

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- LO 1 Understand and use the Dictionary() class of the gensim.corpora.dictionary module
- LO 2 Understand the Word2Vec class of gensim.models module
- LO 3 Grasp the concept of 'keyed vectors'
- LO 4 Use the Word2Vec class of Gensim
- LO 5 Study the source code of certain files in Gensim module
- LO 6 Study the FastText algorithm and its implementation in Gensim
- LO 7 Understand and use the LDA model
- LO 8 Understand and use the TfIdfModel in Gensim
- LO 9 Use API (provided in Gensim to download and use) pre-trained NLP models and datasets

25.1 INTRODUCTION

The Gensim is a vast library which implements many so called "Vector space models" (VSM) of NLP. This chapter discusses some of the important algorithms implementing the VSM of NLP and their implementation in Gensim.

25.2 INTRODUCTION TO THE GENSIM LIBRARY

This section discusses some basic concepts of the Gensim library.

25.2.1 Introduction to Corpus, Vector Space Model and Related Terms

As per Gensim documentation¹, some important concepts in Gensim documentation are:

- **Corpus:** A corpus is simply a collection of digital documents. During training of a “model”, you can use a corpus. Further this training is unsupervised.
- **Vector Space Model for Documents (VSM):** VSM has been explained in an appendix. In Gensim, a VSM for a document is the “features” it contains. So the number of “features” that a document has are the number of “dimensions” of the vector space for the document.
- **Sparse vector representation:** The concept of vector has already been explained. A concept which needs explanation is representation of a “sparse vector”. So there are two questions, i.e., What is a sparse vector? And how to represent it? These questions are best answered by an example. Suppose a document has 10 features (with feature ID 0 to 9) and is represented by a vector $\vec{A} = (0, 0, 0, 0, .3, .4, 0, 0, 0, .3)^T$. This is a vector of 10 dimensions but only 3 of the 10 dimensions have a non-zero entry. (Note the superscript T indicates that in normal algebra a vector is represented as a “column”, however to save space, it is written in a row with a superscript of T to indicate a “transverse” of the row). The other 7 are 0. So the vector is “sparsely populated”. So you may represent the vector as ((4, 0.3), (5, 0.4), (9, 0.3)). So you have the index of the non-zero feature and the value of the non-zero feature.
- **Model:** A model can be thought of as a transformation from one vector space to another. For example, if you convert a dictionary of say 10,000 words to a vector space with say 300 features, then the exact method by which you do this is your model.

25.2.2 The Layout of the Gensim Library

The entire Gensim library source code is available on your local machine and you can have a look at it. If you don’t know the path to the Gensim library, then on jupyter notebook you can always give the command `?gensim` and the path to the `__init__.py` will be given as follows:

1	<code>import gensim</code>
2	<code>?gensim</code>
# OUTPUT Type: module String form: <module 'gensim' from 'C:\\Users\\AG\\Anaconda3\\lib\\site-packages\\gensim__init__.py'> File: c:\\users\\ag\\anaconda3\\lib\\site-packages\\gensim__init__.py Docstring: This package contains interfaces and functionality to compute pair-wise document similarities within a corpus of documents.	

¹Detailed documentation on the library is available here: <https://media.readthedocs.org/pdf/gensim/stable/gensim.pdf>

An overview of the packages in Gensim is given in Table 25.1².

TABLE³ 25.1 Overview of the packages in Gensim

gensim	This package contains interfaces and functionality to compute pair-wise document similarities within a corpus of documents.
gensim.interfaces	This module contains basic interfaces used throughout the whole gensim package.
gensim.matutils	This module contains math helper functions.
gensim.utils	This module contains various general utility functions.
gensim.corpora	This package contains implementations of various streaming corpus IO format.
gensim.models	This package contains algorithms for extracting document representations from their raw bag-of-word counts.
gensim.parsing	This package contains functions to pre-process raw text
gensim.scripts	
gensim.similarities	This package contains implementations of pairwise similarity queries.
gensim.summarization	
gensim.topic_coherence	This package contains implementation of the individual components of the topic coherence pipeline.

25.2.3 gensim.corpora

Here only one module in the gensim.corpora, i.e., gensim.corpora.dictionary will be studied. This module has an important class Dictionary. (Note as per Python naming convention, module names should begin with small letter and the class name it contains should begin with capital letter).

You can have a look at the dictionary module source code here⁴. For those who want to avoid going into source code, you can get details of the module on Jupyter as follows:

1	<code>import gensim.corpora.dictionary</code>
2	<code>?Dictionary</code>
3	Type: Dictionary
4	String form: Dictionary(12 unique tokens: ['computer', 'human',
5	'interface', 'response', 'survey']...)
6	Length: 12
7	File: c:\programdata\anaconda3\lib\site-
8	packages\gensim\corpora\dictionary.py
9	Docstring:
10	Dictionary encapsulates the mapping between normalized words and their
11	integer ids.
12	
13	Notable instance attributes:
14	
15	Attributes

²See: https://tedboy.github.io/nlps/api_gensim.html

³Copied from: https://tedboy.github.io/nlps/api_gensim.html

⁴See: https://tedboy.github.io/nlps/_modules/gensim/corpora/dictionary.html#Dictionary . Note that the source code for all the modules is also available locally on your machine in the gensim folder.

```

16 -----
17 token2id : dict of (str, int)
18     token -> tokenId.
19 id2token : dict of (int, str)
20     Reverse mapping for token2id, initialized in a lazy manner to save memory
21     (not created until needed).
22 dfs : dict of (int, int)
23     Document frequencies: token_id -> how many documents contain this token.
24 num_docs : int
25     Number of documents processed.
26 num_pos : int
27     Total number of corpus positions (number of processed words).
28 num_nnz : int
29     Total number of non-zeroes in the BOW matrix (sum of the number of unique
30     words per document over the entire corpus).
31 Init docstring:
32 Parameters
33 -----
34 documents : iterable of iterable of str, optional
35     Documents to be used to initialize the mapping and collect corpus
36     statistics.
37 prune_at : int, optional
38     Dictionary will keep no more than `prune_at` words in its mapping, to
39     limit its RAM footprint.
40
41 Examples
42 -----
43 >>> from gensim.corpora import Dictionary
44 >>>
45 >>> texts = [['human', 'interface', 'computer']]
46 >>> dct = Dictionary(texts) # initialize a Dictionary
47 >>> dct.add_documents([["cat", "say", "meow"], ["dog"]]) # add more document
48 (extend the vocabulary)
49 >>> dct.doc2bow(["dog", "computer", "non_existent_word"])
50 [(0, 1), (6, 1)]

```

Lines 9-11: This is the docstring of the class Dictionary. The docstring tells that the Dictionary class is a “mapping” between the words and their integer ids.

Lines 15-30: These lines tell that the class has a few attributes like token2id, id2token, dfs, num_docs, num_pos, num_nnz.

Lines 31-39: These lines tell that the `__init__()` method of the Dictionary class takes two parameters, i.e., documents and prune_at. You can also see the signature of the `__init__()` method in the dictionary.py file of the corpora module.

25.2.4 The Sample/Toy Data Set of Gensim Library

The Gensim library has a small sample data set (also called toy data set). This data set has been taken from the research paper available at this⁵ link. This data set consists of “titles” of nine technical documents. Out of these 9 documents, five (c1-c5) are about Human-Computer interaction and four (m1-m4) are about graphs. Table 25.2 shows the Title Number and Title name of these five documents:

TABLE 25.2 titles of 9 technical documents

S.No.	Title Number	Title names of nine technical memoranda
1	c1	Human machine interface for Lab ABC computer applications
2	c2	A survey of user opinion of computer system response time
3	c3	The FPS user interface management system
4	c4	System and human system engineering testing of EPS
5	c5	Relation of user-perceived response time to error measurement
6	m1	The generation of random, binary, unordered trees
7	m2	The intersection graph of paths in trees
8	m3	Graph minors IV: Widths of trees and well-quasi-ordering
9	m4	Graph minors: A survey

Further a “data set” of words which have an occurrence of at least 2 or more was created from the “titles” c1-c5 and m1-m4. A total of 12 unique words (with occurrence of at least 2 or more) were found in these 9 titles. Table 25.3 shows a matrix of occurrence of these 12 unique words in the titles of these 9 documents:

TABLE 25.3 distribution of the data set of 12 unique words over 9 documents

S.No.	Terms	Documents								
		c1	c2	c3	c4	c5	m1	m2	m3	m4
1	human	1	0	0	1	0	0	0	0	0
2	interface	1	0	1	0	0	0	0	0	0
3	computer	1	1	0	0	0	0	0	0	0
4	user	0	1	1	0	1	0	0	0	0
5	system	0	1	1	2	0	0	0	0	0
6	response	0	1	0	0	1	0	0	0	0
7	time	0	1	0	0	1	0	0	0	0
8	EPS	0	0	1	1	0	0	0	0	0
9	survey	0	1	0	0	0	0	0	0	1
10	trees	0	0	0	0	0	1	1	1	0
11	graph	0	0	0	0	0	0	1	1	1
12	minors	0	0	0	0	0	0	0	1	1

⁵Deerwester et al. (1990): Indexing by Latent Semantic Analysis available at: http://www.cs.bham.ac.uk/~pxt/IDA/lsa_ind.pdf

The above shown sample data is available in Gensim library. Further, this toy data set has been converted into a “toy corpora” and a “toy dictionary”. All these, namely the “toy data”, the “toy corpora” and the “toy dictionary”, are available in Gensim as `common_texts`, `common_corpus`, `common_dictionary`. Having a look at this sample, data base will help you to understand how the three items namely documents, corpora and dictionary are organized.

- `common_texts`: The text is stored as a list of lists. The outer list represents the entire collection of documents. Each inner list represents one document. Each word/term in a particular document is a member of the inner list in form of a string. Note that common words and stop words are removed. In the sample data of Gensim, there is a list of lists called `common_text` and this `common_text` variable is a list of nine lists and each of these nine lists represents one document.
- `common_dictionary`: This sample data also has a dictionary made from these nine documents. While making the dictionary, the common words/stop words are removed. In Gensim, this sample dictionary is called `common_dictionary` and in the case of the toy data, this toy dictionary or `common_dictionary` has 12 terms/words.
- `common_corpus`: This sample data also has a corpus. Now each word in the dictionary has been assigned an integer value. Since the `common_dictionary` has 12 terms, so the index of words in the dictionary vary from 0 to 11. The `common_corpus` is represented as a list of lists of tuples. The outer list represents the entire corpus. The inner list represents each document in the corpus. The tuple inside the inner list has two numbers. The first number is the index of a particular word in the dictionary and the second number is its frequency in that particular document. So the corpus consists of the integer for the word (as assigned in the dictionary) and the frequency of the word in the document.

You may have a look at this sample data⁶. This data can be accessed as follows:

```

1 from gensim.test.utils import common_texts, common_corpus, common_dictionary
2 # common_texts is a list of lists of strings
3 print('Numb of docs in toy data set->', len(common_texts))
4 print('common_texts->', common_texts)
5 # common_dictionary is a list of only unique words in the common_text
6 print('Length of common dictionary', len(common_dictionary))
7 print('common_dictionary->', common_dictionary)
8 # common_corpus is a list of list of tuples
9 # Each inner list is a document.
10 # Each tuple in inner list represents the index of each word along with its
11 frequency
12 print('Common corpus->', common_corpus)

13 # Output
14 Numb of docs in toy data set-> 9
15 common_texts-> [['human', 'interface', 'computer'], ['survey', 'user',
16 'computer', 'system', 'response', 'time'], ['eps', 'user', 'interface',
17 'system'], ['system', 'human', 'system', 'eps'], ['user', 'response',
18 'time'], ['trees'], ['graph', 'trees'], ['graph', 'minors', 'trees'],
19 ['graph', 'minors', 'survey']]
20 Length of common dictionary 12
21 common_dictionary-> Dictionary(12 unique tokens: ['computer', 'human',
22 'interface', 'response', 'survey']...)
```

⁶This exercise is largely based on the example given at: <https://radimrehurek.com/gensim/tut1.html>

```

23 Common corpus-> [[(0, 1), (1, 1), (2, 1)], [(0, 1), (3, 1), (4, 1), (5, 1),
24 (6, 1), (7, 1)], [(2, 1), (5, 1), (7, 1), (8, 1)], [(1, 1), (5, 2), (8, 1)],
25 [(3, 1), (6, 1), (7, 1)], [(9, 1)], [(9, 1), (10, 1)], [(9, 1), (10, 1), (11,
26 1)], [(4, 1), (10, 1), (11, 1)]]

```

25.2.5 Creating a Dictionary Object

This section deals with creating a Dictionary object (from corpora module of Gensim) and using it to create a bag of words (bow).

Note that the Dictionary consists of all the unique words. For this exercise, a “list of lists” is used. The outer list represents the entire collection, while each inner list represents a document. So the collection is as follows:

```
[["All", "is", "well"], ["Be", "the", "change"], ["Do", "Your", "best"], ["Never", "give", "up"]]
```

The code is as follows:

```

1 from gensim import corpora
2 my_text = [["all", "is", "well"], #Collection of 4 documents
3           ["be", "the", "change"], # 3 words in each document
4           ["do", "your", "best"],
5           ["never", "give", "up"]]
6 my_dict = corpora.Dictionary(my_text)
7 print('my_dict->', my_dict)
8 print('Token to id->', my_dict.token2id)
9 print('id to token is empty->', my_dict.id2token)
10 print('Number of docs->', my_dict.num_docs)
11 print('number of processed words->', my_dict.num_pos)
12 print('No of non-zero->', my_dict.num_nnz)
13 print('Token at id 8->', my_dict[8])
14 print('Now id to token->', my_dict.id2token)
15 #Save the Dictionary object ie serialize it
16 my_path = r"C:\Python34\myScripts\my_saved_dict.dict"
17 my_dict.save(my_path)
18 # Output
19 my_dict-> Dictionary(12 unique tokens: ['all', 'is', 'well', 'be',
20 'change']...)
21 Token to id-> {'all': 0, 'is': 1, 'well': 2, 'be': 3, 'change': 4, 'the': 5,
22 'best': 6, 'do': 7, 'your': 8, 'give': 9, 'never': 10, 'up': 11}
23 id to token is empty-> {}
24 Number of docs->4
25 number of processed words->12
26 No of non-zero->12
27 Token at id 8-> your
28 Now id to token-> {0: 'all', 1: 'is', 2: 'well', 3: 'be', 4: 'change', 5:
29 'the', 6: 'best', 7: 'do', 8: 'your', 9: 'give', 10: 'never', 11: 'up'}

```

Explanation:

Lines 2-5: This is a collection of four documents. Each document is a list of strings. The whole collection is a list of lists. Note that there are 3 unique tokens and 4 lists and none of the words are repeated. So your dictionary will have 12 tokens.

Line 6: Use the Dictionary class of the corpora module to create a Dictionary object.

Lines 7-14: These are the various attributes of the Dictionary object. One important point to note is Lines 9 and 14. In both these lines, the `id2token` attribute is used. However, the output by Line 9 is an empty dictionary (as shown by output in Line 20). But the output of the same attribute from Line 14 is not an empty dictionary. Why? This is because the `id2token` attribute gets “populated” only when you use the Dictionary object (which you did in Line 13).

Line 17: The Dictionary object has a `save()` method to serialize the Dictionary object to hard disc.

25.2.6 Converting a Document into a Bag of Words (using a Dictionary)

In the previous example, you created a “toy dictionary” of 12 tokens and they had id from 0 to 11. Now if you take a new document, you can create a “bag of words” representation of this new document based on the dictionary. These tokens along with their id are:

1	{0: 'all', 1: 'is', 2: 'well', 3: 'be', 4: 'change', 5: 'the', 6: 'best', 7:
2	'do', 8: 'your', 9: 'give', 10: 'never', 11: 'up'}

Suppose you have a new document consisting of some tokens, then you can use the dictionary created earlier to convert the document into a bag of words (BoW). Following things can be said about the BoW representation of a document in Gensim:

- The bag of words is a list of tuples.
- Each tuple consists of two integers. The first integer is the index of that particular word in the dictionary.
- The second integer is the frequency (number of counts) of that word in the document.
- If a word in the document is not present in the dictionary, then that word is not there in the BoW.

Consider the following code (which uses the dictionary created in the previous example to convert a document into a BoW):

1	# new doc has 3 instances of word 'all'
2	# new doc has 2 instances of word 'never'
3	# new doc has some word which are not in the dictionary
4	new_doc = 'all all all never never non_exist1 non_exist1 non_exist2'
5	doc_vec = my_dict.doc2bow(new_doc.lower().split())
6	print(doc_vec)
7	# OUTPUT
8	[(0, 3), (10, 2)]

Can you add the non-existent words to the dictionary? yes

Suppose the document has a word/token which does not exist in the dictionary. Gensim provides a way of adding them to the dictionary. To do so, first have a look at the source code of the `doc2bow()` method⁷. It is as follows:

```

1 def doc2bow(self, document, allow_update=False, return_missing=False):
2     """Convert `document` into the bag-of-words (BoW) format = list of
3     `(token_id, token_count)` tuples.
4     Parameters
5     -----
6     document : list of str
7         Input document.
8     allow_update : bool, optional
9         Update self, by adding new tokens from `document` and updating
10    internal corpus statistics.
11    return_missing : bool, optional
12        Return missing tokens (tokens present in `document` but not in self)
13    with frequencies?
14    Return
15    -----
16    list of (int, int)
17        BoW representation of `document`.
18    list of (int, int), dict of (str, int)
19        If `return_missing` is True, return BoW representation of `document`
20    + dictionary with missing
21    tokens and their frequencies.

```

The method takes three parameters. If you set `allow_update` to `True`, then all words in the document, which are not in the dictionary will be added to the dictionary. Further, if you set the third parameter, i.e., `return_missing` to `True`, the method will return 2 objects, i.e., a list and a dictionary. The list is your normal BoW representation of the document, but the dictionary is the index of the new words added to the dictionary along with their frequency. This will become clear from the following modification to code above.

```

1 doc_with_missing, only_missing = my_dict.doc2bow(new_doc.lower().split(),
2                                                  allow_update = True,
3                                                  return_missing = True)
4 print('doc_with_missing->', doc_with_missing)
5 print('only_missing->', only_missing)
6 # Output
7 doc_with_missing-> [(0, 3), (10, 2), (12, 2), (13, 1)]
8 only_missing-> {'non_exist1': 2, 'non_exist2': 1}

```

Notice that the two words which were present in the document but not in the dictionary have been added to the dictionary. Further note that these two previously non-existing words have been returned

⁷See Lines 213 to 231 of source code at:

<https://github.com/RaRe-Technologies/gensim/blob/develop/gensim/corpora/dictionary.py>

by the method in the form of a Python dictionary object giving the non-existent words as keys and their frequency as values.

Note that the Dictionary object of corpora module of Gensim, has a number of other methods⁸.

25.2.7 Creating Dictionary from a Number of Text Files on Hard Disk

In this exercise, the steps are as follows:

- Create a Dictionary object, using a number of files in a folder. You may also create a function named `iter_all_docs()` to iterate over each document in a given folder and to pick only those documents which have extension `.txt`. Note that `filter` and `yield`⁹ have been used in the function.
- Further you will use the `tokenize()` method of the `utils` module of Gensim to tokenize the words.
- You will also “trim” the Dictionary object, by using the `filter_extremes()` method, whose signature from Gensim documentation¹⁰ is:

```

1 def filter_extremes(self, no_below=5, no_above=0.5, keep_n=100000,
2   keep_tokens=None):
3     """Filter out tokens in the dictionary by their frequency.
4       Parameters
5       -----
6       no_below : int, optional
7           Keep tokens which are contained in at least `no_below` documents.
8       no_above : float, optional
9           Keep tokens which are contained in no more than `no_above` documents
10          (fraction of total corpus size, not an absolute number).
11       keep_n : int, optional
12           Keep only the first `keep_n` most frequent tokens.
13       keep_tokens : iterable of str
14           Iterable of tokens that **must** stay in dictionary after filtering.

```

The code is as follows:

```

1 import os, gensim
2 # Function to iterate over all files in a directory
3 def iter_all_docs(parent_dir):
4     for (dir_path, dir_name, all_files) in os.walk(parent_dir):
5         for file in filter(lambda file: file.endswith('.txt'), all_files):
6             document = open(os.path.join(dir_path, file)).read()
7             yield gensim.utils.tokenize(document, lower=True)
8
9 parent_dir = r"C:\NLTK_DATA\corpora\gutenberg"
10 my_dict = gensim.corpora.Dictionary(iter_all_docs(parent_dir))
11 my_dict.filter_extremes(no_below=1, keep_n=30000)
12 print(my_dict)

```

⁸For details of various attributes and methods of the Dictionary object, see:

<https://www.pydoc.io/pypi/gensim-3.2.0/autoapi/corpora/dictionary/index.html>

⁹A detailed discussion on `lambda` and `filter` is available here: <https://docs.python.org/3/howto/functional.html>

¹⁰See Lines 314 to 327 of: <https://github.com/RaRe-Technologies/gensim/blob/develop/gensim/corpora/dictionary.py>

```

13 #Keep only 10000 most frequent words
14 my_dict.filter_n_most_frequent(10000)
15 print(my_dict)
16 #Remove token '_a_'
17 my_dict.filter_tokens(bad_ids=[my_dict.token2id['_a_']])
18 print(my_dict)
19 #Save the Dictionary
20 my_path = r"C:\Python34\myScripts\my_guten_dict.dict"
21 my_dict.save(my_path)

22 # Output
23 Dictionary(30000 unique tokens: ['_', '_____', '_a_', '_accepted_',
24 '_adair_',...])
25 Dictionary(20000 unique tokens: ['_____', '_a_', '_accepted_', '_adair_',
26 '_addition_',...])
27 Dictionary(19999 unique tokens: ['_____', '_accepted_', '_adair_',
28 '_addition_', '_all_',...])

```

Line 14: Here a method `filter_n_most_frequent()` method is used. The parameter to this method is how many of the most frequent tokens will be removed. Here the parameter is 10000, so that many “most frequent” tokens will be removed.

Looking at the output of the code above you can see that you removed 1 bad id, i.e., `_a_`. But suppose you wanted to remove all such words which are not alphabet or numeric, i.e., you want to keep only words and numbers. How can you do this? This can be done by a regular expression as shown in the code snippet below:

```

1 # following reg exp will keep only alphabet and numeric
2 doc_clean = re.sub(r'[\W_]+', ' ', document)

```

Further suppose you want to remove all stop words from the dictionary. How can you do this? One way of doing this could be to take stop words from the NLTK library and use them to remove the stop words from the dictionary. The steps are as follows:

- Get the English stop words from `nltk.corpus.stopwords.word('english')`
- For each word in the list of stopwords, see if that particular word is in the dictionary and if it is, then get its id and add that id to the list of ids of stop words.

The code snippet for removing stop words from the dictionary is as follows:

```

1 import nltk
2 # Get ids of stop words
3 stop_words = nltk.corpus.stopwords.words('english')
4 stop_id = [my_dict.token2id[word] for word in stop_words
5           if word in my_dict.token2id]
6 # Now remove these ids of stop words from dictionary
7 my_dict.filter_tokens(stop_id)

```

Now you may combine the above two code snippets to the code for creating your dictionary. Note that you must remove the following two lines from the code above:

```

1 #Remove token '_a_'
2 my_dict.filter_tokens(bad_ids=[my_dict.token2id['_a_']])

```

These two lines should be removed because after introducing the regular expression, `_a_` would be automatically removed and so this token is no more present in the dictionary and if you try to get the id related to `_a_`, you will get an error. The final code will be as follows (a print statement has also been included to give first 20 characters of each file being read into the dictionary, just to tell to the reader what is happening inside the `iter_all_docs()` function).

```

1  import os, re
2  import gensim
3  import nltk
4
5  # Function to iterate over all files in a directory
6  def iter_all_docs(parent_dir):
7      for (dir_path, dir_name, all_files) in os.walk(parent_dir):
8          for file in filter(lambda file: file.endswith('.txt'), all_files):
9              document = open(os.path.join(dir_path, file)).read()
10             # Remove all that is not an alphabet or a numeral
11             doc_clean = re.sub(r'[\W_]+', ' ', document)
12             # Show first 20 characters of each document
13             print(doc_clean[:20], '#', end = '')
14             yield gensim.utils.tokenize(doc_clean, lower=True)
15
16  parent_dir = r"C:\NLTK_DATA\corpora\gutenberg"
17  my_dict = gensim.corpora.Dictionary(iter_all_docs(parent_dir))
18  my_dict.filter_extremes(no_below=1, keep_n=30000)
19  print(my_dict)
20  #Keep only 10000 most frequent words
21  my_dict.filter_n_most_frequent(10000)
22  print(my_dict)
23  # Get ids of stop words
24  stop_words = nltk.corpus.stopwords.words('english')
25  stop_id = [my_dict.token2id[word] for word in stop_words
26             if word in my_dict.token2id]
27  # Now remove these ids of stop words from dictionary
28  my_dict.filter_tokens(stop_id)
29  print(my_dict)
30  #Save the Dictionary
31  my_path = r"C:\Python34\myScripts\my_guten_dict.dict"
32  my_dict.save(my_path)
33  # Output
34  Emma by Jane Austen # Persuasion by Jane # Sense and Sensibili # The King
35  James Bibl # Poems by William Bl # Stories to Tell to # The Adventures of B
36  # Alice s Adventures # The Ball and The Cr # The Wisdom of Fathe # The Man
37  Who Was Thu # The Parent s Assist # Moby Dick by Herman # Paradise Lost by Jo
38  # The Tragedie of Jul # The Tragedie of Ham # The Tragedie of Mac # Leaves of
39  Grass by #Dictionary(30000 unique tokens: ['abbey', 'abbots', 'abdy',
40  'abhor', 'abhorred']...)
41  Dictionary(20000 unique tokens: ['abbots', 'abdy', 'abolition', 'absences',
42  'absented']...)
43  Dictionary(20000 unique tokens: ['abbots', 'abdy', 'abolition', 'absences',
44  'absented']...)

```

25.3 UNDERSTANDING AND USING THE WORD2VEC CLASS

There is an important class in Gensim, called Word2Vec. This section discusses this class and how to use this class to build models.

25.3.1 Word2Vec class of gensim.models module

The concepts relating to vector space model have been explained in the previous chapter.

You can get details of the Word2Vec class in the gensim.models module.

The signature of the Word2Vec is as follows:

```
1 Init signature: Word2Vec(sentences=None, corpus_file=None, size=100,
2 alpha=0.025, window=5, min_count=5, max_vocab_size=None, sample=0.001,
3 seed=1, workers=3, min_alpha=0.0001, sg=0, hs=0, negative=5,
4 ns_exponent=0.75, cbow_mean=1, hashfxn=<built-in function hash>, iter=5,
5 null_word=0, trim_rule=None, sorted_vocab=1, batch_words=10000,
6 compute_loss=False, callbacks=(), max_final_vocab=None)
```

You can see that the constructor to this class takes a very large number of parameters. However, most parameters take some “default” values and can be ignored. The four important ones which you will need are as follows¹¹:

	Parameters:
1	sentences : For this parameter, you may provide a list of lists.
2	size : This parameter is optional and of type “int” with default value of 100. For this you may provide the “ <i>Dimensionality of vectors</i> ”.
3	window : This parameter is optional and of type “int” with default value of 5. It represents the Maximum distance between the current and predicted word within a sentence.
4	min_count : This parameter is optional and of type “int” with default value of 5. It ignores all words with total frequency lower than this. sg : This parameter is optional and may have values 0 or 1. Default is 0. This parameter specifies the training algorithm: 1 for skip-gram; otherwise CBOW.

Another thing that you need to know about the Word2Vec class is that it has three important attributes namely: (1) **wv**, (2) **vocabulary**, and (3) **trainables**. Out of these three, only two are relevant for the present discussion namely: (1) **wv**, and (2) **vocabulary**. Note that both these attributes represent objects. The following table describes them:

	Attributes
1	wv : The parameter wv gives you an object of type ‘Word2VecKeyedVectors’. This object essentially contains the “mapping” between words and “embeddings”. (The concept will become clear once you use this later in the chapter)

¹¹ Note that some “parameters” may take “more than one type of data” or “data in different formats”. However, here only the common data used in this exercise are given. So if you want to see all the options for a parameter, check out the source code file.

- 2 vocabulary: The parameter vocabulary gives you an object of type 'Word2VecVocab'. This object represents the vocabulary (sometimes called Dictionary in gensim) of the model.

25.3.2 Using Word2Vec

This exercise demonstrates the use of Word2vec. For this example, you will use the text data available here¹². Download the data and save it as a text file. (You can copy it and save it using a text editor like notepad++, giving extension .txt.). Screenshot of the file is shown in Figure 25.1.



FIGURE 25.1 Screen shot of <http://download.tensorflow.org/data/questions-words.txt> from where the txt data is to be downloaded for this exercise

After downloading the above text file, you will do the following things in writing the script:

- Create a Class named File2Sents. When you create an object of this class, you will pass it the path to the parent directory in which the text file(s) are present. You must implement an `__iter__()` method in this class to make the objects of this class iterable. The advantage of using an iterable is that for very large corpora you can save RAM. (Using an iterator for loading files is a common coding practice while using Gensim or other language modelling libraries.)
- Create a list of lists of strings by iterating over each sentence of the File2Sent object created in the above step. After this you will have a list of lists. Each inner list represents a document and each string in the inner list represents a word.
- Create a Word2Vec model using class Word2Vec constructor and passing it as parameter the list of lists created earlier. Some other parameters need to be given also. In the example code below, you have given five parameters to the constructors which are: `my_model = Word2Vec(all_sents, size =`

¹²<http://download.tensorflow.org/data/questions-words.txt>

10, window = 5, min_count= 0, workers =4). Here size is dimension of the vector. A typical vector could have a dimension of around 100, but here it has been kept to 10.

- Use the 'wv' attribute of the Word2Vec object to create an object of type Word2VecKeyedVectors.

Explanation:

- An object of type Word2Vec has an "attribute" 'wv.
- So you can access this attribute using Word2Vec.wv.
- Now this attribute in turn automatically creates an object of type Word2VecKeyedVectors.
- The next questions are what is a Word2VecKeyedVectors¹³? And why do you need it?
- As per Gensim source code documentation "This object essentially contains the mapping between "words" and their "embeddings".
- The "words" are strings and the "vector representation" is a numpy array. So Word2VecKeyedVectors is a "mapping" between "words, i.e., strings" and their "vector representations, i.e., a numpy array".
- After training, it can be used directly to query those embeddings in various ways¹⁴." Shortly the source code will also be examined.

- After having created an object of type Word2VecKeyedVectors, you will access one of its many methods namely word_vec()¹⁵. This method gives the vector representation of the word given to the method as parameter. Since you gave the size =10 in the construction of Word2Vec object, this method will give as output a 10 dimension vector of the input word.

Why do you need to have Word2VecKeyedVectors?

The reason you have Word2VecKeyedVectors are:

- That "trained vectors" are and should be "independent" from the "way they are trained" meaning that it should not matter what particular algorithm you used to train the vectors. Once you have trained your vectors (may be using Word2Vec or say an algorithm say FastText, you should be able to use the "embedding" regardless of how they were trained).
- So Word2VecKeyedVectors are independent from the way they were trained.
- Keyed vectors occupy less RAM so are easier to save.

- Finally you can serialize (i.e., save to disc) the Word2VecKeyedVectors by using save() method.

The code is as follows:

```

1 import os
2 from gensim.models import Word2Vec
3 from gensim.test.utils import get_tmpfile
4
5 class File2Sents(object):
6     def __init__(self, parent_dir):
7         self.parent_dir = parent_dir
8 
```

¹³For details see:- <https://radimrehurek.com/gensim/models/keyedvectors.html>

¹⁴See lines 613-615 of file word2vec.py in .\gensim\models module

¹⁵You can see the source code for this method in lines 421-452 of file keyedvectors.py in .\gensim\models

```

9     def __iter__(self):
10         for fname in os.listdir(self.parent_dir):
11             for line in open(os.path.join(self.parent_dir, fname)):
12                 yield line.split()
13
14     # sentences is an iterator
15     sentences = File2Sents(r'C:\testData') # Use your file location instead
16     #Use the iterator to make a list of lists of strings
17     all_sents = [sent for sent in sentences]
18     #Confirm that all_sents is actually a list of lists of strings
19     print(all_sents)
20     #Create a model
21     my_model = Word2Vec(sentences = all_sents, size = 10, window = 5, min_count=
22     0, workers =4, sg = 0)
23     #Create an object of type gensim.models.keyedvectors.Word2VecKeyedVectors
24     #Use the .wv attribute of Word2Vec object
25     my_vec = my_model.wv
26     # Use the word_vec() method of Word2VecKeyedVectors object
27     print('Vector representation for king->', my_vec.word_vec('king'))
28     fname = get_tmpfile(r"C:\gensimModels\saved_vectors.kv")
29     my_vec.save(fname)
30
31     # Output for print statement in line 18
32     [[':', 'capital-common-countries'], ['Athens', 'Greece', 'Baghdad',
33     'Iraq'], ['Athens', 'Greece', 'Bangkok', 'Thailand'], ['Athens',
34     'Greece', 'Beijing', 'China'], ['Athens', 'Greece', 'Berlin',
35     'Germany'],.....#Rest of output truncated to save space
36     # Output for print statement in line 24
37     Vector representation for king-> [-0.17738968  0.48128685  0.13709027 -
38     0.00647458  0.55266404 -0.89207804
39     0.32079458 -0.0495178  0.18800315  0.23434964]

```

Lines 5-12: This is a class which implements the iterator function.

Line 15: Here you iterate over each sentence in the file and create a list of lists of strings.

Line 21: Here you create the model of type Word2Vec. The parameters to the method are:

- sentences = all_sents. This parameters contains the list of list of words (in the form of strings).
- size: the number of dimensions of the vector.
- window: the number of context words.
- min_count: tells the model to ignore words with total count less than this number.
- workers: the number of threads being used.
- sg: whether to use skip-gram or CBOW.

Line 25: Here you use the wv attribute to convert the Word2Vec model into a Word2VecKeyedVectors (also called KeyedVectors) model. There are many advantages of using a KeyedVectors model. It occupies less memory. It is also common to various models. This basically means is that you have various algorithms to create models of various types like Word2Vec (Of Google) or FastText (Of Facebook). So while Word2Vec and FastText are not compatible to each other, but a KeyedVectors model created from either of these models is similar and therefore compatible.

25.3.3 Some Common Methods of the Word2VecKeyedVectors Class

There are a number of other methods available in this class. Some common ones (copied from source code) are:

```
def most_similar(self, positive=None, negative=None, topn=10, restrict_vocab=None,
indexer=None):
    """Find the top-N most similar words. Positive words contribute positively towards
    the similarity, negative words negatively. This method computes cosine similarity
    between a simple mean of the projection weight vectors of the given words and the
    vectors for each word in the model. The method corresponds to the `word-analogy` and
    `distance` scripts in the original word2vec implementation.

    Parameters
    positive : list of str optional-> List of words that contribute positively.
    negative : list of str, optional-> List of words that contribute negatively.
    topn : int, optional-> Number of top-N similar words to return.
    restrict_vocab : int, optional-> Optional integer which limits the range of vectors
    which are searched for most-similar values. For example, restrict_vocab=10000 would
    only check the first 10000 word vectors in the vocabulary order. (This may be
    meaningful if you've sorted the vocabulary by descending frequency.)
    Returns-> list of (str, float)-> Sequence of (word, similarity)."""

def words_closer_than(self, w1, w2):
    """Get all words that are closer to `w1` than `w2` is to `w1`.

    Parameters
    w1 : str-> Input word.
    w2 : str-> Input word.
    Returns-> list (str). List of words that are closer to `w1` than `w2` is to `w1`.

def similar_by_word(self, word, topn=10, restrict_vocab=None):
    """Find the top-N most similar words.

    Parameters
    word : str-> Word
    topn : {int, False}, optional-> Number of top-N similar words to return. If topn is
    False, similar_by_word returns the vector of similarity scores.
    restrict_vocab : int, optional-> Optional integer which limits the range of vectors
    which are searched for most-similar values. For example, restrict_vocab=10000 would
    only check the first 10000 word vectors in the vocabulary order. (This may be
    meaningful if you've sorted the vocabulary by descending frequency.)
    Returns-> list of (str, float)-> Sequence of (word, similarity)."""

def similar_by_vector(self, vector, topn=10, restrict_vocab=None):
    """Find the top-N most similar words by vector.

    Parameters
    vector : numpy.array-> Vector from which similarities are to be computed.
    topn : {int, False}, optional-> Number of top-N similar words to return. If topn is
    False, similar_by_vector returns the vector of similarity scores.
    restrict_vocab : int, optional-> Optional integer which limits the range of vectors
    which are searched for most-similar values. For example, restrict_vocab=10000 would
    only check the first 10000 word vectors in the vocabulary order. (This may be
    meaningful if you've sorted the vocabulary by descending frequency.)
    Returns-> list of (str, float)-> Sequence of (word, similarity)."""
```

```

def distance(self, w1, w2):
    """Compute cosine distance between two words. Calculate 1 -
    :meth:`~gensim.models.keyedvectors.WordEmbeddingsKeyedVectors.similarity`.
    Parameters
    w1 : str-> Input word.
    w2 : str-> Input word.
    Returns-> float-> Distance between `w1` and `w2`."""

def similarity(self, w1, w2):
    """Compute cosine similarity between two words.
    Parameters
    w1 : str-> Input word.
    w2 : str-> Input word.
    Returns-> float-> Cosine similarity between `w1` and `w2`."""

```

The following script uses some of the methods detailed above to clarify the concepts:

```

1 from gensim.test.utils import get_tmpfile
2 from gensim.models import KeyedVectors
3
4 # Path to saved KeyedVectors model
5 fname = get_tmpfile(r"C:\gensimModels\saved_vectors.kv")
6 # Load the keyedvectors model
7 word_vectors = KeyedVectors.load(fname, mmap='r')
8 #Confirm that object loaded from memory is of type Word2KeyedVectors
9 print('Type->', word_vectors)
10 #Find list of 5 words similar to given word
11 word_list1 = word_vectors.most_similar(positive = ['London'], topn = 5)
12 print(word_list1)
13 #Find words closer to w1 than w2
14 word_list2 = word_vectors.words_closer_than(w1 = 'London', w2 = 'Paris')
15 print('Words closer to London than to Paris', word_list2)
16 #Find cosine distance between w1 and w2
17 word_dist = word_vectors.distance(w1 = 'London', w2 = 'Paris')
18 print('Word distance->', word_dist)
19 #Find vector representation of given word
20 word_vec = word_vectors.word_vec('London')
21 print('vector of word London->', word_vec)
22 # Use vector representation of a word to find words with similar vectors
23 word_sim2vec = word_vectors.similar_by_vector(word_vec, topn = 5)
24 print('words similar to vector of london', word_sim2vec)
30 # Output for print statement in line 18
31 Type-><gensim.models.keyedvectors.Word2VecKeyedVectors object at 0x0EB31E50>
32 [(('English', 0.9813432693481445), ('Roseau', 0.8832843899726868),
33 ('Belgrade', 0.8551099896430969), ('Baghdad', 0.8473684191703796), ('Niamey',
34 0.8416839241981506))]
35 Words closer to London than to Paris ['Baghdad', 'Tehran', 'Ankara',
36 'Belgrade', 'Niamey', 'Cyprus', 'Greenland', 'Morocco', 'Roseau', 'Yerevan',
37 'English']

```

```

38 Word distance->0.17602890729904175
39 vector of word London-> [-0.261399270.530188441.07105040.92083263 -0.3914581
40 -1.6739217
41 2.12282320.92644211.1550874 -0.3501836 ]
42 words similar to vector of london [('London', 1.0), ('English',
43 0.9813432693481445), ('Roseau', 0.8832843899726868), ('Belgrade',
44 0.8551099896430969), ('Baghdad', 0.8473684191703796)]

```

25.4 USING THE LINESENTENCE() CLASS OF THE WORD2VEC CLASS TO ITERATE OVER A FILE

It has been mentioned many times that the various classes and methods in the Gensim library, generally require text in the form of list of lists of strings/tokens. In earlier scripts, you had written code which could create such a list of lists from a file on the hard disk.

But the word2vec.py module of Gensim library also provides an inbuilt class called LineSentence which can iterate over a file containing sentences. This class takes as one of its parameters source = "Path_to_file_on_disk".

The relevant code of the source file word2vec.py is shown in Figure 25.2.

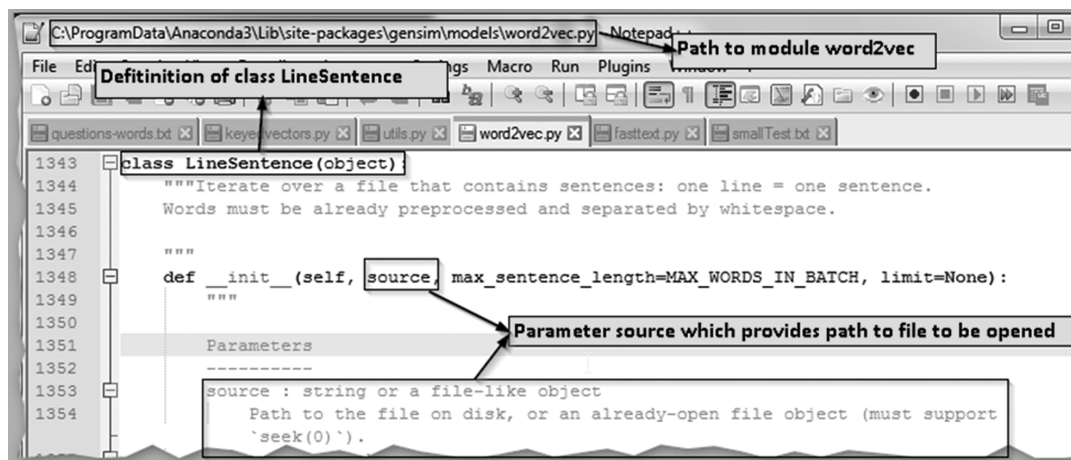


FIGURE 25.2 Screenshot of relevant portions of source code of class LineSentence available in file word2vec.py

You will use this class object to iterate over a file when creating the next model.

25.5 SOME OTHER MODELS IN GENSIM

In the previous section the Word2Vec class and related model was discussed. Gensim has a few other models like FastText, LDA model and Tfidf model. They are discussed in this section.

25.5.1 FastText

The FastText algorithm is by Facebook and is actually a modification/extension of the Word2Vec algorithm. The major difference between the two algorithms is that FastText breaks up a word into “components” and then treats each of these components as separate words or “grams”. For example, trigrams from a word say “pilot” could be <pil, ilo, lot>. So the word embedding for word “pilot” will be the sum of all these n-grams.

So a Word2Vec algorithm treats a word as the ‘smallest unit’ but FastText breaks up a word further. The advantage of this approach is that FastText is able to vectorize words based on their components and can even identify misspelt words.

You will be using the FastText class in the fasttext module. The signature of the FastText class on Jupyter is as follows:

1	<code>from gensim.models.fasttext import FastText</code>
2	<code>?FastText</code>
3	Init signature: <code>FastText(sentences=None, corpus_file=None, sg=0, hs=0,</code>
4	<code>size=100, alpha=0.025, window=5, min_count=5, max_vocab_size=None,</code>
5	<code>word_ngrams=1, sample=0.001, seed=1, workers=3, min_alpha=0.0001, negative=5,</code>
6	<code>ns_exponent=0.75, cbow_mean=1, hashfxn=<built-in function hash>, iter=5,</code>
7	<code>null_word=0, min_n=3, max_n=6, sorted_vocab=1, bucket=2000000,</code>
8	<code>trim_rule=None, batch_words=10000, callbacks=(), compatible_hash=True)</code>

As usual, the FastClass() class has a large number of parameters, out of which most have default values and can be ignored. Some important ones are as follows:

	Parameters
1	<code>sentences</code> : This is an optional parameter and can take a “list of lists of tokens, i.e., strings”. For very large corpora, you may like to give it an “iterable” that streams sentences from disk/ network. Note if you don’t provide any value for this parameter then the model is left “uninitialized”.
2	<code>corpus_file</code> : This is an optional parameter. It provides “path to the corpus file” used in the <code>LineSentence()</code> method described earlier. Note that either you provide the “sentences” parameter or the “corpus_file” parameter but not both.
3	<code>min_count</code> : This is an optional parameter of type integer ie “int”.Its default value is 5. The model ignores all words with total frequency lower than this.
4	<code>size</code> : This parameter is optional and of type integer ie “int” with default value of 100. This parameter sets the “Dimensionality of the word vectors”. <code>window</code> : This parameter is optional and of type integer ie “int” with default value of 5. It indicates the maximum distance between the current and predicted word within a sentence.
5	<code>window</code> : This parameter is optional and of type integer ie “int” and has a default value of 5. It indicates the maximum distance between the current and predicted word within a sentence.

6	<code>sg</code> : This parameter (<code>sg</code> stands for skip-gram) is optional and can have value of 0 or 1 with default value of 0. If it is 1, then the training algorithm is skip-gram. If it is 0 then the training algorithm is CBOW.
7	<code>max_vocab_size</code> : This parameter is optional and of type integer ie "int" with a default value of None. What it does it that it sets the limit to number of "unique words". If the number of unique words is greater than this number then the "infrequent words" are removed. If you don't give any value to this parameter then it will have a default value of None meaning that there is "no limit" to the number of unique words.
8	<code>iter</code> : This parameter is optional and of type integer. It has a default value of 5. It indicates the number of iterations (epochs) over the corpus. Keep it small if your data is very large
9	<code>min_n</code> : This parameter is optional an of type integer, i.e., "int". It has default value of 3. It indicates the minimum length of char n-grams to be used for training word representations.
10	<code>max_n</code> : This parameter is optional and of type integer ie "int". It has a default value of 6. It indicates the max length of char ngrams to be used for training word representations. Set <code>`max_n`</code> to be lesser than <code>`min_n`</code> to avoid char ngrams being used. Note that <code>min_n</code> and <code>max_n</code> together indicate the subwords ie the substrings contained in a word between the minimum size (<code>min_n</code>) and the maximal size (<code>max_n</code>). By default, it take all the subword between 3 and 6 characters.
11	<code>word_ngrams</code> : This parameter is optional and can value of 0 or 1. Default value is 1. • If 1, uses enriches word vectors with subword(n-grams) information. • If 0, this is equivalent to <code>:class:`~gensim.models.word2vec.Word2Vec`</code>.

The `Fasttext` class also has a number of attributes which are very useful in working with it. The following table gives summary of some important attributes of `fasttext` class (as implemented in Gensim).

Attributes	
1	<code>wv</code> : This attribute is an object of class <code>FastTextKeyedVectors</code> . This object essentially contains the mapping between words and embeddings. These are similar to the embeddings computed in the class <code>Word2Vec</code> . The difference is that here are also include vectors for n-grams. This allows the model to compute embeddings even for <code>**unseen**</code> words (that do not exist in the vocabulary), as the aggregate of the n-grams included in the word. After training the model, this attribute can be used directly to query those embeddings in various ways.
2	<code>vocabulary</code> : This attribute is an object of class <code>FastTextVocab</code> . This object represents the vocabulary of the model. Besides keeping track of all unique words, this object provides extra functionality, such as discarding extremely rare words.
3	<code>Trainables</code> : This attribute is not discussed. You may look up the online documentation for details

You may now use the `FastText` class of the `fasttext` module to create a model. The steps are as follows:

- Use the `LineSentence()` method of the `Word2Vec` class to create an iterable object over the file. This iterable object then can be used to create a `Word2Vec` model and also a `FastText` model.
- Create a `Word2Vec` model with dimension of 10, a context window of 5.
- Create a `Word2Vec` model
- Create a `FastText` model
- Train the `FastText` model
- Test the training data. You know that the training data contains the word “Hanoi” and “Vietnam”, but it does not contain “HanoiViet”. So a `Word2Vec` model won't be able to find a vector corresponding to “HanoiViet”, but a `FastText` model will get it.

```

1 from gensim.models.word2vec import Word2Vec, LineSentence
2 from gensim.models.fasttext import FastText
3 # -----Using LineSentence to iterate over file having sentences-----
4 path_to_file = r"C:\testData\questions-words.txt"
5 my_data = LineSentence(path_to_file)
6 #-----Create Word2Vec model
7 vec_dim = 10 #Dimension of vectors
8 my_w2v = Word2Vec(sentences = my_data, sg=1, size=vec_dim, window=5,
9                   min_count=0)
10 my_w2kv = my_w2v.wv
11 print('Word2Vec model->', my_w2v)# Just to check if model ok
12 #-----Create FastText model
13 my_ft = FastText(sentences=None, sg=0, size=vec_dim, window=5)
14 my_ft.build_vocab(my_data)
15 numb_words = my_ft.corpus_count
16 print('Total number of words in vocab->', numb_words)
17 epoch_val = my_ft.epochs # epoch is number of training passes
18 print('epoch->', epoch_val)
19 #-----Train FastText model-----
20 my_ft.train(sentences = my_data, total_examples =numb_words,
21            epochs =epoch_val)
22 print(my_ft)
23 #-----Using the Word2Vec KeyedVectors-----
24 # There is no word HanoiViet in vocabulary.
25 #So Word2Vec model throws exception
26 try:
27     print('W2V for Hanoi->', my_w2kv.word_vec('Hanoi'))
28     print('W2V for HanoiViet->', my_w2kv.word_vec('HanoiViet'))
29 except:
30     print('No W2V vector for HanoiViet')
31 #-----Using the FastText model
32 print("'Hanoi' in model->", 'Hanoi' in my_ft.wv.vocab)
33 print("'HanoiViet' in model->", 'HanoiViet' in my_ft.wv.vocab)
34 print("Vector for 'Hanoi'", my_ft['Hanoi'])
35 #FastText finds vector for a word 'HanoiViet' even when not in vocabulary
36 print("Vector for 'HanoiViet'->", my_ft['HanoiViet'])
37 #-----Save FastText model to disk-----
38 path_to_model = r'C:\gensimModels\fasttext.model'
39 my_ft.save(path_to_model)

```

```

40 # Output
41 Word2Vec model-> Word2Vec(vocab=920, size=10, alpha=0.025)
42 Total number of words in vocab-> 19558
43 epoch-> 5
44 FastText(vocab=906, size=10, alpha=0.025)
45 W2V for Hanoi-> [-0.90847117 -0.95077175  0.7906368  -1.4416816   1.2880864
46 -1.2888509
47  0.80608696  0.37066874 -0.3560006  -0.8283224 ]
48 No W2V vector for HanoiViet
49 'Hanoi' in model-> True
50 'HanoiViet' in model-> False
51 Vector for 'Hanoi' [ 0.16464461  0.46445355  1.909329   0.5756341
52 0.7161262 -0.9473982
53 -0.09109268  0.2295851  0.12116321  1.1532576 ]
54 Vector for 'HanoiViet'-> [ 0.47841874  0.8655129  1.7606767  0.62247527
55 0.5488354 -1.2647735
56 -0.2614947  0.00320541  0.13236214  1.1716952 ]

```

Lines 38-39: Here you have saved the FastText model to disk.

The following script shows how to reload the saved model and use it for some language processing.

```

1 from gensim.models.fasttext import FastText
2
3 path_to_model = r'C:\gensimModels\fasttext.model'
4 my_saved_ft = FastText.load(path_to_model)
5 #Note the wv attribute gives an object of type KeyedVector
6 keyed_vec = my_saved_ft.wv
7 print('Vector corresponding to "India"->', keyed_vec['India'])
8 voc_len = len(keyed_vec.vocab)
9 print('Vocabulary length->', voc_len)
10 #Words with lowest index are most common and vice-versa
11 print('Most common 3 words->', keyed_vec.index2word[0],
12       keyed_vec.index2word[1], keyed_vec.index2word[2])
13 print('Least common 3 words->', keyed_vec.index2word[-2],
14       keyed_vec.index2word[-3], keyed_vec.index2word[-4])
15 # KeyedVectors object has similarity and doesnt_match methods
16 print('Similarity London Paris->', keyed_vec.similarity('London', 'Paris'))
17 odd_word = 'Delhi Islamabad London Paris India'.split()
18 print('odd word in sentence->', keyed_vec.doesnt_match(odd_word))
19
20 # Output
21 Vector corresponding to "India"-> [ 0.320433770.240962343.1353068
22 -0.2535870.07314809 -1.1692151
23 -1.1090282 -1.3621750.196722761.9921006 ]
24 Vocabulary length->906
25 Most common 3 words-> California Texas Florida
26 Least common 3 words-> : uncle stepdaughter
27 Similarity London Paris->0.9545561
28 odd word in sentence-> India

```

25.5.2 LDA model

You will write a script to implement LDA in the following steps:

Step1: You will make a list of documents. For the present purpose, each document will be a quotation surrounded by quotation marks. For example, you have a quotation `q1 = "Challenges are what make life interesting and overcoming them is what makes life meaningful."` So `q1` is a document.

Step2: Here you will tokenize the documents (which are string of sentences as mentioned in Step1) and keep the data in form of list of list of tokens where each token is in form of a string. So suppose you have 2 documents "All is well" and "Do your best", then after tokenizing, the collection of documents will become `[["All", "is", "well"], ["Do", "your", "best"]]`. This format is to be noted because in Gensim, the various Classes (like Dictionary) and methods (like `doc2bow`) expect the collection of documents parameter in this format.

Step3: Here you will create a Dictionary object of the `corpora` module, using as parameter, the collection of documents you created in the earlier step.

Step4: You will create a bag of words using the `doc2bow()` method of the Dictionary object you created earlier.

Step5: Create a LDA model using the `LdaModel` class of `gensim.models.ldamodel` module

```

1 import gensim
2 from gensim import corpora
3 from gensim.models.ldamodel import LdaModel
4 from gensim.parsing.preprocessing import remove_stopwords
5 #Some quotations
6 q1 = "Challenges are what make life interesting and overcoming them is what
7 makes life meaningful."
8 q2 = "Three things cannot be long hidden: the sun, the moon, and the truth."
9 q3 = "A soldier will fight long and hard for a bit of colored ribbon."
10 q4 = "As long as poverty, injustice and gross inequality persist in our
11 world, none of us can truly rest."
12 #Create a list of strings. Each string represents a doc
13 docs = [q1, q2, q3, q4]
14 # Create a list of lists
15 all_toks = []
16 for doc in docs:
17     doc_without_stop = remove_stopwords(doc)
18     doc_tok = gensim.utils.tokenize(doc_without_stop, lower=True)
19     new_toks = [atok for atok in doc_tok]
20     all_toks.append(new_toks)
21
22 print('all toks->', all_toks) # Just to check
23 print('*****') # Separator
24 # Create Dictionary object
25 my_dict = corpora.Dictionary(all_toks)
26 # Create corpora
27 my_corpus = [my_dict.doc2bow(a_tok) for a_tok in all_toks]
```

```

28 print('corpus->', my_corpus)
29 print('*****')# Seperator
30 #Create LDA Model
31 my_lda = LdaModel(my_corpus, num_topics=2, id2word = my_dict, passes=20)
32 # Print the model-topic distribution
33 print(my_lda.print_topics(num_topics=6, num_words=4))

35 # Output
36 all toks-> [['challenges', 'life', 'interesting', 'overcoming', 'makes',
37 'life', 'meaningful'], ['three', 'things', 'long', 'hidden', 'sun', 'moon',
38 'truth'], ['a', 'soldier', 'fight', 'long', 'hard', 'bit', 'colored',
39 'ribbon'], ['as', 'long', 'poverty', 'injustice', 'gross', 'inequality',
40 'persist', 'world', 'truly', 'rest']]
41 *****
42 corpus-> [[(0, 1), (1, 1), (2, 2), (3, 1), (4, 1), (5, 1)], [(6, 1), (7, 1),
43 (8, 1), (9, 1), (10, 1), (11, 1), (12, 1)], [(7, 1), (13, 1), (14, 1), (15,
44 1), (16, 1), (17, 1), (18, 1), (19, 1)], [(7, 1), (20, 1), (21, 1), (22, 1),
45 (23, 1), (24, 1), (25, 1), (26, 1), (27, 1), (28, 1)]]
46 *****
47 [(0, '0.085*life" + 0.051*"interesting" + 0.051*"overcoming" +
48 0.051*"meaningful"''), (1, '0.080*long" + 0.048*gross" + 0.048*poverty" +
49 0.048*as"'')]

```

The above script created an LDA model from four quotations q1 to q4.

25.5.3 TfidfModel in Gensim

The following script shows how to create a Tfidf model in Gensim:

```

1 import gensim
2 from gensim.models import TfidfModel
3 from gensim.parsing.preprocessing import remove_stopwords
4 from gensim.corpora import Dictionary
5 #1-----Create documents -----
6 doc_bio = "Biology is the natural science that studies life and living
7 organisms, including their physical structure, chemical processes, molecular
8 interactions, physiological mechanisms, development and evolution"
9 doc_zool = "Zoology or animal biology is the branch of biology that studies
10 the animal kingdom, including the structure, embryology, evolution,
11 classification, habits, and distribution of all animals, both living and
12 extinct, and how they interact with their ecosystems."
13 doc_bot = "Botany, also called plant science, plant biology or phytology, is
14 the science of plant life and a branch of biology."
15 doc_comp_sc = "Computer science is the study of the theory, experimentation,
16 and engineering that form the basis for the design and use of computers."
17 doc_comp_tech = "Information technology is the use of computers to store,
18 retrieve, transmit, and manipulate data, or information, often in the context
19 of a business or other enterprise"
20 doc_software = "Computer software, or simply software, is a generic term that

```

```

21 refers to a collection of data or computer instructions that tell the
22 computer how to work, in contrast to the physical hardware from which the
23 system is built, that actually performs the work."
24 doc_hardware = "Computer hardware includes the physical parts or components
25 of a computer, such as the central processing unit, monitor, keyboard,
26 computer data storage, graphic card, sound card, speakers and motherboard. By
27 contrast, software is instructions that can be stored and run by hardware."
28
29 all_docs = [doc_bio, doc_zoo1, doc_bot, doc_comp_sc,
30             doc_comp_tech, doc_software, doc_hardware]
31 #2-----Create list of lists of strings-----
32 def doc2tok(doc_all):
33     all_toks = []
34     for a_doc in doc_all:
35         doc_without_stop = remove_stopwords(a_doc)
36         doc_tok = gensim.utils.tokenize(doc_without_stop, lower=True)
37         new_toks = [atok for atok in doc_tok]
38         all_toks.append(new_toks)
39     return all_toks
40 all_toks = doc2tok(all_docs)
41 print('All tokens->', all_toks)
42 #3-----Create Dictionary-----
43 my_dict = Dictionary(documents = all_toks)
44 my_dict.save(r'C:\gensimDict\someDocs.dict')
45 #4-----Create corpus-----
46 my_corpus = [my_dict.doc2bow(a_tok) for a_tok in all_toks]
47 print('Corpus->', my_corpus)
48 #5-----Create TfIdf model-----
49 my_tfidf = TfIdfModel(my_corpus)
50 print(my_tfidf)
51 #6-----Show Tf-Idf for document no1-----
52 tfidf_doc1 = my_tfidf[my_corpus[0]]
53 print('Tfidf_doc1->', tfidf_doc1)
54 # Output ----Truncated to save space
55 All tokens-> [['biology', 'natural', 'science', 'studies', 'life', 'living', 'organisms',
56 'including', 'physical', 'structure', 'chemical', 'processes',
57 'molecular', 'interactions', 'physiological', 'mechanisms', 'development',
58 'evolution'], ['zoology', ...]]
59 Corpus-> [[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1),
60 (8, 1), (9, 1), (10, 1), (11, 1), (12, 1), (13, 1), (14, 1), (15, 1), (16,
61 1), (17, 1)], [(0, 2), ...]]
62 TfIdfModel(num_docs=7, num_nnz=108)
63 TfIdf_doc1-> [(0, 0.12540617304247417), (1, 0.2880086877193568), (2,
64 0.2880086877193568), (3, 0.18541792320417194), (4, 0.18541792320417194), (5,
65 0.2880086877193568), (6, 0.18541792320417194), (7, 0.18541792320417194), (8,
66 0.2880086877193568), (9, 0.2880086877193568), (10, 0.2880086877193568), (11,
67 0.2880086877193568), (12, 0.12540617304247417), (13, 0.2880086877193568),
68 (14, 0.2880086877193568), (15, 0.12540617304247417), (16,
69 0.18541792320417194), (17, 0.18541792320417194)]

```



25.5.5 API for pre-trained NLP models and datasets in Gensim

The creators of Gensim (RaRe Technologies) have also provided some corpora and pre-trained models on github¹⁸. Further to make these corpora and models easy to download and deploy, they have provided APIs which download the data automatically. To download the dataset and pre-trained model you simply need to use the functionality provided by gensim.downloader module. In the following code, you will download a dataset named “text8” and a pre-trained model named “glove-twitter-25”. The dataset and model will be downloaded and stored in the ~/gensim-data home folder. (On author’s computer it is C:\Users\DS\gensim-data). You can use the api.info method to get information about a particular dataset or model. The code is as follows:

```

1 import gensim.downloader as api
2 #Download dataset named text8 and print its metadata
3 corpus_text8 = api.load('text8')
4 print('text8->', api.info('text8'))
5 print('text8 corpus object->', corpus_text8)
6 #Download dataset named text8 and print its metadata
7 model_gt25 = api.load("glove-twitter-25")
8 print('glove-twitter-25 model details->', api.info('glove-twitter-25'))
9 print('glove-twitter-25 model object->', model_gt25)

22 # Output. . .
23 text8-> {'num_records': 1701, 'record_format': 'list of str (tokens)',
24 'file_size': 33182058, 'reader_code': 'https://github.com/RaRe-
25 Technologies/gensim-data/releases/download/text8/__init__.py', 'license': 'not
26 found', 'description': 'First 100,000,000 bytes of plain text from Wikipedia.
27 Used for testing purposes; see wiki-english-* for proper full Wikipedia
28 datasets.', 'checksum': '68799af40b6bda07dfa47a32612e5364', 'file_name':
29 'text8.gz', 'read_more': ['http://matmahoney.net/dc/textdata.html'], 'parts':
30 1}
31 text8 corpus object-><text8.Dataset object at 0x27CB1A90>
32 glove-twitter-25 model details-> {'num_records': 1193514, 'file_size':
33 109885004, 'base_dataset': 'Twitter (2B tweets, 27B tokens, 1.2M vocab,
34 uncased)', 'reader_code': 'https://github.com/RaRe-Technologies/gensim-
35 data/releases/download/glove-twitter-25/__init__.py', 'license':
36 'http://opendatacommons.org/licenses/pddl/', 'parameters': {'dimension': 25},
37 'description': 'Pre-trained vectors based on 2B tweets, 27B tokens, 1.2M vocab,
38 uncased (https://nlp.stanford.edu/projects/glove/).', 'preprocessing':
39 'Converted to w2v format with `python -m gensim.scripts.glove2word2vec -i
40 <fname> -o glove-twitter-25.txt`.', 'read_more':
41 ['https://nlp.stanford.edu/projects/glove/',
42 'https://nlp.stanford.edu/pubs/glove.pdf'], 'checksum':
43 '50db0211d7e7a2dcd362c6b774762793', 'file_name': 'glove-twitter-25.gz', 'parts':
44 1}
45 glove-twitter-25 model object-><gensim.models.keyedvectors.Word2VecKeyedVectors
46 object at 0x27CB1C10>

```

¹⁸The corpora and models are available here: <https://github.com/RaRe-Technologies/gensim-data>

You can create a model from the dataset you have downloaded. The following code shows how to create a Word2Vec model from the downloaded dataset and to use the model to find the vector representation of a word:

```

1 from gensim.models.word2vec import Word2Vec
2 import gensim.downloader as api
3 corpus_text8 = api.load('text8')
4 # Create w2v from downloaded dataset, with dimension 25
5 my_w2v = Word2Vec(sentences = corpus_text8, size = 25)
6 w2v_kv = my_w2v.wv
7 #Use the model to get vector representation of a word
8 print('Vector of king->', w2v_kv.word_vec('king'))

9 # Output. . .
10 Vector of king-> [-0.6764224 -1.18675822.6187992 -1.89375411.64215841.7179773
11 -2.07794242.9210784 -3.4910505 -3.08494424.0966377 -0.619812
12 -1.8496962 -0.99269050.106861491.92018322.91087462.430908
13 -3.64872225.66998674.921379 -3.76596470.14415641.2285326
14 1.6053306 ]

```

You can even use the pre-trained model directly as shown in the following code:

```

1 import gensim.downloader as api
2 # Load the pretrained model
3 model_gt25 = api.load("glove-twitter-25")
4 # Use the model
5 print('Another vector of king->', model_gt25.word_vec('king'))
6 print('Words similar to king->', model_gt25.most_similar('king'))

7 # Output. . .
8 Another vector of king-> [-0.74501 -0.119920.373290.36847 -0.4472 -
9 0.22880.70118
10 0.828720.39486 -0.583470.414880.37074 -3.6906 -0.20101
11 0.11472 -0.346610.362080.095679 -0.017650.68498 -0.049013
12 0.54049 -0.21005 -0.653970.64556 ]
13 Words similar to king-> [( 'prince', 0.93374103307724), ( 'queen',
14 0.9202420711517334), ( 'aka', 0.9176923036575317), ( 'lady', 0.9163240790367126),
15 ( 'jack', 0.9147354960441589), ( "'s", 0.9066898226737976), ( 'stone',
16 0.8982374668121338), ( 'mr.', 0.8919408917427063), ( 'the', 0.8893439173698425),
17 ( 'star', 0.8892088532447815)]

```

CONCEPTUAL QUESTIONS

1. What is a “corpus” in Gensim?
2. What is a “model”?
3. What is sparse vector representation? Suppose you have a vector say $A^T = (1, 1, 0, 0, 0, 2, 0, 0, 0, 0, 3)$. How would you represent it as a sparse vector?
4. What does the term “toy data set” mean?
5. What does the term “bag of words” mean?

6. What does the “Dictionary” class do in Gensim?

As per the documentation on Dictionary class, it has an attribute token2id. The docstring of the Dictionary class shows the details of the token2id attribute as follows:

```
token2id : dict of (str, int)
    token -> tokenId.
```

What does “token -> tokenId” mean?

7. Similarly, the docstring shows that the Dictionary class also has an attribute namely id2token and gives its details as follows:

```
id2token : dict of (int, str)
    Reverse mapping for token2id, initialized in a lazy manner to save memory
    What does this mean? What does the term “initializing in a lazy manner mean”?
```

8. In Gensim, you can use a “Dictionary object” to convert a document into a BoW representation consisting of tuples. Each tuple has two integers. Explain what these integers represent.
9. In Gensim, a Dictionary object can be “trimmed” using the filter_extreme() method. What is “trimming”? How does the filter_extreme() method work?
10. The Word2Vec class in Gensim has a very important attribute namely wv. What is this attribute “wv”? What does it do?
11. What is Word2VecKeyedVectors? How are they different from Word2Vec? Which one of them is a “model”?
12. When you already have a model in form of Word2Vec, then why do you need a separate Word2VecKeyedVectors object?
13. The Word2VecKeyedVectors consists of a “mapping” between a “word which is a string and its “embedding which is a ndarray”. Explain.
14. What is the FastText algorithm? How does it differ from Word2Vec?
15. What is topic modelling? What is LDA? What does it do?
16. Explain TfidfModel in Gensim.

EXERCISE

1. Look at the script below. Certain questions are given in comments (In bold). Answer them. (You can confirm your answers by running the script.)

```
1 from gensim.corpora import Dictionary
2 my_text = [['A', 'cat'], ['A', 'rat'], ['A', 'mat']]
3 my_dict = Dictionary(my_text)
4 print('documents in collection->', my_dict.num_docs)
5 # What is the number of tokens in my_dict?
6 print('number of tokens in dictionary->', len(my_dict))
7 # How many words are processed in the Dictionary object?
8 print('number of processed words->', my_dict.num_pos)
9 # Why is token to id empty?
10 print('tokens and their id->', my_dict.id2token) # Empty
11 # What is the token at id 1?
12 print('token at id 1->', my_dict[1])
13 # Now why is token to id populated?
14 print('tokens and their id->', my_dict.id2token) # Now not Empty
```

BEYOND TEXTBOOK

1. Suppose you have two different Dictionary objects. Can you merge them into a single Dictionary object?

Yes. Gensim provides a method of Dictionary class, i.e., `merge_with()`¹⁹

The following example shows how two dictionaries are merged.

```

1  from gensim.corpora import Dictionary
2  # Create 2 corpora
3  corpora1 = [['A', 'B', 'C'], ['D', 'E', 'F'], ['E', 'F', 'G']]
4  corpora2 = [['A', 'D', 'E'], ['G', 'H'], ['X', 'Y', 'Z']]
5  d1 = Dictionary(corpora1)
6  d2 = Dictionary(corpora2)
7  print('d1->', d1.token2id)
8  print('d2->', d2.token2id)
9  # tokens at id 1 are different for d1 and d2
10 print(d1[1])
11 print(d2[1])
12 # merge d2 with d1
13 d1.merge_with(d2)
14 # Lets check d1 on a document which has tokens of d2 also
15 # A and D are common. B only in d1. H and X only in d2
16 a_doc = ['A', 'B', 'D', 'H', 'X']
17 word_vec = d1.doc2bow(a_doc)
18 print(word_vec)
19 # Output...
20 d1-> {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6}
21 d2-> {'A': 0, 'D': 1, 'E': 2, 'G': 3, 'H': 4, 'X': 5, 'Y': 6, 'Z': 7}
22 B
23 D
24 [(0, 1), (1, 1), (3, 1), (7, 1), (8, 1)]

```

Notice the following:

- If there are common tokens in d1 and d2, then the id of d1 is used.
- If there are tokens only of d2, then these tokens are present in the BoW representation of the document, but they are assigned new id. For example, in the above case, term H and X were present in d2 only. So when you convert the document a_doc to its Bow, then you get new id for H and X.

¹⁹The source code of this method is available from line 535 onwards at:

<https://github.com/RaRe-Technologies/gensim/blob/develop/gensim/corpora/dictionary.py>

1. Twitter assignment

Twitter is a great social media platform for downloading text data. The following project is proposed:

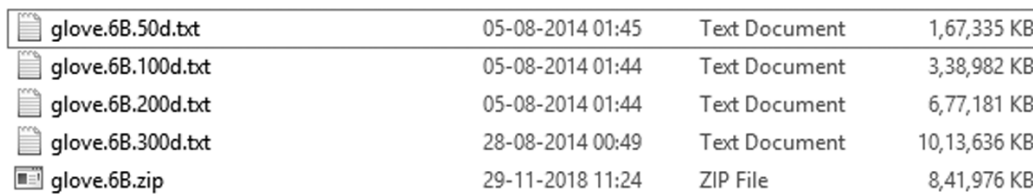
- Identify some subject specific twitter handles (maybe sports, nature, politics, news, etc.)
- Make a dictionary of these handles. The key to the dictionary will be the subject and the values will be the handles relating to this subject. (You did a similar exercise using the tweepy library in an earlier chapter.)
- Download tweets (200 each) from each of these handles and get their text data.
- Clean up the text data (for example, you may remove special characters or URLs, etc., using the re (regular expression) module.
- From the cleaned text dat, create a subject specific Gensim dictionary.
- Use this dictionary to form a BoW representation of text related to the subject.

2. GloVe assignment

The Stanford University has developed GloVe²⁰ which stands for Global Vectors for Word Representation. GloVe is an Unsupervised learning algorithm.

Through this assignment you will learn to use the word to vector representation created by GloVe for various text analysis task. Note that this assignment is not based on Gensim library. The text representation of GloVe can be downloaded from this²¹ link.

The data is in zipped format. Screen shot of unzipped data is shown in Figure 25.4.



glove.6B.50d.txt	05-08-2014 01:45	Text Document	1,67,335 KB
glove.6B.100d.txt	05-08-2014 01:44	Text Document	3,38,982 KB
glove.6B.200d.txt	05-08-2014 01:44	Text Document	6,77,181 KB
glove.6B.300d.txt	28-08-2014 00:49	Text Document	10,13,636 KB
glove.6B.zip	29-11-2018 11:24	ZIP File	8,41,976 KB

FIGURE 25.4 Screen shot of GloVe models downloaded from <http://nlp.stanford.edu/> and then unzipped. The numbers 100, 200 and 300 indicate the ‘dimensions’ of the vectors

On unzipping you get 4.txt files. The number 50d indicates that the vector representation of the word has 50 dimensions. Similarly the file with name containing 100d has each word represented as a vector with 100 dimensions.

The screen shot of the word “cricket” and its 50 dimension vector representation is shown in Figure 25.5.

For the sake of comparison, Figure 25.6 shows the 100 dimension representation of the same word “cricket”:

²⁰See: <https://nlp.stanford.edu/projects/glove/>

²¹Download from: <http://nlp.stanford.edu/data/glove.6B.zip>

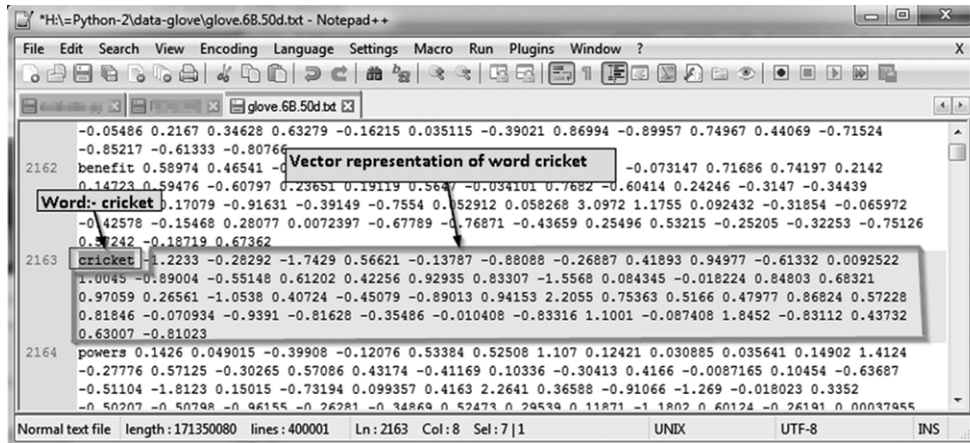


FIGURE 25.5 Screen shot of the vector representation of word “cricket” in 50 dimensions.

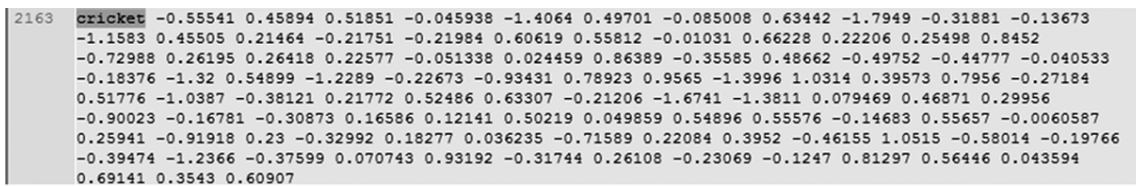


FIGURE 25.6 Screen shot of the vector representation of word “cricket” in 100 dimensions.

In this assignment the focus is not on trying to understand that “how GloVe was created,” rather the focus is on “how to use a pre-trained model like GloVe”. Following is an example script of how to create a word-vector model representation of the words in the downloaded text file. Here the 50d vectors have been used. The steps in the script are:

- Create a dictionary `glove_model` to store the model.
- Open the text file and read its lines into a list called `all_lines`.
- Take each line in `all_lines` one by one and split it into individual tokens. (Note the first token will be a string representing the word whose vector is in the rest of the tokens. Note that the first token will have index 0. Note that the rest of the tokens are floating numbers but in string format.)
- Make the first token, i.e., at index 0 as key of the `glove_model` dictionary and make the rest of the tokens as the value of the dictionary. (Note you need to convert each value from a string to float.)
- Finally you can get the vector representation of a particular word. (Ideally you should surround this in a try-catch block in case the vector for a word does not exist.)

```
1 import numpy as np
2 # Give path to your file location instead
3 glove_file = r"H:\Python-2\stanford.edu-glove.6B\glove.6B.50d.txt"
4 # glove_model is a dictionary whose keys are the words
5 # and whose values are the vectors of these words
6 glove_model = {}
```

```

7 with open(glove_file, encoding="utf8" ) as fhandle:
8     all_lines = fhandle.readlines()
9     for a_line in all_lines:
10         split_line = a_line.split()
11         # The first item in each line is the word
12         a_word = split_line[0]
13         # The rest of the line is the word vector but in string form
14         # So convert them to float
15         word_vec = np.array(split_line[1:], dtype = 'float32')
16         # a_word becomes key, word_vec becomes value of glove_model
17         glove_model[a_word] = word_vec
18
19 # get vector for word 'cricket'
20 print(glove_model['cricket'])
21 # You can check that the vector has 50d
22 print('dimension of vector->', len(glove_model['cricket']))
23
24 # Output...
25 [-1.2233   -0.28292  -1.7429    0.56621  -0.13787  -0.88088
26  -0.26887   0.41893   0.94977  -0.61332   0.0092522  1.0045
27  -0.89004  -0.55148   0.61202   0.42256   0.92935   0.83307
28  -1.5568   0.084345  -0.018224  0.84803   0.68321   0.97059
29   0.26561  -1.0538   0.40724  -0.45079  -0.89013   0.94153
30   2.2055   0.75363   0.5166   0.47977   0.86824   0.57228
31   0.81846  -0.070934  -0.9391  -0.81628  -0.35486  -0.010408
32  -0.83316   1.1001  -0.087408   1.8452  -0.83112   0.43732
33   0.63007  -0.81023  ]
34 dimension of vector-> 50

```

3. Using vector representation of a word to get a “similar” word.

The previous exercise showed how to get the vector representation of a word from a pre-trained word-vector model like GloVe. In this exercise, you are required to get words similar to a given word based on the similarity of their vector representation.

There could be many ways of doing this. One way of doing this in NumPy is shown in the script below. This script finds words closest to the word ‘gun’.

The steps in the logic are as follows:

- Write a function which accepts a word (‘gun’ in this case), whose close words are to be found.
- Now your model is a dictionary whose keys are the words and whose values are the vector representations of these words. So get the vector representation of the given word, i.e., ‘gun’ from the model.
- Now go over each word in the dictionary model one by one and compare the vector of each word to the vector of the given word ‘gun’. For comparison, use a NumPy method `isclose()`²². The `isclose()` method provides a parameter `atol` which determines the allowable difference in magnitude between individual items of the 2 arrays. Here an arbitrary value of 1.0 has been taken. You can try with different values. In general if you increase the tolerance number, more words will figure in the `sim_words` list.

²²`isclose()` method returns an array of boolean True or False. See:

<https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.allclose.html#numpy.allclose>

- Add up the number of True. Take an arbitrary number as cut-off. In the given script it is 48 meaning that if 48 out of the 50 dimensions are within the tolerance of 0.1, then the word will be considered close to the given word. You can vary the threshold. In general if you decrease the threshold, more words will figure in the `sim_list`

```

1 import numpy as np
2 # Give path to your file location instead
3 glove_file = r"H:\Python-2\stanford.edu-glove.6B\glove.6B.50d.txt"
4 #----Step1--- Create the model
5 # glove_model is a dictionary whose keys are the words
6 # and whose values are the vectors of these words
7 glove_model = {}
8 with open(glove_file, encoding="utf8" ) as fhandle:
9     all_lines = fhandle.readlines()
10    for a_line in all_lines:
11        split_line = a_line.split()
12        # The first item in each line is the word
13        a_word = split_line[0]
14        # The rest of the line is the word vector but in string form
15        # So convert them to float
16        word_vec = np.array(split_line[1:], dtype = 'float32')
17        # a_word becomes key, word_vec becomes value of glove_model
18        glove_model[a_word] = word_vec
19 #----Step2--- Write a function which finds similar words
20 def get_sim(given_word):
21     list_close_words = []
22     # get vector of given word
23     given_vec = glove_model[given_word]
24     # iterate over all words and their vectors in model
25     for a_word, a_vec in glove_model.items():
26         # vec_diff is a bool array. Can chose different val for atol
27         vec_diff = np.isclose(given_vec, a_vec, atol = 1.0)
28         # get number of True counts. Note True is 1
29         true_count = np.sum(vec_diff)
30         # You can choose a different val for thresh
31         thresh = 48
32         if true_count > thresh:
33             list_close_words.append(a_word)
34     return list_close_words
35 #---- Step3--- Test the function
36 sim_words = get_sim('gun')
37 print(sim_words)
38 print(len(sim_words))
39
40 # Output...
41 ['gun', 'guns', 'weapon']
42 3

```

4. GloVe in Gensim

The assignment here is to convert the GloVe word-vector representation into a format which can be used by the Gensim library. The advantage of doing this is that you can then use the various methods of the Gensim library to do various text processing tasks using GloVe.

The format of word2vec in Gensim is slightly different from that in Glove, but Gensim provides a tool for converting between the two formats²³.

The only difference between the glove vector file format and the word2vec file format is one line at the beginning of the .txt of the word2vec format which has
<num words> <num dimensions>

The tool works on command line. The general format on command line is as follows:

```
1 python -m gensim.scripts.glove2word2vec --input <glove_file> --output <w2v_file>
```

For example on the author's computer the glove file is at: H:\=Python-2\stanford.edu-glove.6B\glove.6B.50d.txt and the output file was at: H:\=Python-2\stanford.edu-glove.6B\glove.6B.50d.w2v.txt. So the screen shot of command on the author's computer was as shown in Figure 25.7.

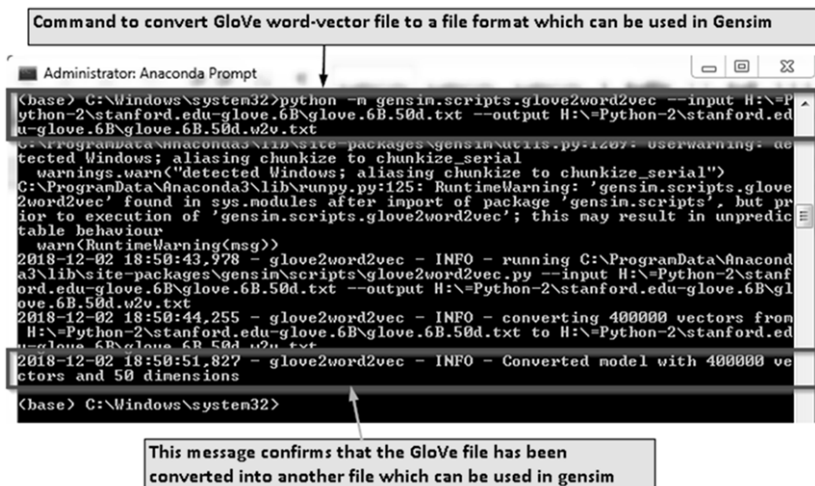


FIGURE 25.7 Screen shot shows how a Word2Vector file in glove format is converted to a format suitable for Gensim.

So the task is to load a word-vector GloVe file into a Gensim model and then to use this model for text analysis. An example script of how this can be done is as follows:

```

1 from gensim.models import KeyedVectors
2 # Path to glove_file converted from use in Gensim
3 path_gensim_glove = r'H:\=Python-2\stanford.edu-glove.6B\glove.6B.50d.w2v.txt'
4 # Create a gensim model from the glove file
  
```

²³The source code for this tool is available at the following link. Examination of the code show that there is very little difference in the format of the two word-vector files.

See: <https://github.com/RaRe-Technologies/gensim/blob/develop/gensim/scripts/glove2word2vec.py>

```

5 glove_model = KeyedVectors.load_word2vec_format(path_gensim_glove,
6 binary=False)
7 # Check that the loaded model is of type Word2VecKeyedVectors
8 print(type(glove_model))
9 # Get 5 words similar to 'fly' and 'plane' but not related to 'bird'
10 print(glove_model.most_similar(positive = ['fly', 'plane'],
11                                     negative = ['bird'], topn = 5))
12 # OUTPUT
13 <class 'gensim.models.keyedvectors.Word2VecKeyedVectors'>
14 [(('planes', 0.8001785278320312), ('takeoff', 0.7513437271118164), ('airplane',
15 0.7508190870285034), ('flight', 0.7457747459411621), ('landed',
16 0.7353001832962036))]
17 C:\ProgramData\Anaconda3\lib\site-packages\gensim\matutils.py:737: Future
18 Warning: Conversion of the second argument of issubdtype from `int` to
19 `np.signedinteger` is deprecated. In future, it will be treated as `np.int32 ==
20 np.dtype(int).type`.
21 if np.issubdtype(vec.dtype, np.int):

```