sentence with gberg_sent_tokens[0] and the second with gberg_sent_tokens[1].
- To tokenize a sentence into words, use word_tokenize() from nltk, e.g. word_tokenize(gberg_sent_tokens[1]). This splits the sentence into a list of words and punctuation, stripping whitespace and newlines.
- You can directly access words from this list by index, for example, word_tokenize(gberg_sent_tokens[1])[14] gives the 15th word of the second sentence.
- Instead of doing sentence tokenization followed by word tokenization manually, nltk provides a convenient gutenberg.sents() method, which returns a list of lists where each inner list is a tokenized sentence (a list of word tokens).
- Thus, using gutenberg.sents() is often simpler and more convenient, as it combines sentence splitting and word tokenizing in a single step, producing a tokenized corpus ready for analysis or modeling.

Here's an example code snippet illustrating this process:

python

```
from nltk.corpus import gutenberg
from nltk.tokenize import sent_tokenize, word_tokenize
# Sentence tokenization of raw text
gberg_sent_tokens = sent_tokenize(gutenberg.raw())
print(gberg_sent_tokens[0])  # First element with title and first sentence
```

```
print(gberg_sent_tokens[1])  # Second element, first standalone sentence
# Word tokenization of second sentence
words = word_tokenize(gberg_sent_tokens[1])
print(words)
print(words[14])  # 15th word in the sentence
# Using built-in sents() method for tokenized sentences
gberg_sents = gutenberg.sents()
print(gberg_sents[0])  # First sentence as list of words and tokens
```

```
'[Emma by Jane Austen 1816]\n\nVOLUME I\n\nCHAPTER I\n\n\nEmma Wood-
house, handsome, clever, and rich, with a comfortable home\nand happy
disposition, seemed to unite some of the best blessings\nof existence;
and had lived nearly twenty-one years in the world\nwith very little to
distress or vex her.'
```

A stand-alone sentence is found in the second element, which you can view by executing `gberg_sent_tokens[1]`:

```
"She was the youngest of the two daughters of a most affectionate,
\nindulgent father; and had, in consequence of her sister's mar-
riage,\nbeen mistress of his house from a very early period."
```

```
['She',
 'was',
 'the',
 'youngest',
 'of',
 'the',
 'two',
 'daughters',
 'of',
 'a',
 'most',
 'affectionate',
 ',',
 'indulgent',
 'father',
 ';',
 'and',
 'had',
 ',',
 'in',
 'consequence',
 'of',
 'her',
 'sister',
 '''',
 's',
 'marriage',
 ',',
 'been',
 'mistress',
 'of',
 'his',
 'house',
 'from',
 'a',
 'very',
 'early',
 'period',
 '.']
```

Figure 11.1 The second sentence of Jane Austen's classic *Emma* tokenized to the word level

- The gutenberg.sents() method in nltk returns the corpus as a list of sentences, where each sentence is itself a list of word tokens. Importantly, this method separates the title page and chapter markers into separate sentence elements, rather than blending them with the text.

  For example, calling gberg_sents[0:3] on the Emma text yields:

  text

  [['[', 'Emma', 'by', 'Jane', 'Austen', '1816', ']'],
  ['VOLUME', 'I'],
  ['CHAPTER', 'I']]

  This means that the title and chapter headings are individual elements in the list.

- Because of this, the first actual sentence of the book Emma appears as the fourth sentence element (index 3), and the second actual sentence (the one containing the word "father")

appears as the fifth sentence element (index 4). Hence, to access the 15th word ("father") in the second actual sentence, you would use gberg_sents[4][14].

- This approach allows for fine-grained access to text components while preserving the natural sentence structure and tokenization in the corpus, making it convenient for analysis or modeling tasks requiring precise sentence and token positions.

Converting All Characters to Lowercase:

The example you provided shows how to convert the words in a tokenized sentence to lowercase using a list comprehension:

python

```
[w.lower() for w in gberg_sents[4]]
```

- This converts each word token in the fifth sentence of the corpus to its lowercase version. Lowercasing is a common preprocessing step in NLP as it normalizes text, preventing the model from treating the same word with different capitalization (e.g., "She" vs. "she") as distinct tokens. This improves consistency and reduces the vocabulary size.
- To handle stop words and punctuation, you can combine nltk's English stopword list with Python's string.punctuation like this:

python

```
import string
from nltk.corpus import stopwords
stpwrds = stopwords.words('english') + list(string.punctuation)
```

Then, to remove all these stopwords and punctuation from a sentence, you can use a list comprehension incorporating the lowercase step:

python

```
[w.lower() for w in gberg_sents[4] if w.lower() not in stpwrds]
```

This results in a list of lowercase tokens with common stopwords and punctuation removed, which is helpful in many NLP applications where such words add little semantic meaning. However, be cautious that some stopwords like negations ("not") might be important in tasks like sentiment analysis.

- Thus, combining lowercase conversion with stopword and punctuation removal using list comprehensions is an effective and concise way to clean text data for downstream deep learning or NLP tasks

- The code example provided effectively removes stop words and punctuation from a tokenized sentence using a list comprehension:

python

```
[w.lower() for w in gberg_sents[4] if w.lower() not in stpwrds]
```

Here is how it works:

- Each word token in the sentence gberg_sents[4] is converted to lowercase.
- It then checks if the lowercase word is NOT in the combined stopword and punctuation list stpwrds.
- Only the words that are not stop words or punctuation are included in the resulting list.
- The result is a much shorter list of tokens containing only meaningful words that typically contribute significant semantic content, such as:
- ['youngest', 'two', 'daughters', 'affectionate', 'indulgent', 'father', 'consequence', 'sister', 'marriage', 'mistress', 'house', 'early', 'period']
- This approach is very common in NLP preprocessing to filter out common filler words, conjunctions, articles, and punctuation that often do not add useful information for many downstream NLP or deep learning tasks. It reduces noise and vocabulary size, helping models focus on the core semantic content.

Stemming

Stemming with NLTK's PorterStemmer is a popular way to reduce words to their root forms by chopping off suffixes. In your Example 11.3, the list comprehension adds stemming to the previous stop-word removal and lowercasing step:

python

```
[stemmer.stem(w.lower()) for w in gberg_sents[4] if w.lower() not in stpwrds]
```

This outputs stemmed words such as:

['youngest', 'two', 'daughter', 'affection', 'indulg', 'father', 'consequ', 'sister', 'marriag', 'mistress', 'hous', 'earli', 'period']

Key effects include:

- "daughters" reduced to "daughter" (singularized form)

- "house" reduced to "hous" (roots shared with related words like "housing")
- "early" reduced to "earli" (captures different tenses)

- Stemming pools related words to the same root so that the model can learn a more robust representation with aggregated examples, which is helpful in smaller corpora with fewer total word occurrences.
- However, with large corpora, stemming might lose important nuances by conflating distinct but related words, while larger data allows modeling these subtle distinctions separately, preserving richer meanings.

PorterStemmer usage example:

python

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
words = ['daughters', 'house', 'early']
stemmed = [stemmer.stem(word) for word in words]
print(stemmed)  # ['daughter', 'hous', 'earli']
```

This outputs the following:

```
['youngest',
 'two',
 'daughter',
 'affection',
 'indulg',
 'father',
 'consequ',
 'sister',
 'marriag',
 'mistress',
 'hous',
 'earli',
 'period']
```

Stemming is thus a valuable text preprocessing step especially for small datasets, but its use should be decided based on corpus size and task needs.

Handling n-grams

- To treat a bigram like New York as a single token instead of two, we can use the Phrases() and Phraser() methods from the gensim library. As demonstrated in Example 11.4, we use them in this way:

1. Phrases() to train a "detector" to identify how often any given pair of words occurs together in our corpus (the technical term for this is bigram collocation) relative to how often each word in the pair occurs by itself
2. Phraser() to take the bigram collocations detected by the Phrases() object and then use this information to create an object that can efficiently be passed over ourcorpus, converting all bigram collocations from two consecutive tokens into a single token

Example 11.4 Detecting collocated bigrams

```
phrases = Phrases(gberg_sents)
bigram = Phraser(phrases)
```

- By running bigram.phrasegrams, we output a dictionary of the count and score of each bigram. The topmost lines of this dictionary are provided in Figure 11.2.
- Each bigram in Figure 11.2 has a count and a score associated with it. The bigram two daughters, for example, occurs a mere 19 times across our Gutenberg corpus. This bigram has a fairly low score (12.0), meaning the terms two and daughters do not occur together very frequently relative to how often they occur apart. In contrast, the bigram Miss Ta ylor occurs more often (48 times), and the terms Miss and Taylor occur much more frequently together relative to how often they occur on their own (score of 453.8).

```
{(b'two', b'daughters'): (19, 11.966813731181546),
 (b'her', b'sister'): (195, 17.7960829227865),
 (b"'", b's'): (9781, 31.066242737744524),
 (b'very', b'early'): (24, 11.01214147275924),
 (b'Her', b'mother'): (14, 13.529425062715127),
 (b'long', b'ago'): (38, 63.22343628984788),
 (b'more', b'than'): (541, 29.023584433996874),
 (b'had', b'been'): (1256, 22.306024648925288),
 (b'an', b'excellent'): (54, 39.063874851750626),
 (b'Miss', b'Taylor'): (48, 453.75918026073305),
 (b'very', b'fond'): (28, 24.134280468850747),
 (b'passed', b'away'): (25, 12.35053642325912),
 (b'too', b'much'): (173, 31.376002029426687),
 (b'did', b'not'): (935, 11.728416217142811),
 (b'any', b'means'): (27, 14.096964108090186),
 (b'wedding', b'-'): (15, 17.4695197740113),
 (b'Her', b'father'): (18, 13.129571562488772),
 (b'after', b'dinner'): (21, 21.5285481168817),
```

Figure 11.2 A dictionary of bigrams detected within our corpus

- Scanning over the bigrams in Figure 11.2, notice that they are marred by capitalized words and punctuation marks. We'll resolve those issues in the next section, but in the meantime let's explore how the bigram object we've created can be used to convert

bigrams from two consecutive tokens into one. Let's tokenize a short sentence by using the split() method on a string of characters wherever there's a space, as follows:

tokenized_sentence = "Jon lives in New York City".split()

- If we print tokenized_sentence, we output a list of unigrams only: ['Jon', 'lives', 'in', 'New', 'York', 'City']. If, however, we pass the list through our genism bigram object by using bigram[tokenized_sentence], the list then contains the bigram

***New York*: ['Jon', 'lives', 'in', 'New_York', 'City'].**

Preprocessing the Full Corpus:

- To preprocess the entire Project Gutenberg corpus including steps like lowercasing, removing punctuation and stopwords, stemming, and collocating bigrams, you can compose a pipeline as follows:

1. Load the tokenized sentences from the corpus.
2. Lowercase all tokens and remove stopwords and punctuation.
3. Apply stemming to normalize word forms.

Use gensim's Phrases and Phraser to detect and transform bigram collocations on the cleaned tokens.

- Andrew Maas's Stanford Large Movie Review Dataset (50K reviews for binary sentiment classification) skips aggressive preprocessing like stopword removal and stemming. Stopwords carry sentiment signals (e.g., "not good"), while the large corpus size enables word-vector models to naturally cluster similar stems (e.g., "love"/"loved") in embedding space without explicit normalization

Example 11.5 **Removing capitalization and punctuation from Project Gutenberg corpus**

```
lower_sents = []
for s in gberg_sents:
    lower_sents.append([w.lower() for w in s if w.lower()
                        not in list(string.punctuation)])
```

- In this example, we begin with an empty list we call lower_sents, and then we append preprocessed sentences to it using a for loop.8 For preprocessing each sentence within the loop, we used a variation on the list comprehension from Example 11.2, in this case removing only punctuation marks while converting all characters to lowercase.

With punctuation and capitals removed, we can set about detecting collocated bigrams across the corpus afresh:

```
lower_bigram = Phraser(Phrases(lower_sents))
```

- Relative to Example 11.4, this time we created our gensim lower_bigram object in a single line by chaining the Phrases() and Phraser() methods together.
- The top of the output of a call to lower_bigram.phrasegrams is provided in Figure 11.3: Comparing these bigrams with those from Figure 11.2, we do indeed observe that they are all in lowercase (e.g., miss taylor) and bigrams that included punctuation marks are nowhere to be seen.

```
{(b'two', b'daughters'): (19, 11.080802900992637),
 (b'her', b'sister'): (201, 16.93971298099339),
 (b'very', b'early'): (25, 10.516998773665177),
 (b'her', b'mother'): (253, 10.70812618607742),
 (b'long', b'ago'): (38, 59.226442015336005),
 (b'more', b'than'): (562, 28.529926612065935),
 (b'had', b'been'): (1260, 21.583193129694834),
 (b'an', b'excellent'): (58, 37.41859680854167),
 (b'sixteen', b'years'): (15, 131.42913000977515),
 (b'miss', b'taylor'): (48, 420.4340982546865),
 (b'mr', b'woodhouse'): (132, 104.19907841850323),
 (b'very', b'fond'): (30, 24.185726346489627),
 (b'passed', b'away'): (25, 11.751473221742694),
 (b'too', b'much'): (177, 30.36309017383541),
 (b'did', b'not'): (977, 10.846196223896685),
 (b'any', b'means'): (28, 14.294148100212627),
 (b'after', b'dinner'): (22, 18.60737125272944),
 (b'mr', b'weston'): (162, 91.63290824201266),
```

Figure 11.3 Sample of a dictionary of bigrams detected within our lowercased and punctuation-free corpus

- Examining the results in Figure 11.3 further, however, it appears that the default minimum thresholds for both count and score are far too liberal. That is, word pairs like two daughters and her sister should not be considered bigrams.
- To attain bigrams that we thought were more sensible, we experimented with more conservative count and score thresholds by increasing them by powers of 2.
- Following this approach, we were generally satisfied by setting the optional Phrases() arguments to a min(imum) count of 32 and to a score threshold of 64, as shown in Example 11.6.

Example 11.6 Detecting collocated bigrams with more conservative thresholds

```
lower_bigram = Phraser(Phrases(lower_sents,
                               min_count=32, threshold=64))
```

- Although it's not perfect, because there are still a few questionable bigrams like great deal and few minutes, the output from a call to lower_bigram.phrasegrams is now largely

defensible, as shown in Figure 11.4.

- Armed with our well-appointed lower_bigram object from Example 11.6, we can at last use a for loop to iteratively append for ourselves a corpus of cleaned-up sentences, as in Example 11.7

Example 11.7 Creating a "clean" corpus that includes bigrams

```
clean_sents = []
for s in lower_sents:
    clean_sents.append(lower_bigram[s])
```

```
{(b'miss', b'taylor'): (48, 156.44059469941823),
 (b'mr', b'woodhouse'): (132, 82.04651843976633),
 (b'mr', b'weston'): (162, 75.87438262077481),
 (b'mrs', b'weston'): (249, 160.68485093258923),
 (b'great', b'deal'): (182, 93.36368125424357),
 (b'mr', b'knightley'): (277, 161.74131790625913),
 (b'miss', b'woodhouse'): (173, 229.03802722366902),
 (b'years', b'ago'): (56, 74.31594785893046),
 (b'mr', b'elton'): (214, 121.3990121932397),
 (b'dare', b'say'): (115, 89.94000515807346),
 (b'frank', b'churchill'): (151, 1316.4456593286038),
 (b'miss', b'bates'): (113, 276.39588291692513),
 (b'drawing', b'room'): (49, 84.91494947493561),
 (b'mrs', b'goddard'): (58, 143.57843432545658),
 (b'miss', b'smith'): (58, 73.03442128232508),
 (b'few', b'minutes'): (86, 204.16834974753786),
 (b'john', b'knightley'): (58, 83.03755747111268),
 (b'don', b't'): (830, 250.30957446808512),
```

Figure 11.4 Sample of a more conservatively thresholded dictionary of bigrams

```
['sixteen',
 'years',
 'had',
 'miss_taylor',
 'been',
 'in',
 'mr_woodhouse',
 's',
 'family',
 'less',
 'as',
 'a',
 'governess',
 'than',
 'a',
 'friend',
 'very',
 'fond',
 'of',
 'both',
 'daughters',
 'but',
 'particularly',
 'of',
 'emma']
```

Figure 11.5 Clean, preprocessed sentence from the Project Gutenberg corpus

- As an example, Figure 11.5 shows the seventh element of our clean corpus (clean_sents[6]), a sentence that includes the bigrams miss taylor and mr woodhouse.

Creating Word Embeddings with word2vec:

Word embeddings are numerical vector representations of words in a continuous, lower - dimensional space that capture semantic meanings, relationships, and contextual similarity among words. These embeddings enable machines to process and understand natural language more effectively compared to traditional methods.

Key points about word embeddings:

- Represent words as dense vectors where semantically similar words are geometrically closer.
- Capture syntactic and semantic information by exploiting the distributional hypothesis: words appearing in similar contexts tend to have similar meanings.
- Can be trained on large corpora by neural network-based architectures such as Word2Vec's Continuous Bag of Words (CBOW) or Skip-gram models.
- Allow for improved performance in various NLP tasks including sentiment analysis, text classification, and language modeling.
- Provide a more compact and meaningful representation than sparse methods like bag-of-words or TF-IDF.
- While a single line of code can generate embeddings (e.g., gensim's Word2Vec), it is important to understand the underlying theory and tune parameters to suit the characteristics of the corpus and the downstream task for better results.

The Essential Theory Behind word2vec

- The essential theory behind word2vec involves learning vector representations of words based on their surrounding context in a corpus, following the distributional hypothesis: "a word is known by the company it keeps." It is an unsupervised learning technique applied without labeled data.

  Word2vec has two main architectures:

  - Skip-gram (SG): Given a target word, it predicts the surrounding context words. It focuses on learning how a single word relates to its neighbors.
  - Continuous Bag of Words (CBOW): It predicts the target word from the average of all its surrounding context words within a window. The "bag of words" concept means that the order or position of context words is ignored.

- For example, in the phrase "**you shall know a word by the company it keeps,"** with a window size of three, CBOW takes all words surrounding "word" collectively (left and right) to predict "word." It treats these context words as a set without sequence, computing their average vector.
- Both methods learn embeddings representing word meanings by maximizing the probability of context-target word pairs, with CBOW generally being faster but SG sometimes capturing rare words better. The resulting word vectors enable models to infer semantic relationships and similarities effectively.
- Having considered the intuitiveness of the "BOW" component of the CBOW moniker, let's also consider the "continuous" part of it: The target word and context

Table 11.1 **Comparison of word2vec architectures**

Architecture	Predicts	Relative Strengths
Skip-gram (SG)	Context words given target word	Better for a smaller corpus; represents rare words well
CBOW	Target word given context words	Multiple times faster; represents frequent words slightly better

- word windows slide continuously one word at a time from the first word of the corpus all the way through to the final word. At each position along the way, the target word is estimated given the context words. Via stochastic gradient descent, the location of words within vector space can be shifted, and thereby these target-word estimates can gradually be improved.
- In practice, and as summarized in Table 11.1, the SG architecture is a better choice when you're working with a small corpus. It represents rare words in word-vector space well. In contrast, CBOW is much more computationally efficient, so it is the better option when you're working with a very large corpus. Relative to SG, CBOW also represents frequently occurring words slightly better.

Word Vector Evaluation Methods

Word vectors can be evaluated using two main approaches: intrinsic and extrinsic evaluations.

Extrinsic Evaluation

- Measures word vector quality through performance in downstream NLP tasks (e.g., sentiment analysis, named entity recognition).

- Requires full pipeline execution, including model training, making it computationally expensive.
- Provides direct evidence of utility since improvements correlate with task accuracy.

Intrinsic Evaluation

- Tests word vectors on standalone linguistic subtasks, such as arithmetic analogies (e.g., king - man + woman ≈ queen).
- Quick to run and helps debug intermediate representations.
- Limitation: May not guarantee downstream task improvements without established correlation.

Trade-offs: Extrinsic offers definitive validation but is resource-intensive; intrinsic provides fast feedback but indirect evidence of quality. Choose based on development stage—intrinsic for rapid iteration, extrinsic for final validation.

Running word2vec

As mentioned earlier, and as shown in Example 11.8, word2vec can be run in a single line of code—albeit with quite a few arguments.

Example 11.8 **Running word2vec**

```
model = Word2Vec(sentences=clean_sents, size=64,
                 sg=1, window=10, iter=5,
                 min_count=10, workers=4)
```

Here's a breakdown of each of the arguments we passed into the Word2Vec() method from the gensim library:

■ **sentences:** Pass in a list of lists like clean_sents as a corpus. Elements in the higher-level list are sentences, whereas elements in the lower-level list can be wordlevel tokens.

■ **size:**

- The number of dimensions in the word-vector space that will result from running word2vec. This is a hyperparameter that can be varied and evaluated extrinsically or intrinsically. Like other hyperparameters in this book, there is a Goldilocks sweet spot.
- You can home in on an optimal value by specifying, say, 32 dimensions and varying this value by powers of 2. Doubling the number of dimensions will double the computational complexity of your downstream deep learning model, but if doing this results in mar kedly higher model accuracy then this extrinsic evaluation suggests that the extra complexity could be worthwhile.

- On the other hand, halving the number of dimensions halves computational complexity downstream: If this can be done without appreciably decreasing your NLP model's accuracy, then it should be.
- By performing a handful of intrinsic inspections (which we'll go over shortly), we found 64 dimensions to provide more sensible word vectors than 32 dimensions for this particular case.

  Doubling this figure to 128, however, provided no noticeable improvement.

■ **sg:** Set to 1 to choose the skip-gram architecture, or leave at the 0 default to choose CBOW. As summarized in Table 11.1, SG is generally better suited to small datasets like our Gutenberg corpus.

■ **window:** For SG, a window size of 10 (for a total of 20 context words) is a good bet, so we set this hyperparameter to 10. If we were using CBOW, then a window size of 5 (for a total of 10 context words) could be near the optimal value. In either case, this hyperparameter can be experimented with and evaluated extrinsically or intrinsically. Small adjustments to this hyperparameter may not be perceptibly impactful, however.

■ **iter:** By default, the gensim Word2Vec() method iterates over the corpus fed into it (i.e., slides over all of the words) five times. Multiple iterations of word2vec is analogous to multiple epochs of training a deep learning model. With a small corpus like ours, the word vectors improve over several iterations. With a very large corpus, on the other hand, it might be cripplingly computationally expensive to run even two iterations—and, because there are so many examples of words in a very large corpus anyway, the word vectors might not be any better.

■ **min_count:** This is the minimum number of times a word must occur across the corpus in order to fit it into word-vector space. If a given target word occurs only once or a few times, there are a limited number of examples of its contextual words this, a minimum count of about 10 is often reasonable. The higher the count, the smaller the vocabulary of words that will be available to your downstream NLP task. This is yet another hyperparameter that can be tuned, with extrinsic evaluations likely being more illuminating than intrinsic ones because the size of the vocabulary you have to work with could make a considerable impact on your downstream NLP application.

■ **workers:** This is the number of processing cores you'd like to dedicate to training. If the CPU on your machine has, say, eight cores, then eight is the largest number of parallel worker threads you can have. In this case, if you choose to use fewer than eight cores, you're leaving compute resources available for other tasks.

In our GitHub repository, we saved our model using the `save()` method of word2vec objects:

```
model.save('clean_gutenberg_model.w2v')
```

Instead of running word2vec yourself, then, you're welcome to load up our word vectors using this code:

```
model = gensim.models.Word2Vec.load('clean_gutenberg_model.w2v')
```

If you do choose the word vectors we created, then the following examples will produce the same outputs.21 We can see the size of our vocabulary by calling len(model.wv.vocab). This tells us that there are 10,329 words (well, more specifically, tokens) that occur at least 10 times within our clean_sents corpus. One of the words in our vocabulary is *dog*. As shown in Figure 11.6, we can output its location in 64-dimensional word-vector space by running model.wv['dog'].

As a

```
array([ 0.38401067,  0.01232518, -0.37594706, -0.00112308,  0.38663676,
        0.01287549,  0.398965  ,  0.0096426 , -0.10419296, -0.02877572,
        0.3207022 ,  0.27838793,  0.62772304,  0.34408906,  0.23356602,
        0.24557391,  0.3398472 ,  0.07168821, -0.18941355, -0.10122284,
       -0.35172758,  0.4038952 , -0.12179806,  0.096336  ,  0.00641343,
        0.02332107,  0.7743452 ,  0.03591069, -0.20103034, -0.1688079 ,
       -0.01331445, -0.29832968,  0.08522387, -0.02750671,  0.32494134,
       -0.14266558, -0.4192913 , -0.09291836, -0.23813559,  0.38258648,
        0.11036541,  0.005807  , -0.16745028,  0.34308755, -0.20224966,
       -0.77683043,  0.05146591, -0.5883941 , -0.0718769 , -0.18120563,
        0.00358319, -0.29351747,  0.153776  ,  0.48048878,  0.22479494,
        0.5465321 ,  0.29695514,  0.00986911, -0.2450937 , -0.19344331,
        0.3541134 ,  0.3426432 , -0.10496043,  0.00543602], dtype=float32)
```

Figure 11.6 The location of the token "dog" within the 64-dimensional word-vector space we generated using a corpus of books from Project Gutenberg

rudimentary intrinsic evaluation of the quality of our word vectors, we can use the most_similar() method to confirm that words with similar meanings are found in similar locations within our word-vector space.23 For example, to output the three words that are most similar to *father* in our word-vector space, we can run this code:

```
model.wv.most_similar('father', topn=3)
```

This outputs the following:

```
[('mother', 0.8257375359535217),
 ('brother', 0.7275018692016602),
 ('sister', 0.7177823781967163)]
```

This output indicates that mother, brother, and sister are the most similar words to father in our word-vector space. In other words, within our 64-dimensional space, the word that is closest24 to father is the word mother. Table 11.2 provides some additional examples of the words most similar to (i.e., closest to) particular words that we've picked from our wordvector vocabulary,

all five of which appear pretty reasonable given our small Gutenberg corpus.25 Suppose we run the following line of code:

```
model.wv.doesnt_match("mother father sister brother dog".split())
```

We get the output `dog`, indicating that *dog* is the least similar relative to all the other possible word pairs. We can also use the following line to observe that the similarity score between *father* and *dog* is a mere `0.44`:

```
model.wv.similarity('father', 'dog')
```

Table 11.2 **The words most similar to select test words from our Project Gutenberg vocabulary**

Test Word	Most Similar Word	Cosine Similarity Score
father	mother	0.82
dog	puppy	0.78
eat	drink	0.83
day	morning	0.76
ma_am	madam	0.85

This similarity score of `0.44` is much lower than the similarity between *father* and any of *mother*, *brother*, or *sister*, and so it's unsurprising that *dog* is relatively distant from the other four words within our word-vector space.

As a final little intrinsic test, we can compute word-vector analogies as in Figure 2.7. For example, to calculate $v_{father} - v_{man} + v_{woman}$, we can execute this code:

```
model.wv.most_similar(positive=['father', 'woman'], negative=['man'])
```

The top-scoring word comes out as `mother`, which is the correct answer to the analogy. Suppose we likewise execute this code:

```
model.wv.most_similar(positive=['husband', 'woman'], negative=['man'])
```

In this case, the top-scoring word comes out as `wife`, again the correct answer, thereby suggesting that our word-vector space may generally be on the right track.

Plotting Word Vectors

Word vectors typically exist in high-dimensional spaces (e.g., 64, 100, or 300 dimensions), which humans cannot easily visualize. Dimensionality reduction techniques like t-distributed stochastic neighbor embedding (t-SNE) help by mapping these high-dimensional word vectors down to two or three dimensions for visualization.

Key points about t-SNE for word vector visualization:

- n_components sets the output dimensionality: 2D or 3D projections.

- n_iter controls the number of optimization iterations: more iterations may improve the embedding quality but increase computation time.
- t-SNE preserves the local similarities of points, meaning that semantically similar words tend to cluster together in the reduced visualization space.
- This technique allows visual intuition about the relationships between word vectors, such as grouping synonyms or conceptually related words.
- In practice, after generating word embeddings (e.g., with word2vec), run t-SNE on the embedding vectors, then plot the resulting coordinates, often annotating points with words to interpret clusters visually.

Thus, t-SNE is a powerful tool to gain insights into high-dimensional word embeddings by providing an interpretable 2D or 3D visual representation.

Example 11.9 t-SNE for dimensionality reduction

```
tsne = TSNE(n_components=2, n_iter=1000)
X_2d = tsne.fit_transform(model.wv[model.wv.vocab])
coords_df = pd.DataFrame(X_2d, columns=['x','y'])
coords_df['token'] = model.wv.vocab.keys()
```

Running t-SNE as in Example 11.9 may take some time on your machine, so you're welcome to use our results if you're feeling impatient by running the following code:[27,28]

```
coords_df = pd.read_csv('clean_gutenberg_tsne.csv')
```

Whether you ran t-SNE to produce `coords_df` on your own or you loaded in ours, you can check out the first few lines of the DataFrame by using the `head()` method:

```
coords_df.head()
```

Our output from executing `head()` is shown in Figure 11.7.

Example 11.10 provides code for creating a static scatterplot (Figure 11.8) of the two-dimensional data we created with t-SNE (in Example 11.9).

Example 11.10 Static two-dimensional scatterplot of word-vector space

```
_ = coords_df.plot.scatter('x', 'y', figsize=(12,12),
                           marker='.', s=10, alpha=0.2)
```

	x	y	token
0	62.494060	8.023034	emma
1	8.142966	33.342200	by
2	62.507140	10.078477	jane
3	12.477635	17.998343	volume
4	25.736960	30.876250	i

Figure 11.7 This is a Pandas DataFrame containing a two-dimensional representation of the word-vector space we created from the Project Gutenberg corpus. Each unique token has an *x* and *y* coordinate.

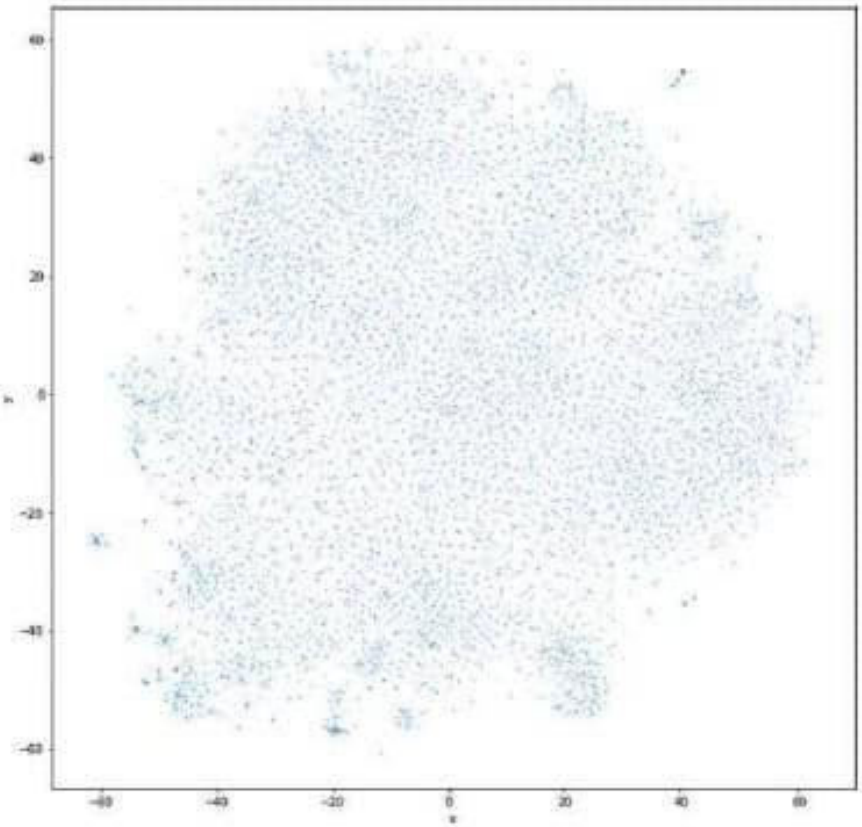


Figure 11.8 Static two-dimensional word-vector scatterplot

On its own, the scatterplot displayed in Figure 11.8 may look interesting, but there's little actionable information we can take away from it. Instead, we recommend using the bokeh library to create a highly interactive—and actionable—plot, as with the code provided in Example 11.11

Example 11.11 Interactive bokeh plot of two-dimensional word-vector data

```
output_notebook()
subset_df = coords_df.sample(n=5000)
p = figure(plot_width=800, plot_height=800)
_ = p.text(x=subset_df.x, y=subset_df.y, text=subset_df.token)
show(p)
```

The code in Example 11.11 produces the interactive scatterplot in Figure 11.9 using the x and y coordinates generated using t-SNE.

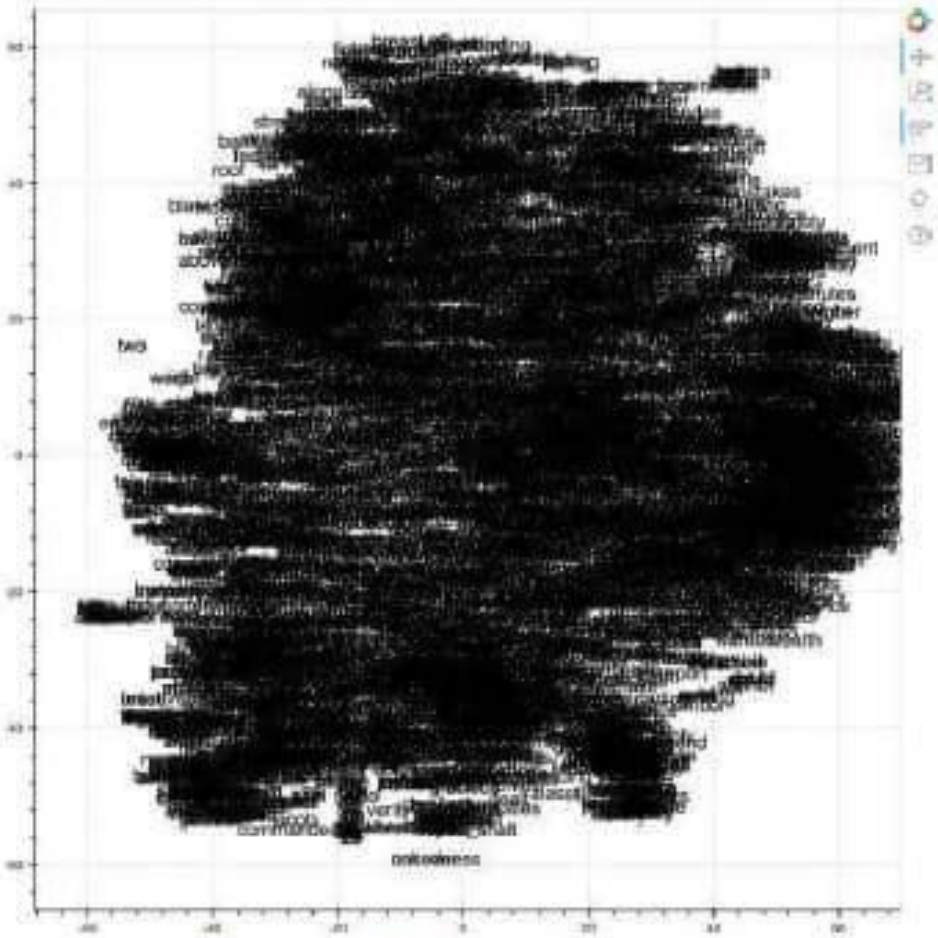


Figure 11.9 Interactive bokeh two-dimensional word-vector plot

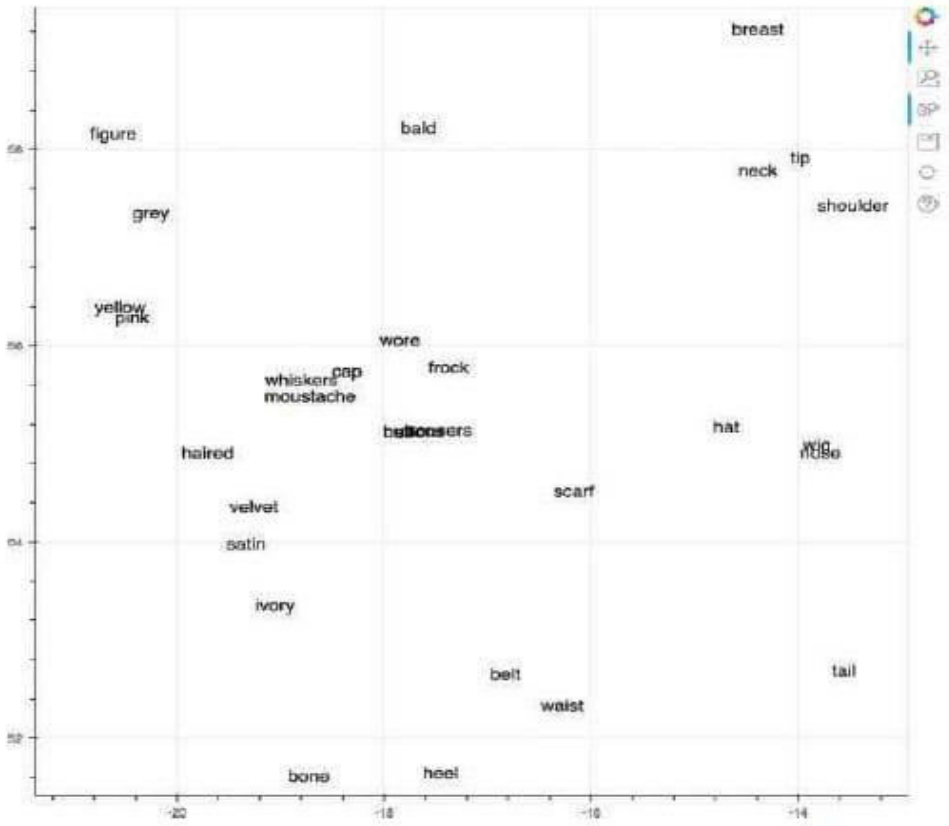


Figure 11.10 Clothing words from the Project Gutenberg corpus, revealed by zooming in to a region of the broader bokeh plot from Figure 11.9

- t-SNE visualizations of word vectors enable interactive exploration of high-dimensional embeddings reduced to 2D/3D space. Use the plot's Wheel Zoom tool to magnify dense word clouds, making individual labels legible for detailed inspection.
    - Interactive zooming reveals semantic clusters, such as:
    - Items of clothing grouped together
    - Nearby related categories (human anatomy parts, colors, fabric types)
- This provides subjective intrinsic evaluation by checking if synonyms and related terms cluster as expected. It also highlights preprocessing issues like unwanted punctuation, bigrams, or tokens appearing in the vocabulary.
- Zooming facilitates qualitative assessment of embedding quality and preprocessing effectiveness without downstream task training

The Area under the ROC Curve

- In machine learning and deep learning, evaluating model performance is critical, especially for classification problems.
- Traditional accuracy metrics may fail when dealing with imbalanced datasets.

- More robust measures the Receiver Operating Characteristic (ROC) curve and its summarized form, the Area Under the Curve (AUC), are widely used.

Limitations of Accuracy Metric

Traditional accuracy applies a fixed threshold (typically $\hat{y} > 0.5$) to classify outputs, missing nuance:

- $\hat{y} = 0.48 \rightarrow$ negative (despite high confidence)
- $\hat{y} = 0.51 \rightarrow$ positive (barely crossing threshold)

ROC AUC: Superior Binary Evaluation

- The ROC (Receiver Operating Characteristic) Curve is a graphical representation that shows the diagnostic ability of a binary classifier system as its discrimination threshold varies.
- The X-axis represents the False Positive Rate (FPR).
- The Y-axis represents the True Positive Rate (TPR) or Sensitivity.

Calculations of the TPR and FPR :

$$\text{TPR} = \frac{TP}{TP + FN}$$

$$\text{FPR} = \frac{FP}{FP + TN}$$

Here,
- **TP** = True Positives
- **FP** = False Positives
- **TN** = True Negatives
- **FN** = False Negatives

Understanding the Area Under the Curve

The AUC is the integral of the ROC curve. It quantifies the overall ability of a model to distinguish between classes.

Formula :

$$AUC = \int_0^1 TPR(FPR^{-1}(x))\, dx$$

Here,

AUC = 1.0 → Perfect classifier.

AUC = 0.5 → Random guessing (no discrimination).

AUC < 0.5 → Classifier performs worse than random.

- Receiver Operating Characteristic Area Under Curve (ROC AUC) evaluates across all thresholds (0.0 to 1.0):

- Combines True Positive Rate (TPR/sensitivity) and False Positive Rate (FPR) into single score.
- Higher AUC indicates better discrimination between classes regardless of threshold.
- Originated in WWII radar detection; insensitive to class imbalance.

Threshold	True Positive	False Positive	True Negative	False Negative	False Positive Rate	True Positive Rate	Area Under ROC Curve
0	0	0	0	0	0	0	0.00600
0.9	20	1	79	0	0.012	1.0	0.01235
0.7	18	2	78	2	0.025	0.9	0.01020
0.5	16	3	77	4	0.037	0.8	0.01875
0.3	14	5	75	6	0.062	0.7	0.79730

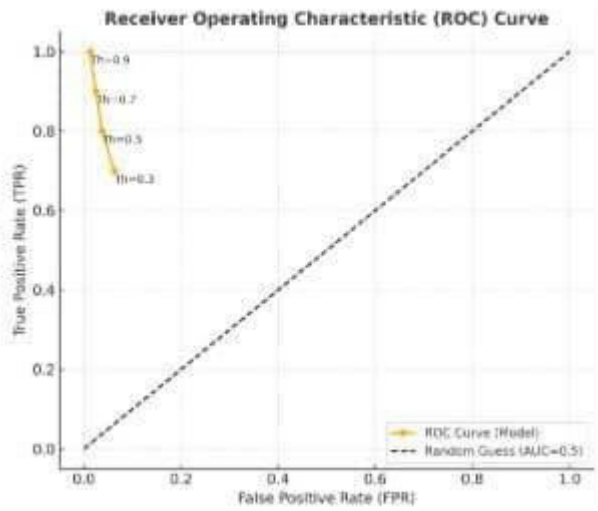
Using the integral formula we calculated the AREA between the values 0-1 .

And the area given by 0.00600+0.01235+0.01020+0.01875+0.79730=0.8446.

Construction of ROC Curve

Steps to construct the ROC Curve

- Compute predicted probabilities for all instances.
- Sort the predictions from highest to lowest.
- Vary the classification threshold from 1 to 0.
- For each threshold, calculate TPR and FPR.
- Plot FPR (X-axis) vs TPR (Y-axis).



The Confusion Matrix

The matrix is a straightforward 2 x 2 table of how confused a model is while attempting to act as a binary classifier.

Table 11.3 **A confusion matrix**

		actual y	
		1	0
predicted y	1	True positive	False positive
	0	False negative	True negative

- To bring the confusion matrix to life with an example, let's return to the hot dog / not hot dog binary classifier that we've used to construct silly examples over many of the preceding chapters:
  - ■ When we provide some input x to a model and it predicts that the input represents a hot dog, then we're dealing with the first row of the table, because the predicted y = 1. In that case,
    - ■ **True positive:** If the input is actually a hot dog (i.e., actual y = 1), then the model correctly classified the input.
    - ■ **False positive:** If the input is actually not a hot dog (i.e., actual y = 0), then the model is confused.
  - ■ When we provide some input x to a model and it predicts that the input does not represent a hot dog, then we're dealing with the second row of the table, because predicted y = 0. In that case,
    - ■ **False negative:** If the input is actually a hot dog (i.e., actual y = 1), then the model is also confused in this circumstance.
    - ■ **True negative:** If the input is actually not a hot dog (i.e., actual y = 0), then the model correctly classified the input.

Calculating the ROC AUC Metric

We provide four inputs to a binary-classification model. Two of these inputs are actually hot dogs (y = 1), and two of them are not hot dogs (y = 0). For each of these inputs, the model outputs some predicted y.

To calculate the ROC AUC metric, we consider each of the $\hat{y}$ values output by the model as the binary-classification threshold in turn. Let's start with the lowest $\hat{y}$, which is 0.3 (see the "0.3 threshold" column in Table 11.5). At this threshold, only the first input

Table 11.4 **Four hot dog / not hot dog predictions**

y	$\hat{y}$
0	0.3
1	0.5
0	0.6
1	0.9

- To calculate the ROC AUC metric we consider each of the y values output by the model as the binary-classification threshold in turn. Let's start with the lowest û, which is 0.3. At this threshold, only the first input is classed as not a hot dog, whereas the second through fourth inputs (all with > 0.3) are all classed as hot dogs.

- We can compare each of these four predicted classifications with the confusion matrix:

  1. True negative (TN): This is actually not a hot dog (y = 0) and was correctly predicted as such.

  2. True positive (TP): This is actually a hot dog (y = 1) and was correctly predicted as such.

  3. False positive (FP): This is actually not a hot dog (y = 0) but it was erroneously predicted to be one.

  4. True positive (TP): Like input 2, this is actually a hot dog (y = 1) and was correctly predicted as such.

- The same process is repeated with the classification threshold set to 0.5 and yet again with the threshold set to 0.6, allowing us to populate the remaining columns.
- The highest î value (in this case, 0.9) can be skipped as a potential threshold, because at such a high threshold we'd be considering all four instances to not be hot dogs, making it a ceiling instead of a classification boundary.

Table 11.5 Four hot dog / not hot dog predictions, now with intermediate ROC AUC calculations

y	$\hat{y}$	0.3 threshold	0.5 threshold	0.6 threshold
0 (not hot dog)	0.3	0 (TN)	0 (TN)	0 (TN)
1 (hot dog)	0.5	1 (TP)	0 (FN)	0 (FN)
0 (not hot dog)	0.6	1 (FP)	1 (FP)	0 (TN)
1 (hot dog)	0.9	1 (TP)	1 (TP)	1 (TP)
True Positive Rate = $\frac{TP}{TP+FN}$		$\frac{2}{2+0}=1.0$	$\frac{1}{1+1}=0.5$	$\frac{1}{1+1}=0.5$
False Positive Rate = $\frac{FP}{FP+TN}$		$\frac{1}{1+1}=0.5$	$\frac{1}{1+1}=0.5$	$\frac{0}{0+2}=0.0$

$$
\begin{aligned}
True\ Positive\ Rate &= \frac{(TP\ count)}{(TP\ count)\ +\ (FN\ count)} \\
&= \frac{2}{2+0} \\
&= \frac{2}{2} \\
&= 1.0
\end{aligned}
$$

$$
\begin{aligned}
False\ Positive\ Rate &= \frac{(FP\ count)}{(FP\ count)\ +\ (TN\ count)} \\
&= \frac{1}{1+1} \\
&= \frac{1}{2} \\
&= 0.5
\end{aligned}
$$

The final stage in calculating ROC AUC is to create a plot. The points that make up the shape of the receiver operating characteristic (ROC) curve are the false positive rate (horizontal, x-axis coordinate) and true positive rate (vertical, y-axis coordinate) at each of the available thresholds, plus two extra points in the bottom-left and top-right corners of the plot. Specifically, these five points are:

(0, 0) for the bottom-left corner

(0, 0.5) from the 0.6 threshold

(0.5, 0.5) from the 0.5 threshold

(0.5, 1) from the 0.3 threshold

(1, 1) for the top-right corner

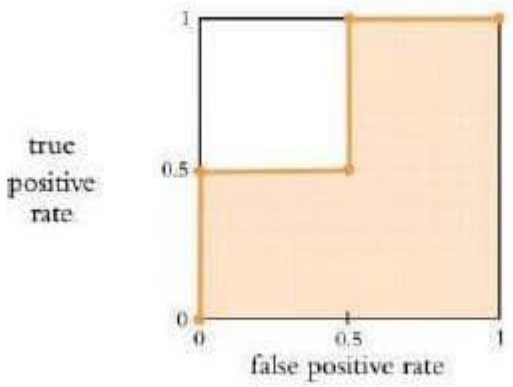


Figure 11.11 The (orange-shaded) area under the curve of the receiving operator characteristic, determined using the TPRs and FPRs from Table 11.5

ROC AUC Interpretation

ROC AUC evaluates binary classifiers across all possible thresholds by plotting True Positive Rate

(TPR) vs. False Positive Rate (FPR):

AUC = 0.5: Random performance (diagonal line, equivalent to coin flip)

AUC = 1.0: Perfect classifier (FPR=0, TPR=1 at all thresholds)

Real-world AUC: Between 0.5-1.0, limited by data noise creating an unreachable "ceiling"

Design goal: Maximize TPR while minimizing FPR across the full y^ range (0.0-1.0).

Evaluation Strategy

Use ROC AUC alongside accuracy and cost metrics for binary NLP classifiers (e.g., sentiment analysis):

Primary: ROC AUC (threshold-independent, comprehensive)

Secondary: Accuracy (simple but threshold-sensitive)

Tertiary: Cost (business-specific error weighting)

This multi-metric approach provides complete performance assessment for deep learning binary models.

Natural Language Classification with Familiar Networks

Loading the IMDb Film Reviews

Example 11.12 provides the dependencies we need for our dense sentiment classifier.

Example 11.12 Loading sentiment classifier dependencies

```
import keras
from keras.datasets import imdb # new!
from keras.preprocessing.sequence import pad_sequences # new!


from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout
from keras.layers import Embedding # new!
from keras.callbacks import ModelCheckpoint # new!
import os # new!
from sklearn.metrics import roc_auc_score, roc_curve # new!
import pandas as pd
import matplotlib.pyplot as plt # new!
%matplotlib inline
```

It's a good programming practice to put as many hyperparameters as you can at the top of your file. This makes it easier to experiment with these hyperparameters. It also makes it easier for you (or, indeed, your colleagues) to understand what you were doing in the f ile when you return to it (perhaps much) later. With this in mind, we place all of our hyperparameters together in a single cell within our Jupyter notebook. The code is provided in Example 11.13.

Example 11.13 Setting dense sentiment classifier hyperparameters

```
# output directory name:
output_dir = 'model_output/dense'

# training:
epochs = 4
batch_size = 128

# vector-space embedding:
n_dim = 64
n_unique_words = 5000
n_words_to_skip = 50
max_review_length = 100
pad_type = trunc_type = 'pre'

# neural network architecture:
n_dense = 64
dropout = 0.5
```

output_dir: A directory name (ideally, a unique one) in which to store our model's parameters after each epoch, allowing us to return to the parameters from any epoch of our choice at a later time.

epochs: The number of epochs that we'd like to train for, noting that NLP models often overfit to the training data in fewer epochs than machine vision models.

batch_size: As before, the number of training examples used during each round of model training

n_dim: The number of dimensions we'd like our word-vector space to have.

n_unique_words: With word2vec we included tokens in our word-vector vocabulary only if they occurred at least a certain number of times within our corpus.

n_words_to_skip: Instead of removing a manually curated list of stop words from their word-vector vocabulary.

max_review_length: Each movie review must have the same length so that TensorFlow knows the shape of the input data that will be flowing through our deep learning model.

pad_type: By selecting 'pre', we add padding characters to the start of every review. The alternative is 'post', which adds them to the end.

trunc_type: As with pad_type, our truncation options are 'pre' or 'post'. The former will remove words from the beginning of the review, whereas the latter will remove them from the end. By selecting 'pre', we're making (a bold!) assumption that the end of film reviews tend to include more information on review sentiment than the beginning.

n_dense: The number of neurons to include in the dense layer of our neural network architecture.

dropout: How much dropout to apply to the neurons in the dense layer.

Loading in the film review data is a one-liner, provided in Example 11.14.

Example 11.14 Loading IMDb film review data

```
(x_train, y_train), (x_valid, y_valid) = \
    imdb.load_data(num_words=n_unique_words, skip_top=n_words_to_skip)
```

The labels (y_train and y_valid) are binary, based on these star ratings:

- Reviews with a score of four stars or fewer are a negative review (y = 0).
- Reviews with a score of seven stars or more, meanwhile, are classed as a positive review (y = 1)
- Moderate reviews: those with five or six stars—are not included in the dataset, making the binary classification task easier for any model.

By specifying values for the num_words and skip_top arguments when calling imdb.load_data(), we are limiting the size of our word-vector vocabulary and removing the most common (stop) words, respectively.

EXAMINING THE IMDb data:

```
array([ [2, 2, 2, 2, 2, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 2,
173, 2, 256, 2, 2, 100, 2, 838, 112, 50, 670, 2, 2, 2, 480, 284, 2, 150,
2, 172, 112, 167, 2, 336, 385, 2, 2, 172, 4536, 1111, 2, 546, 2, 2, 447,
2, 192, 50, 2, 2, 147, 2025, 2, 2, 2, 2, 1920, 4613, 469, 2, 2, 71, 87,
2, 2, 2, 530, 2, 76, 2, 2, 1247, 2, 2, 2, 515, 2, 2, 2, 626, 2, 2, 2, 62,
386, 2, 2, 316, 2, 106, 2, 2, 2223, 2, 2, 480, 66, 3785, 2, 2, 130, 2, 2,
2, 619, 2, 2, 124, 51, 2, 135, 2, 2, 1415, 2, 2, 2, 2, 215, 2, 77, 52, 2,
2, 407, 2, 82, 2, 2, 2, 107, 117, 2, 2, 256, 2, 2, 2, 3766, 2, 723, 2, 7
1, 2, 530, 476, 2, 400, 317, 2, 2, 2, 2, 1029, 2, 104, 88, 2, 381, 2, 29
7, 98, 2, 2071, 56, 2, 141, 2, 194, 2, 2, 2, 226, 2, 2, 134, 476, 2, 480,
2, 144, 2, 2, 2, 51, 2, 2, 224, 92, 2, 104, 2, 226, 65, 2, 2, 1334, 88,
2, 2, 283, 2, 2, 4472, 113, 103, 2, 2, 2, 2, 2, 178, 2],
       [2, 194, 1153, 194, 2, 78, 228, 2, 2, 1463, 4369, 2, 134, 2, 2, 71
5, 2, 118, 1634, 2, 394, 2, 2, 119, 954, 189, 102, 2, 207, 110, 3103, 2,
2, 69, 188, 2, 2, 2, 2, 2, 249, 126, 93, 2, 114, 2, 2300, 1523, 2, 647,
2, 116, 2, 2, 2, 2, 229, 2, 340, 1322, 2, 118, 2, 2, 130, 4901, 2, 2, 100
2, 2, 89, 2, 952, 2, 2, 2, 455, 2, 2, 2, 2, 1543, 1905, 398, 2, 1649, 2,
2, 2, 163, 2, 3215, 2, 2, 1153, 2, 194, 775, 2, 2, 2, 349, 2637, 148, 60
5, 2, 2, 2, 123, 125, 68, 2, 2, 2, 349, 165, 4362, 98, 2, 2, 228, 2, 2,
2, 1157, 2, 299, 120, 2, 120, 174, 2, 220, 175, 136, 50, 2, 4373, 228, 2,
2, 2, 656, 245, 2350, 2, 2, 2, 131, 152, 491, 2, 2, 2, 2, 1212, 2, 2, 2,
371, 78, 2, 625, 64, 1382, 2, 2, 168, 145, 2, 2, 1690, 2, 2, 2, 1355, 2,
2, 2, 52, 154, 462, 2, 89, 78, 285, 2, 145, 95],
```

Figure 11.12 The first two film reviews from the training dataset of Andrew Maas and colleagues' (2011) IMDb dataset. Tokens are in an integer-index format.

Executing x_train[0:6], we can examine the first six reviews from the training dataset, the first two of which are shown in Figure 11.12. These reviews are natively in an integer-index format, where each unique token from the dataset is represented by an integer.

0: Reserved as the padding token (which we'll soon add to the reviews that are shorter than max_review_length).

1: Would be the starting token, which would indicate the beginning of a review. As per the next bullet point, however, the starting token is among the top 50 most common to kens and so is shown as "unknown."

2: Any tokens that occur very frequently across the corpus (i.e., they're in the top 50 most common words) or rarely (i.e., they're below the top 5,050 most common words) will be outside of our word-vector vocabulary and so are replaced with this unknown token.

3: The most frequently occurring word in the corpus.

4: The second-most frequently occurring word.

5: The third-most frequently occurring, and so on.

Using the following code from Example 11.15, we can see the length of the first six reviews in the training dataset,

Example 11.15 Printing the number of tokens in six reviews

```
for x in x_train[0:6]:
    print(len(x))
```

To view the reviews as natural language, we create an index of words as follows, where PAD, START, and UNK are customary for representing padding, starting, and unknown tokens, respectively:

```
word_index = keras.datasets.imdb.get_word_index()
word_index = {k:(v+3) for k,v in word_index.items()}
word_index["PAD"] = 0
word_index["START"] = 1
word_index["UNK"] = 2
index_word = {v:k for k,v in word_index.items()}
```

Then we can use the code in Example 11.16 to view the film review of our choice—in this case, the first review from the training data.

Example 11.16 Printing a review as a character string

```
' '.join(index_word[id] for id in x_train[0])
```

The resulting string should look identical to the output shown in Figure 11.13

```
"UNK UNK UNK UNK UNK brilliant casting location scenery story direction e
veryone's really suited UNK part UNK played UNK UNK could UNK imagine bei
ng there robert UNK UNK UNK amazing actor UNK now UNK same being director
UNK father came UNK UNK same scottish island UNK myself UNK UNK loved UNK
fact there UNK UNK real connection UNK UNK UNK UNK witty remarks througho
ut UNK UNK were great UNK UNK UNK brilliant UNK much UNK UNK bought UNK U
NK UNK soon UNK UNK UNK released UNK UNK UNK would recommend UNK UNK ever
yone UNK watch UNK UNK fly UNK UNK amazing really cried UNK UNK end UNK U
NK UNK sad UNK UNK know what UNK say UNK UNK cry UNK UNK UNK UNK must UNK
been good UNK UNK definitely UNK also UNK UNK UNK two little UNK UNK play
ed UNK UNK UNK norman UNK paul UNK were UNK brilliant children UNK often
left UNK UNK UNK UNK list UNK think because UNK stars UNK play them UNK g
rown up UNK such UNK big UNK UNK UNK whole UNK UNK these children UNK ama
zing UNK should UNK UNK UNK what UNK UNK done don't UNK think UNK whole s
tory UNK UNK lovely because UNK UNK true UNK UNK someone's life after UNK
UNK UNK UNK UNK us UNK"
```

Figure 11.13 The first film review from the training dataset, now shown as a character string

With our index of words (index_words) already available to us, we simply need to download the full reviews:

(all_x_train,_),(all_x_valid,_) = imdb.load_data()

Then we modify Example 11.16 to execute join() on the full-review list of our choice (i.e., all_x_train or all_x_valid), as provided in Example 11.17.

Example 11.17 Print full review as character string

' '.join(index_word[id] for id in all_x_train[0])

Executing this outputs the full text of the review of our choice—again, in this case, the first training review—as shown in Figure 11.14.

```
"START this film was just brilliant casting location scenery story direct
ion everyone's really suited the part they played and you could just imag
ine being there robert redford's is an amazing actor and now the same bei
ng director norman's father came from the same scottish island as myself
so i loved the fact there was a real connection with this film the witty
remarks throughout the film were great it was just brilliant so much that
i bought the film as soon as it was released for retail and would recomme
nd it to everyone to watch and the fly fishing was amazing really cried a
t the end it was so sad and you know what they say if you cry at a film i
t must have been good and this definitely was also congratulations to the
two little boy's that played the part's of norman and paul they were just
brilliant children are often left out of the praising list i think becaus
e the stars that play them all grown up are such a big profile for the wh
ole film but these children are amazing and should be praised for what th
ey have done don't you think the whole story was so lovely because it was
true and was someone's life after all that was shared with us all"
```

Figure 11.14 The first film review from the training dataset, now shown in full as a character string

STANDARDIZING THE LENGTH OF THE REVIEWS

Keras provides a convenient pad_sequences() method that enables us to both pad and truncate documents of text in a single line.

we standardize our training and validation data in this way, as shown in Example 11.18.

Example 11.18 Standardizing input length by padding and truncating

```
x_train = pad_sequences(x_train, maxlen=max_review_length,
                        padding=pad_type, truncating=trunc_type, value=0)
x_valid = pad_sequences(x_valid, maxlen=max_review_length,
                        padding=pad_type, truncating=trunc_type, value=0)
```

when printing reviews (e.g., with x_train[0:6]) or their lengths (e.g., with the code from Example 11.15), we see that all of the reviews have the same length of 100 (because we set max_review_length = 100). Examining x_train[5]—which previously had a length of only 43 tokens—with code similar to Example 11.16, we can observe that the beginning of the review has been padded with 57 PAD tokens (see Figure 11.15).

```
'PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD
PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD P
AD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PA
D PAD PAD UNK begins better than UNK ends funny UNK UNK russian UNK crew
UNK UNK other actors UNK UNK those scenes where documentary shots UNK UNK
spoiler part UNK message UNK UNK contrary UNK UNK whole story UNK UNK doe
s UNK UNK UNK UNK'
```

Figure 11.15 The sixth film review from the training dataset, padded with the PAD token at the beginning so that—like all the other reviews—it has a length of 100 tokens

DENSE NETWORK:

A neural network architecture to classify film reviews by their sentiment. A baseline bdense network model for this task is shown in Example 11.19.

Example 11.19 Dense sentiment classifier architecture

```
model = Sequential()
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(Flatten())
model.add(Dense(n_dense, activation='relu'))
model.add(Dropout(dropout))
# model.add(Dense(n_dense, activation='relu'))
# model.add(Dropout(dropout))
model.add(Dense(1, activation='sigmoid'))
```

Let's break the architecture down line by line:

- Keras Sequential() method to invoke a sequential model
- Embedding() layer enables us to create word vectors from a corpus of documents—in this case, the 25,000 movie reviews of the IMDb training dataset.
- Flatten() layer enables us to pass a many-dimensional output (here, a two-dimensional output from the embedding layer) into a onedimensional dense layer.
- Dense() layers, we used a single one consisting of relu activations in this architecture, with Dropout() applied to it.
- neuron is sigmoid because we'd like it to output probabilities between 0 and 1

model.summary(), we discover that our fairly simple NLP model has quite a few parameters, as shown in Figure 11.16:

In the embedding layer, the 320,000 parameters come from having 5,000 words, each one with a location specified in a 64-dimensional word-vector space (64 × 5,000 = 320,000).

Each of our film-review inputs consists of 100 tokens, with each token specified by 64 word-vector-space coordinates (64 × 100 = 6,400).

Each of the 64 neurons in the dense hidden layer receives input from each of the 6,400 values flowing out of the flatten layer, for a total of 64 × 6,400 = 409,600

the single neuron of the output layer has 64 weights—one for the activation output by each of the neurons in the preceding layer—plus its bias, for a total of 65 parameters.

Summing up the parameters from each of the layers, we have a grand total of 730,000 of them.

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 100, 64)	320000
flatten_1 (Flatten)	(None, 6400)	0
dense_1 (Dense)	(None, 64)	409664
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 1)	65

Total params: 729,729
Trainable params: 729,729
Non-trainable params: 0

Figure 11.16 Dense sentiment classifier model summary

Example 11.20 **Compiling our sentiment classifier**

```
model.compile(loss='binary_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

As shown in Example 11.20, we compile our dense sentiment classifier with a line of code that should already be familiar from recent chapters, except that—because we have a single output neuron within a binary classifier—we use binary_crossentropy cost in place of the categorical_crossentropy cost we used for our multiclass MNIST classifiers.

Example 11.21, we create a ModelCheckpoint() object that will allow us to save our model parameters after each epoch during training. By doing this, we can return to the parameters from our epoch of choice later on during model evaluation or to make inferences in a production system. If the output_dir directory doesn't already exist, we use the makedirs() method to make it.

Example 11.21 Creating an object and directory for checkpointing model parameters after each epoch

```
modelcheckpoint = ModelCheckpoint(filepath=output_dir+
                                  "/weights.{epoch:02d}.hdf5")
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
```

Example 11.22 Fitting our sentiment classifier

```
model.fit(x_train, y_train,
          batch_size=batch_size, epochs=epochs, verbose=1,
          validation_data=(x_valid, y_valid),
          callbacks=[modelcheckpoint])
```

To evaluate the results of the best epoch more thoroughly, we use the Keras load_ weights() method to load the parameters from the second epoch (weights.02.hdf5) back into our model, as in Example 11.23.

Example 11.23 Loading model parameters

```
model.load_weights(output_dir+"/weights.02.hdf5")
```

We can then calculate validation set $\hat{y}$ values for the best epoch by passing the `predict_proba()` method on the `x_valid` dataset, as shown in Example 11.24.

Example 11.24 Predicting $\hat{y}$ for all validation data

```
y_hat = model.predict_proba(x_valid)
```

With y_hat[0], for example, we can now see the model's prediction of the sentiment of the first movie review in the validation set. For this review, yˆ = 0.09, indicating the model estimates that there's a 9 percent chance the review is positive and, therefore, a 91 percent chance it's negative. Executing y_valid[0] informs us that yˆ = 0 for this review—that is, it is in fact a negative review—so the model's yˆ is pretty good! If you're curious about what the content of the negative review was, you can run a slight modification on Example 11.17 to access the full text of the all_x_valid[0] list item, as shown in Example 11.25.

Example 11.25 Printing a full validation review

```
' '.join(index_word[id] for id in all_x_valid[0])
```

Example 11.26 Plotting a histogram of validation data $\hat{y}$ values

```
plt.hist(y_hat)
_ = plt.axvline(x=0.5, color='orange')
```

The histogram output is provided in Figure 11.18. The plot shows that the model often has a strong opinion on the sentiment of a given review: Some 8,000 of the 25,000 reviews (~32 percent of them) are assigned a yˆ of less than 0.1, and ~6,500 (~26 percent) are given a yˆ greater than 0.9

The vertical orange line in Figure 11.18 marks the 0.5 threshold above which reviews are considered by a simple accuracy calculation to be positive.

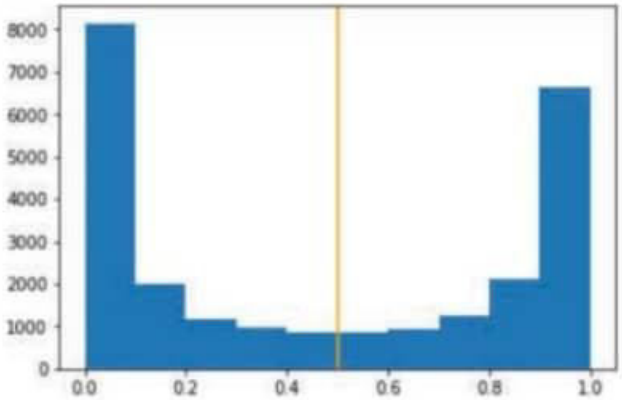


Figure 11.18 Histogram of validation data $\hat{y}$ values for the second epoch of our dense sentiment classifier

Example 11.27 Calculating ROC AUC for validation data

```
pct_auc = roc_auc_score(y_valid, y_hat)*100.0
"{:0.2f}".format(pct_auc)
```

Printing the output in an easy-to-read format with the format() method, we see that the percentage of the area under the receiver operating characteristic curve is (a fairly high) 92.9 percent.

Example 11.28 Creating a ydf DataFrame of y and $\hat{y}$ values

```
float_y_hat = []
for y in y_hat:
    float_y_hat.append(y[0])
ydf = pd.DataFrame(list(zip(float_y_hat, y_valid)),
                   columns=['y_hat', 'y'])
```

Printing the first 10 rows of the resulting ydf DataFrame with ydf.head(10), we see the output shown in Figure 11.19.

	y_hat	y
0	0.089684	0
1	0.982754	1
2	0.746905	1
3	0.543328	0
4	0.997054	1
5	0.833994	1
6	0.766254	1
7	0.008032	0
8	0.812743	0
9	0.729463	1

Figure 11.19 DataFrame of y and $\hat{y}$ values for the IMDb validation data

Example 11.29 Ten cases of negative validation reviews with high $\hat{y}$ scores

```
ydf[(ydf.y == 0) & (ydf.y_hat > 0.9)].head(10)
```

Example 11.30 Ten cases of positive validation reviews with low $\hat{y}$ scores

```
ydf[(ydf.y == 0) & (ydf.y_hat > 0.9)].head(10)
```

An example of a false positive—a negative review (y = 0) with a very high model score (ŷ = 0.97)—that was identified by running the code in Example 11.29 is provided in Figure 11.20.

```
"START wow another kevin costner hero movie postman tin cup waterworld bo
dyguard wyatt earp robin hood even that baseball movie seems like he make
s movies specifically to be the center of attention the characters are al
most always the same the heroics the flaws the greatness the fall the red
emption yup within the 1st 5 minutes of the movie we're all supposed to b
e in awe of his character and it builds up more and more from there br br
and this time the story story is just a collage of different movies you d
on't need a spoiler you've seen this movie several times though it had di
fferent titles you'll know what will happen way before it happens this is
like mixing an officer and a gentleman with but both are easily better mo
vies watch to see how this kind of movie should be made and also to see h
ow an good but slightly underrated actor russell plays the hero"
```

Figure 11.20 An example of a false positive: This negative review was misclassified as positive by our model.

CONVOLUTIONAL NETWORKS

convolutional layers are particularly adept at detecting spatial patterns.

Here, we use them to detect spatial patterns among words like the not-good sequence and see whether they can improve upon the performance of our dense network at classifying film reviews by their sentiment.

The dependencies for this model are identical to those of our dense sentiment classifier, except that it has three new Keras layer types, as provided in Example 11.31.

Example 11.31 Additional CNN dependencies

```
from keras.layers import Conv1D, GlobalMaxPooling1D
from keras.layers import SpatialDropout1D
```

The hyperparameters for our convolutional sentiment classifier are provided in Example 11.32.

Example 11.32 Convolutional sentiment classifier hyperparameters

```
# output directory name:
output_dir = 'model_output/conv'

# training:
epochs = 4
batch_size = 128

# vector-space embedding:
n_dim = 64
n_unique_words = 5000
max_review_length = 400
pad_type = trunc_type = 'pre'
drop_embed = 0.2 # new!

# convolutional layer architecture:
n_conv = 256 # filters, a.k.a. kernels
k_conv = 3 # kernel length

# dense layer architecture:
n_dense = 256
dropout = 0.2
```

unique directory name ('conv') for storing model parameters after each epoch of training.

Our number of epochs and batch size remain the same.

Our vector-space embedding hyperparameters remain the same, except that We quadrupled max_review_length to 400. With drop_embed, we'll be adding dropout to our embedding layer.

convolutional sentiment classifier will have two hidden layers after the embedding layer:

- A convolutional layer with 256 filters (n_conv), each with a single dimension (a length) of 3 (k_conv).
- A dense layer with 256 neurons (n_dense) and dropout of 20 percent.

Example 11.33 Convolutional sentiment classifier architecture

```
model = Sequential()

# vector-space embedding:
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(SpatialDropout1D(drop_embed))

# convolutional layer:
model.add(Conv1D(n_conv, k_conv, activation='relu'))
# model.add(Conv1D(n_conv, k_conv, activation='relu'))
model.add(GlobalMaxPooling1D())

# dense layer:
model.add(Dense(n_dense, activation='relu'))
model.add(Dropout(dropout))

# output layer:
model.add(Dense(1, activation='sigmoid'))
```

embedding layer is the same as before, except that it now has dropout applied to it. Flatten(), because the Conv1D() layer takes both dimensions of the embedding layer output.

Relu activation within our one-dimensional convolutional layer. The layer has 256 unique filters, each of which is free to specialize in activating when it passes over a particular three-token sequence. The activation map for each of the 256 filters has a length of 398, for a 256×398 output shape.

Global max-pooling is common for dimensionality reduction within deep learning NLP models.

NETWORKS DESIGNED FOR SEQUENTIAL DATA

RECURRENT NEURAL NETWORKS:

Consider the following sentences: Jon and Grant are writing a book together. They really enjoyed writing it. The human mind can track the concepts in the second sentence quite easily. You already know that “they” in the second sentence refers to your authors, and “it” refers to the book we’re writing. Although this task is easy for you, however, it is not so trivial for a neural network.

The high-level structure of a recurrent neural network (RNN) is shown in Figure 11.25. On the left, the purple line indicates the loop that passes information between steps in the network. As in a dense network, where there is a neuron for each input, so too is there a neuron for each input here.

In the case of Figure 11.25, each word is represented by a distinct timestep in the RNN sequence, so the network might be able to learn that "Jon" and "Grant" were writing the book, thereby associating these terms with the word "they" that occurs later in the sequence.

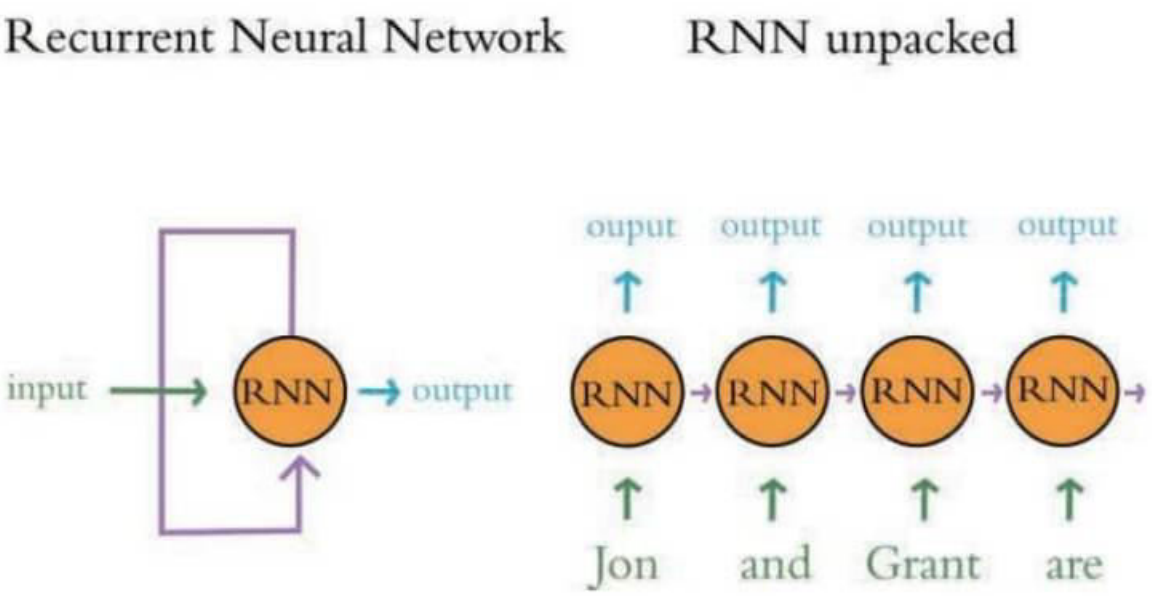


Figure 11.25 Schematic diagram of a recurrent neural network

Implementing RNN in Keras:

Adding a recurrent layer to a neural network architecture to create an RNN is straightforward in Keras, as we illustrate in our RNN Sentiment Classifier.

For the sake of brevity and readability, please note that the following code cells are identical across all the Jupyter notebooks in this chapter, including the Dense and Convolutional Sentiment Classifier notebooks.

The code cells that vary are those in which we:

1. Set hyperparameters
2. Design the neural network architecture

The hyperparameters for our RNN are as shown in Example 11.34

Example 11.34 RNN sentiment classifier hyperparameters

```
# output directory name:
output_dir = 'model_output/rnn'

# training:
epochs = 16 # way more!
batch_size = 128

# vector-space embedding:
n_dim = 64
n_unique_words = 10000
max_review_length = 100 # lowered due to vanishing gradient over time
pad_type = trunc_type = 'pre'
drop_embed = 0.2

# RNN layer architecture:
n_rnn = 256
drop_rnn = 0.2
```

Our RNN model architecture is provided in Example 11.35.

Example 11.35 RNN sentiment classifier architecture

```
from keras.layers import SimpleRNN

model = Sequential()
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(SpatialDropout1D(drop_embed))
model.add(SimpleRNN(n_rnn, dropout=drop_rnn))
model.add(Dense(1, activation='sigmoid'))
```

Long Short term Memory Units:

LSTMs were introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997,45 but they are more widely used in NLP deep learning applications today than ever before. The basic structure of an LSTM layer is the same as the simple recurrent layers captured in Figure 11.25.

LSTMs receive input from the sequence of data (e.g., a particular token from a natural language document), and they also receive input from the previous time point in the sequence. The

difference is that inside each cell in a simple recurrent layer (e.g., SimpleRNN() in Keras), you'll find a single neural network activation function such as a tanh function, which transforms the RNN cell's inputs to generate its output. In contrast, the cells of an LSTM layer contain a far more complex structure, as depicted in Figure 11.26

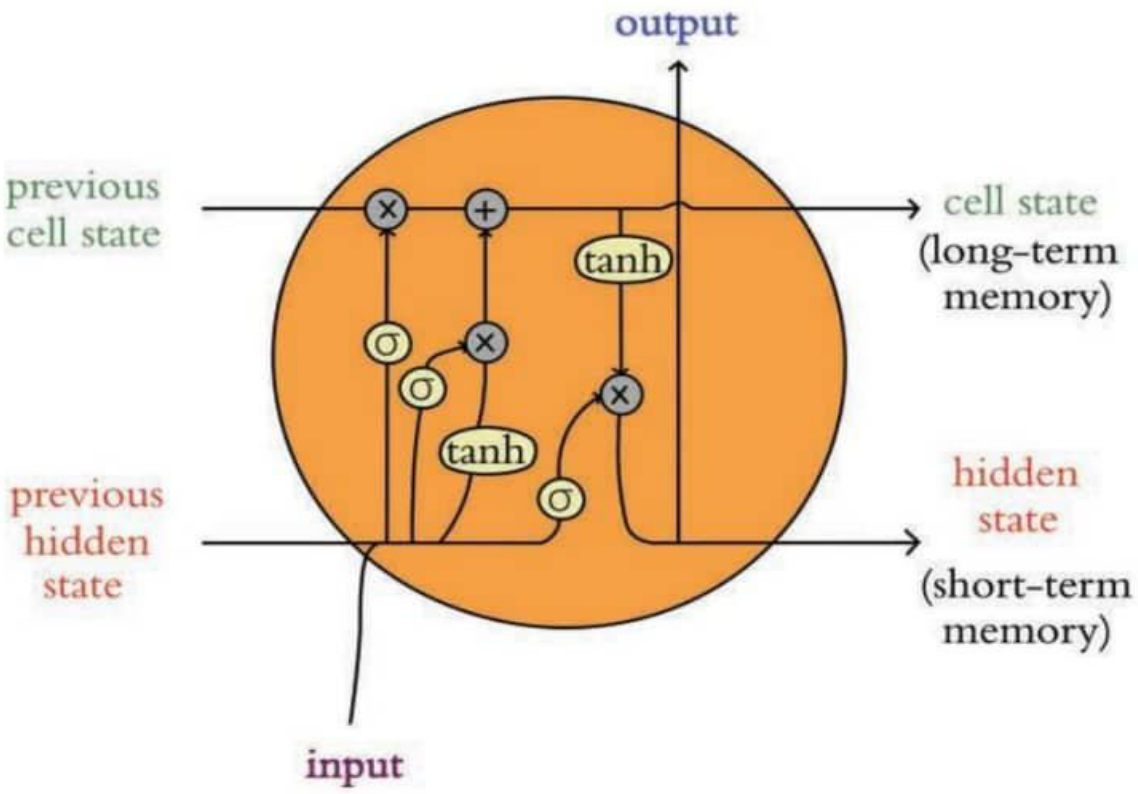


Figure 11.26 Schematic diagram of an LSTM

Those two linear transformations (a multiplication and an addition operation) are points where a cell in an LSTM layer can add information to the cell state, information that will be passed onto the next cell in the layer. In either case, there is a sigmoid activation (represented by σ in the figure) before the information is added to the cell state. Because a sigmoid activation produces values between 0 and 1, these sigmoids act as "gates" that decide whether new information (from the current timestep) is added to the cell state or not.

The new information at the current timestep is a simple concatenation of the current timestep's input and the hidden state from the preceding timestep. This concatenation has two chances to be incorporated into the cell state—either linearly or following a nonlinear tanh activation—and in either case it's those sigmoid gates that decide whether the information is combined.

After the LSTM has determined what information to add to the cell state, another sigmoid gate decides whether the information from the current input is added to the final cell state, and this results in the output for the current timestep.

Another way to distill this LSTM content is:

■ The cell state enables information to persist along the length of the sequence, through each timestep in a given LSTM cell. It is the long-term memory of the LSTM.

■ The hidden state is analogous to the recurrent connections in a simple RNN and represents the short-term memory of the LSTM.

■ Each module represents a particular point in the sequence of data (e.g., a particular token from a natural language document).

■ At each timestep, several decisions are made (using those sigmoid gates) about whether the information at that particular timestep in the sequence is relevant to the local (hidden state) and global (cell state) contexts.

■ The first two sigmoid gates determine whether the information from the current timestep is relevant to the global context (the cell state) and how it will be combined into that stream. ■ The final sigmoid gate determines whether the information from the current timestep is relevant to the local context (i.e., whether it is added to the hidden state, which doubles as the output for the current timestep)

Implementing LSTM with Keras:

Example 11.36 LSTM sentiment classifier hyperparameters

```
# output directory name:
output_dir = 'model_output/LSTM'


epochs = 4
batch_size = 128

# vector-space embedding:
n_dim = 64
n_unique_words = 10000
max_review_length = 100
pad_type = trunc_type = 'pre'
drop_embed = 0.2

# LSTM layer architecture:
n_lstm = 256
drop_lstm = 0.2
```

LSTM model architecture is also the same as our RNN architecture, except that we replaced the SimpleRNN() layer with LSTM(); see Example 11.37.

Example 11.37 LSTM sentiment classifier architecture

```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(SpatialDropout1D(drop_embed))
model.add(LSTM(n_lstm, dropout=drop_lstm))
model.add(Dense(1, activation='sigmoid'))
```

Bidirectional LSTMs

Bidirectional LSTMs (or Bi-LSTMs, for short) are a clever variation on standard LSTMs. bidirectional LSTMs involve backpropagation in both directions (backward and forward over timesteps) across some one-dimensional input.

Bi-LSTMs are a popular choice in modern NLP applications because their ability to learn patterns both before and after a given token within an input document facilitates high - performing models.

Converting our LSTM architecture (Example 11.37) into a Bi-LSTM architecture is painless. We need only wrap our LSTM() layer within the Bidirectional() wrapper, as shown in Example 11.38.

Example 11.38 Bidirectional LSTM sentiment classifier architecture

```
from keras.layers import LSTM
from keras.layers.wrappers import Bidirectional # new!

model = Sequential()
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(SpatialDropout1D(drop_embed))
model.add(Bidirectional(LSTM(n_lstm, dropout=drop_lstm)))
model.add(Dense(1, activation='sigmoid'))
```

Stacked Recurrent Models:

Stack recurrent layers, use the argument return_sequences=True.

This asks the recurrent layer to return the hidden states for each step in the layer's sequence. The resulting output now has three dimensions, matching the dimensions of the input sequence that was fed into it. The default behavior of a recurrent layer is to pass only the final hidden state to the next layer.

This works perfectly well if we're passing this information to, say, a dense layer. If, however, we'd like the subsequent layer in our network to be another recurrent layer, that subsequent recurrent layer must receive a sequence as its input.

Thus, to pass the array of hidden states from across all individual timesteps in the sequence (as opposed to only the single final hidden state value) to this subsequent recurrent layer, we set the optional return_sequences argument to True.

Example 11.39 Stacked recurrent model architecture

```
from keras.layers import LSTM
from keras.layers.wrappers import Bidirectional

model = Sequential()
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(SpatialDropout1D(drop_embed))
model.add(Bidirectional(LSTM(n_lstm_1, dropout=drop_lstm,
                             return_sequences=True))) # new!
model.add(Bidirectional(LSTM(n_lstm_2, dropout=drop_lstm)))
model.add(Dense(1, activation='sigmoid'))
```

A **stacked recurrent model** is an RNN architecture in which the **output of one recurrent layer becomes the input to the next recurrent layer**.

It allows the network to learn **hierarchical representations** of sequential data.

Why stack RNN layers?

Stacking gives the model more power:

Low-level sequence patterns

Earlier layers capture basic features (e.g., word patterns, local temporal features).

High-level patterns

Upper layers capture deeper semantics (e.g., sentence meaning, long-term trends).

Better representation learning

More layers = more abstract understanding of the sequence.

Seq2Seq and Attention

Seq2Seq (sequence-to-sequence) models take one sequence as input and generate another sequence as output.

Examples:

- Translating English → French
- Converting a sentence → summary
- Converting speech → text

Seq2Seq models are often used in **Neural Machine Translation (NMT)** like Google Translate.

A Seq2Seq model has two parts:

Encoder

- Reads the entire input sentence.
- Produces a final hidden state called **context**.

Decoder

- Uses this context to generate the output sentence.

But what is the problem?

The encoder gives **only one vector** (the final hidden state) to the decoder.

This is like:

- Reading a long paragraph,
- And then trying to translate it *only from memory* of the last line.

So long sentences become difficult because the single context vector cannot hold everything.

This is called the **bottleneck problem**.

Why was Attention introduced?

Attention solves the bottleneck.

Instead of giving only **one final hidden state**, attention gives the decoder **all hidden states** from the encoder.

How Attention Works (in simple steps)

Imagine the decoder is writing a translation.

For each output word, it can **look back** at any part of the input sentence.

Step-by-step:

1. Encoder produces **a hidden state for every word** in the input (h1, h2, h3, ..., hn)
2. When the decoder wants to generate an output word, it:
    - Looks at all encoder hidden states

- Calculates a **score** for each hidden state → meaning "How important is this input word for producing this output word?"

3. These scores are passed through **softmax**→ gives you **attention weights** (importance percentages)
4. Weighted hidden states are combined to form a **context vector for that timestep**
5. Decoder uses this context to produce the next output word

Attention was developed to overcome the computational bottleneck associated with context.

Instead of passing a single hidden state vector (the final one) from the encoder to the decoder, with attention we pass the full sequence of hidden states to the decoder. Each of these hidden states is associated with a single step in the input sequence, although the decoder might need the context from multiple steps in the input to inform its behavior at any given step during decoding. To achieve this, for each step in the sequence the decoder calculates a score for each of the hidden states from the encoder. Each encoder hidden state is multiplied by the softmax of its score.

Transfer Learning:

Transfer learning means:

Use a model that is already trained on huge amounts of data, and then fine-tune it for your smaller, specific task.

Important Transfer Learning Models in NLP

1. ULMFiT (Universal Language Model Fine-Tuning)

ULMFiT was **one of the first successful transfer learning techniques in NLP**.

Key Idea:

- First train a language model on a large general text (pretraining)
- Then fine-tune it on your specific NLP task (sentiment analysis, classification, etc.)

ULMFiT basically showed:

"You can reuse knowledge learned from one text task for another."

2. ELMo (Embeddings from Language Models)

ELMo improved how word embeddings work.

Traditional word embeddings:

- One fixed vector for each word
- SAME vector even if meaning changes with context (e.g., "bank" in *river bank* vs *bank account*)

What ELMo does:

ELMo gives **contextual word embeddings**.

That means:

The meaning of the word changes depending on the sentence.

3. BERT (Bidirectional Encoder Representations from Transformers)

Why is BERT special?

- Reads text **in both directions** (left → right and right → left)
- Learns deep context
- Pretrained on huge datasets
- Can be fine-tuned for many tasks: question answering, text classification, translation, summarization.

Why BERT is important?

It set **new state-of-the-art results** in many NLP tasks.

It made transfer learning in NLP extremely effective and accessible.