Lab 6. Vector Representation

Overview

This lab introduces you to the various ways in which text can be represented in the form of vectors. You will start by learning why this is important, and the different types of vector representation. You will then perform one-hot encoding on words, using the `preprocessing` package provided by scikit-learn, and character-level encoding, both manually and using the powerful Keras library. After covering learned word embeddings and pre-trained embeddings, you will use `Word2Vec` and `Doc2Vec` for vector representation for **Natural Language Processing** (**NLP**) tasks, such as finding the level of similarity between multiple texts.

What Is a Vector?

As an example, let's say that we're defining the weather at a given place using five features: temperature, humidity, precipitation, wind speed, and air pressure. The units that these would be measured in are Celsius, percentage, centimeters, kilometers per hour (km/h), and millibar (mbar), respectively. The following are the values for two places:

Weather	Place 1	Place 1
Temperature	25	32
Humidity	50	60
Precipitation	1	0
Wind speed	18	7
Air pressure	1200.0	1019.0

So, we can represent the weather of these places in vector form as follows:

- Vector for place 1: [25, 50, 1, 18, 1200.0]
- Vector for place 2: [32, 60, 0, 7, 1019.0]

Vector representation techniques can be broadly classified into two categories:

- Frequency-based embeddings
- Learned word embeddings

Exercise 6.01: Word-Level One-Hot Encoding

In this exercise, we will one-hot encode words with the help of the `preprocessing` package provided by the scikit-learn library.

Follow these steps to implement this exercise:

1. Open a Jupyter notebook.
2. First, load the file containing the lines from the novel using the `Path` class provided by the `pathlib` library to specify the location of the file. Insert a new cell and add the following code:

```
```
from pathlib import Path
data = Path('../data')
```
```

```
novel_lines_file = data / 'novel_lines.txt'
```

3. Now that you have the file, open it and read its contents. Use the `open()` and `read()` functions to perform these actions. Store the results in the `novel_lines` file variable. Insert a new cell and add the following code to implement this:

```
with novel_lines_file.open() as f:
    novel_lines_raw = f.read()
```

4. After reading the contents of the file, load it by inserting a new cell and adding the following code:

```
novel_lines_raw
```

The code generates the following output:

```
'It is a truth universally acknowledged, that a single man in possession of a good fortun
e, must be in want of a wife.\nHowever little known the feelings or views of such a man ma
y be on his first entering a neighbourhood, this truth is so well fixed in the minds of th
e surrounding families, that he is considered as the rightful property of some one or othe
r of their daughters.\n"My dear Mr. Bennet," said his lady to him one day, "have you heard
that Netherfield Park is let at last?"\nMr. Bennet replied that he had not.\n"But it is,"
returned she; "for Mrs. Long has just been here, and she told me all about it."\nMr. Benne
t made no answer.\n"Do not you want to know who has taken it?" cried his wife impatientl
y.\n"You want to tell me, and I have no objection to hearing it."\nThis was invitation eno
ugh.\n"Why, my dear, you must know, Mrs. Long says that Netherfield is taken by a young ma
n of large fortune from the north of England; that he came down on Monday in a chaise and
four to see the place, and was so much delighted with it that he agreed with Mr. Morris im
mediately; that he is to take possession before Michaelmas, and some of his servants are t
o be in the house by the end of next week."\n"What is his name?"\n"Bingley."\n"Is he marri
ed or single?"\n"Oh! single, my dear, to be sure! A single man of large fortune; four or f
ive thousand a year. What a fine thing for our girls!"\n"How so? how can it affect the
m?"\n"My dear Mr. Bennet," replied his wife, "how can you be so tiresome! You must know th
at I am thinking of his marrying one of them."\n"Is that his design in settling here?"\n"D
esign! nonsense, how can you talk so! But it is very likely that he may fall in love with
one of them, and therefore you must visit him as soon as he comes."\n"I see no occasion fo
r that. You and the girls may go, or you may send them by themselves, which perhaps will b
e still better, for as you are as handsome as any of them, Mr. Bingley might like you the
best of the party."\n"My dear, you flatter me. I certainly have had my share of beauty, bu
t I do not pretend to be any thing extraordinary now. When a woman has five grown up daugh
ters, she ought to give over thinking of her own beauty."\n"In such cases, a woman has not
often much beauty to think of."\n"But, my dear, you must indeed go and see Mr. Bingley whe
n he comes into the neighbourhood."'
```

```
In the output, you will see a lot of newline characters. This is
because we loaded the entire content at once into a single variable
instead of separate lines. You will also see a lot of
non-alphanumeric characters.
```

5. The main objective is to create one-hot vectors for each word in the file. To do this, construct a vocabulary, which is the entire list of unique words in the file, by tokenizing the string into words and removing newlines and non-alphanumeric characters. Define a function named `clean_tokenize()` to do this. Store the vocabulary created using `clean_tokenize()` inside a variable named `novel_lines`. Add the following code:

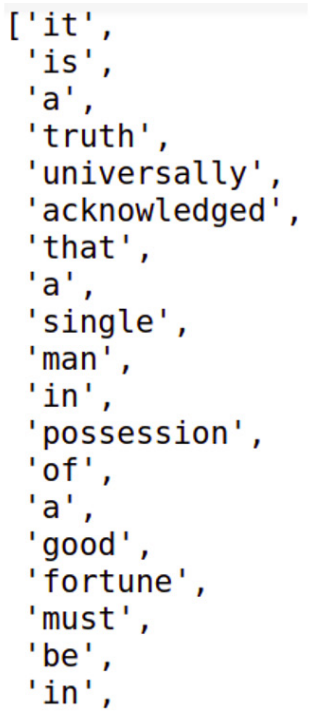
```
import string
import re
```

```
alpha_characters = str.maketrans('', '', string.punctuation)
def clean_tokenize(text):
    text = text.lower()
    text = re.sub(r'\n', '*** ', text)
    text = text.translate(alpha_characters)
    text = re.sub(r' +', ' ', text)
    return text.split(' ')
novel_lines = clean_tokenize(novel_lines_raw)
```

6. Take a look at the content inside `novel_lines` now. It should look like a list. Insert a new cell and add the following code to view it:

   ```
   novel_lines
   ```

   The code generates the following output:

7. Insert a new cell and add the following code to convert the list to a NumPy array and print the shape of the array:

   ```
   import numpy as np
   novel_lines_array = np.array([novel_lines])
   novel_lines_array = novel_lines_array.reshape(-1, 1)
   novel_lines_array.shape
   ```

   The code generates the following output:

   ```
   (459, 1)
   ```

As you can see, the `novel_lines_array` array consists of `459` rows and `1` column. Each row is a word in the original `novel_lines` file.

Note:

NumPy arrays are more specific to NLP algorithms than Python lists. It is the format that is required for the scikit-learn library, which we will be using to one-hot encode words.

8. Now use encoders, such as the `LabelEncoder()` and `OneHotEncoder()` classes from scikit-learn's `preprocessing` package, to convert `novel_lines_array` to one-hot encoded format. Insert a new cell and add the following lines of code to implement this:

```
from sklearn import preprocessing
labelEncoder = preprocessing.LabelEncoder()
novel_lines_labels = labelEncoder.fit_transform(\
                     novel_lines_array)
import warnings
warnings.filterwarnings('ignore')
wordOneHotEncoder = preprocessing.OneHotEncoder()
line_onehot = wordOneHotEncoder.fit_transform(\
              novel_lines_labels.reshape(-1,1))
```

   In the code, the `LabelEncoder()` class encodes the labels, and the `fit_transform()` method fits the label encoder and returns the encoded labels.

9. To check the list of encoded labels, insert a new cell and add the following code:

```
novel_lines_labels
```

The preceding code generates output that looks as follows:

```
array([ 82,  81,   0, 177, 178,   2, 163,   0, 151,  96,  77, 137, 122,
         0,  58,  52, 109,  13,  77, 183, 122,   0, 192,  73,  92,  85,
       164,  45, 127, 181, 122, 156,   0,  96,  99,  13, 125,  70,  47,
        41,   0, 112, 172, 177,  81, 152, 186,  49,  77, 164, 103, 122,
       164, 158,  44, 163,  64,  81,  29,  11, 164, 142, 139, 122, 153,
       126, 127, 128, 122, 165,  31, 110,  33, 106,  17, 143,  70,  86,
       175,  69, 126,  32,  63, 197,  65, 163, 113, 133,  81,  89,  12,
        88, 106,  17, 140, 163,  64,  60, 118,  21,  82,  81, 141, 150,
        51, 107,  93,  62,  83,  15,  68,   7, 150, 176, 100,   5,   1,
        82, 106,  17,  95, 115,   8,  36, 118, 197, 183, 175,  84, 190,
        62, 160,  82,  30,  70, 192,  76, 197, 183, 175, 162, 100,   7,
        74,  63, 115, 120, 175,  66,  82, 172, 184,  80,  40, 191, 110,
        33, 197, 109,  84, 107,  93, 144, 163, 113,  81, 160,  22,   0,
       198,  96, 122,  87,  52,  54, 164, 117, 122,  39, 163,  64,  23,
        37, 125, 104,  77,   0,  27,   7,  53, 175, 145, 164, 136,   7,
       184, 152, 108,  34, 194,  82, 163,  64,   4, 194, 106, 105,  75,
       163,  64,  81, 175, 159, 137,  16, 101,   7, 153, 122,  70, 147,
        10, 175,  13,  77, 164,  71,  22, 164,  38, 122, 114, 185, 187,
        81,  70, 111,  20,  81,  64,  97, 127, 151, 124, 151, 110,  33,
       175,  13, 157,   0, 151,  96, 122,  87,  52,  53, 127,  48, 173,
         0, 196, 187,   0,  46, 169,  51, 130,  55,  72, 152,  72,  24,
        82,   3, 166, 110,  33, 106,  17, 140,  70, 192,  72,  24, 197,
        13, 152, 174, 197, 109,  84, 163,  74,   6, 171, 122,  70,  98,
       126, 122, 166,  81, 163,  70,  35,  77, 148,  68,  35, 116,  72,
        24, 197, 161, 152,  21,  82,  81, 180,  91, 163,  64,  99,  43,
        77,  94, 194, 126, 122, 166,   7, 168, 197, 109, 182,  69,  11,
       154,  11,  64,  28,  74, 145, 115, 121,  51, 163, 197,   7, 164,
        55,  99,  57, 127, 197,  99, 146, 166,  22, 167, 189, 135, 193,
        13, 155,  19,  51,  11, 197,  10,  11,  61,  11,   9, 122, 166,
```

```
The `OneHotEncoder()` class encodes the categorical
integer features as a one-hot numeric array. The
`fit_transform()` method of this class takes the
`novel_lines_labels` array as input. This is a numeric
array, and each feature included in this array is encoded using the
one-hot encoding scheme.
```

10. Create a binary column for each category. A **sparse matrix** is returned as output. To view the matrix, insert a new cell and type the following code:

    ```
    line_onehot
    ```

    The code generates the following output:

    ```
    <459x199 sparse matrix of type '<class 'numpy.float64'>'
                    With 459 stored elements in Compressed Sparse Row format>
    ```

11. To convert the sparse matrix into a **dense array**, use the `toarray()` function. Insert a new cell and add the following code to implement this:

    ```
    line_onehot.toarray()
    ```

The code generates the following output:

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

Note The preceding output shows that we have achieved our objective of one-hot encoding words.

Character-Level One-Hot Encoding

In character-level one-hot encoding, we assign a numeric value to all the possible characters. Consider the word "hello". Let's say our vocabulary contains only twenty-six characters, so our **dictionary** will look like this:

{'a': 0

'b': 1

'c': 2

'd': 3

.....'z': 25}

Now, 'h' will be represented as [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]. Similarly, 'e' can be represented as [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]. Let's see how we can implement this in the next exercise.

Exercise 6.02: Character One-Hot Encoding -- Manual

In this exercise, we will create our own function that can one-hot encode the characters of the word "data". Follow these steps to complete this exercise:

1. Open a Jupyter notebook.
2. To one-hot encode the characters of a given word, create a function named `onehot_word()`. Within this function, create a `lookup` table for each of the characters in the given word. Then, map each character to an index. Add the following code to implement this:

```
```
def onehot_word(word):
    lookup = {v[1]: v[0] for v in enumerate(set(word))}
    word_vector = []
```
```

3. Next, loop through the characters in the word and create a vector named `one_hot_vector` of the same size as the number of characters in the lookup. This vector is filled with zeros. Then, use the `lookup` table to find the position of the character and set that character's value to `1`.

   Note: Execute the code for *step 1* and *step 2* together.

Add the following code:

```
    for c in word:
        one_hot_vector = [0] * len(lookup)

        one_hot_vector[lookup[c]] = 1
        word_vector.append(one_hot_vector)
    return word_vector
```

The function created earlier will return a word vector.

4. Once the `onehot_word()` function has been created, test it by adding some input as a parameter. Add the word "data" as an input to the function. To implement this, add a new cell and write the following code:

```
onehot_vector = onehot_word('data')
print(onehot_vector)
```

The code generates the following output:

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

Exercise 6.03: Character-Level One-Hot Encoding with Keras

In this exercise, we will perform one-hot encoding on a given word using the Keras library. Follow these steps to implement this exercise:

1. Open a Jupyter notebook.
2. Insert a new cell and the following code to import the necessary libraries:

```
```
from keras.preprocessing.text import Tokenizer
import numpy as np
```
```

3. Once you have imported the `Tokenizer` class, create an instance of it by inserting a new cell and adding the following code:

```
char_tokenizer = Tokenizer(char_level=True)
```

Since you are encoding at the character level, in the constructor, `char_level` is set to `True`.

Note:

By default, `char_level` is set to `False` if we are encoding words.

4. To test the `Tokenizer` instance, you will require some text to work on. Insert a new cell and add the following code to assign a string to the `text` variable:

```
```
text = 'The quick brown fox jumped over the lazy dog'
```
```

5. After getting the text, use the `fit_on_texts()` method provided by the `Tokenizer` class. Insert a new cell and add the following code to implement this:

```
char_tokenizer.fit_on_texts(text)
```

In this code, `char_tokenizer` will break `text` into characters and internally keep track of the tokens, the indices, and everything else needed to perform one-hot encoding.

6. Now, look at the possible output. One type of output is the sequence of the characters that is, the integers assigned with each character in the text. The `texts_to_sequences()` method of the `Tokenizer` class helps assign integers to each character in the text. Insert a new cell and add the following code to implement this:

```
seq =char_tokenizer.texts_to_sequences(text)
seq
```

The code generates the following output:

```
[[4],
 [5],
 [2],
 [1],
 [9],
 [6],
 [10],
 [11],
 [12],
 [1],
 [13],
 [7],
 [3],
 [14],
 [15],
 [1],
 [16],
 [3],
 [17],
 [1],
 [18],
```

```
As you can see, there were **44** characters in the `text`
variable. From the output, we can see that for every unique
character in `text`, an integer is assigned.
```

7. Use `sequences_to_texts()` to get text from the sequence with the following code:

```
char_tokenizer.sequences_to_texts(seq)
```

The snippet of the preceding output follows:

```
['t',
 'h',
 'e',
 ' ',
 'q',
 'u',
 'i',
 'c',
 'k',
 ' ',
 'b',
 'r',
 'o',
 'w',
 'n',
 ' ',
 'f',
 'o',
 'x',
 ' ',
 'j',
 'u',
 'm',
 'p',
 'e',
 'd',
 ' ',
 'o',
```

8. Now look at the actual one-hot encoded values. For this, use the `texts_to_matrix()` function. Insert a new cell and add the following code to implement this:

```
char_vectors = char_tokenizer.texts_to_matrix(text)
```

Here, the results of the array are stored in the `char_vectors` variable.

9. In order to view the vector values, just insert a new cell and add the following line:

```
char_vectors
```

On execution, the code displays the array of one-hot encoded vectors:

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 1.]])
```

10. In order to investigate the dimensions of the NumPy array, make use of the `shape` attribute. Insert a new cell and add the following code to execute it:

    ```
    char_vectors.shape
    ```

    The following output is generated:

    ```
    (44, 27)
    ```

    So, `char_vectors` is a NumPy array with `44` rows and `27` columns. This is because we are considering 26 characters and an additional character for space.

11. To access the first row of `char_vectors` NumPy array, insert a new cell and add the following code:

    ```
    char_vectors[0]
    ```

    This returns a one-hot vector, which can be seen in the following figure:

    ```
    array([0 ., 0., 0., 0., 1., 0., 0., 0., 0 .,
           0., 0., 0., 0., 0., 0., 0.,0., 0 .,
           0., 0., 0., 0., 0., 0., 0 ., 0., 0])
    ```

12. To access the index of this one-hot vector, use the `argmax()` function provided by NumPy. Insert a new cell and write the following code to implement this:

    ```
    np.argmax(char_vectors[0])
    ```

    The code generates the following output:

    ```
    4
    ```

13. The `Tokenizer` class provides two dictionaries, `index_word` and `word_index`, which you can use to view the contents of `Tokenizer` in key-value form. Insert a new cell and add the following code to view the `index_word` dictionary:

    ```
    char_tokenizer.index_word
    ```

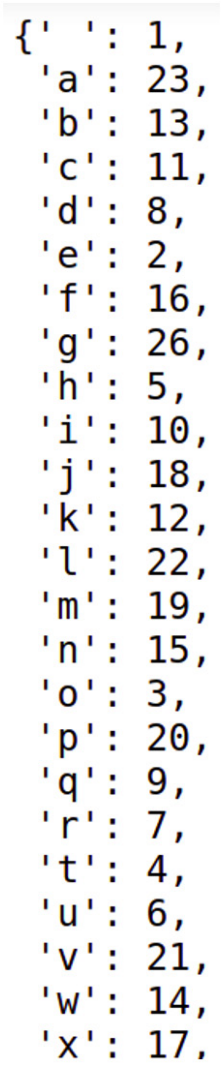
The code generates the following output:

```
{1: ' ',
 2: 'e',
 3: 'o',
 4: 't',
 5: 'h',
 6: 'u',
 7: 'r',
 8: 'd',
 9: 'q',
 10: 'i',
 11: 'c',
 12: 'k',
 13: 'b',
 14: 'w',
 15: 'n',
 16: 'f',
 17: 'x',
 18: 'j',
 19: 'm',
 20: 'p',
 21: 'v',
 22: 'l',
 23: 'a',
 24: 'z',
```

````
As you can see in this figure, the indices act as keys, and the
characters act as values. Now insert a new cell and the following
code to view the `word_index` dictionary:

```
char_tokenizer.word_index
```

The code generates the following output:
````




```
In this figure, the characters act as keys, and the indices act as
values.
```

14. In the preceding steps, you saw how to access the index of a given one-hot vector by using the `argmax()` function provided by NumPy. Using this index as a key, you can access its value in the `index_word` dictionary. To implement this, we insert a new cell and write the following code:

    ```
    char_tokenizer.index_word[np.argmax(char_vectors[0])]
    ```

    The preceding code generates the following output:

    ```
    't'
    ```

    In this code, `np.argmax(char_vectors[0])` produces an output of `4`. This will act as a key in finding the value in the `index_word` dictionary. So, when `char_tokenizer.index_word[4]` is executed, it will scan through the dictionary and find that, for key `4`, the value is `t`, and finally, it will print `t`.

Learned Word Embeddings

The vector representations discussed in the preceding section have some serious disadvantages, as discussed here:

- **Sparsity and large size**: The sizes of one-hot encoded or other frequency-based vectors depend upon the number of unique words in the corpus. This means that when the size of the corpus increases, the number of unique words increases, thereby increasing the size of the vectors in turn.
- **Context**: None of these vector representations consider the words with respect to its context while representing it as a vector. However, the meaning of a word in any language depends upon the context it is used in. Not taking the context into account can often lead to inaccurate results.

Word2Vec

`Word2Vec` is a prediction-based algorithm that represents a word by a vector of a fixed size. This is a form of unsupervised learning algorithm, which means that we need not to provide manually annotated data; we just feed the raw text. It will train a model in such a way that each word is represented in terms of its context throughout the training data.

This algorithm has two variations, as follows:

- **Continuous Bag of Words** (**CBoW**): This model tends to predict the probability of a word given the context. The learning problem here is to predict the word given a fixed-window context---that is, a fixed set of continuous words in text.
- **Skip-Gram model**: This model is the reverse of the CBoW model, as it tends to predict the context of a word.

Exercise 6.04: Training Word Vectors

In this exercise, we will train word vectors. We will be using books freely available on Project Gutenberg for this. We will also see the vector representation using Matplotlib's pyplot framework.

Follow these steps to implement this exercise:

1. Open a Jupyter notebook.
2. Use the `requests` library to load books from the Project Gutenberg website, the `json` library to load a book catalog, and the `regex` package to clean the text by removing newline characters. Insert a new cell and add the following code to implement this:

```
```
import requests
import json
import re
```
```

3. After importing all the necessary libraries, load the `json` file, which contains details of 10 books, including the title, the author, and the ID. Insert a new cell and add the following steps to implement this:

```
```
with open('../data/ProjectGutenbergBooks.json', 'r') \
    as catalog_file:
    catalog = json.load(catalog_file)
```
```

4. To print the details of all the books, insert a new cell and add the following code:

```
catalog
```

The preceding code generates the following output:

```
[{'author': 'Jane Austen', 'id': 1342, 'title': 'Pride and Prejudice'},
 {'author': 'Charles Dickens',
  'id': 46,
  'title': 'A Christmas Carol in Prose'},
 {'author': 'Charles Dickens', 'id': 98, 'title': 'A Tale of Two Cities'},
 {'author': 'Mary Wollstonecraft Shelley',
  'id': 84,
```

5. Create a function named `load_book()`, which will take `book_id` as a parameter and, based on that `book_id`, fetch the book and load it. It should also clean the text by removing the newline characters. Insert a new cell and add the following code to implement this:

```
```
GUTENBERG_URL ='https://www.gutenberg.org/files/{}/{}-0.txt'
def load_book(book_id):
    url = GUTENBERG_URL.format(book_id, book_id)
    contents = requests.get(url).text
    cleaned_contents = re.sub(r'\r\n', ' ', contents)
    return cleaned_contents
```
```

6. Once you have defined our `load_book()` function, you will loop through the catalog, fetch all the `id` instances of the books, and store them in the `book_ids` list. The `id` instances stored in the `book_ids` list will act as parameters for our `load_book()` function. The book information fetched for each book ID will be loaded in the `books` variable. Insert a new cell and add the following code to implement this:

```
book_ids = [ book['id'] for book in catalog ]
books = [ load_book(id) for id in book_ids]
```

To view the information of the `books` variable, add the following code in a new cell:

```
books[:5]
```

A snippet of the output generated by the preceding code is as follows:

```
["ï»¿ The Project Gutenberg EBook of Pride and Prejudice, by Jane Austen  This eBook is f
or the use of anyone anywhere at no cost and with almost no restrictions whatsoever.  You
may copy it, give it away or re-use it under the terms of the Project Gutenberg License i
ncluded with this eBook or online at www.gutenberg.org   Title: Pride and Prejudice  Auth
or: Jane Austen  Release Date: August 26, 2008 [EBook #1342] Last Updated: November 12, 2
019   Language: English  Character set encoding: UTF-8  *** START OF THIS PROJECT GUTENBE
RG EBOOK PRIDE AND PREJUDICE ***     Produced by Anonymous Volunteers, and David Widger
THERE IS AN ILLUSTRATED EDITION OF THIS TITLE WHICH MAY VIEWED AT EBOOK [# 42671 ]  cover
Pride and Prejudice        By Jane Austen          CONTENTS           Chapter 1
Chapter 2           Chapter 3           Chapter 4           Chapter 5           Chapter 6
Chapter 7           Chapter 8           Chapter 9           Chapter 10           Chapter
11          Chapter 12          Chapter 13          Chapter 14          Chapter 15
Chapter 16          Chapter 17          Chapter 18          Chapter 19          Chapt
er 20          Chapter 21          Chapter 22          Chapter 23          Chapter 24
Chapter 25          Chapter 26          Chapter 27          Chapter 28          Chapt
er 29          Chapter 30          Chapter 31          Chapter 32          Chapter 33
Chapter 34          Chapter 35          Chapter 36          Chapter 37          Chapt
er 38          Chapter 39          Chapter 40          Chapter 41          Chapter 42
Chapter 43          Chapter 44          Chapter 45          Chapter 46          Chapt
```

7. Before you can train the word vectors, you need to split the books into a list of documents. In this case, you want to teach the `Word2Vec` algorithm about words in the context of the sentences that they are in. So here, a document is actually a sentence. Thus, you need to create a list of sentences from all 10 books. Insert a new cell and add the following code to implement this:

```
from gensim.summarization import textcleaner
from gensim.utils import simple_preprocess
def to_sentences(book):
    sentences = textcleaner.split_sentences(book)
    sentence_tokens = [simple_preprocess(sentence) \
                       for sentence in sentences]
    return sentence_tokens
```

   In the preceding code, all the text preprocessing takes place inside the `to_sentences()` function that you have defined.

8. Now, loop through each book in `books` and pass each book as a parameter to the `to_sentences()` function. The results should be stored in the `book_sentences` variable. Also, split books into sentences and sentences into documents. The result should be stored in the `documents` variable. Insert a new cell and add the following code to implement this:

```
```
books_sentences = [to_sentences(book) for book in books]
documents = [sentence for book_sent in books_sentences \
             for sentence in book_sent]
```
```

9. To check the length of the documents, use the `len()` function as follows:

```
len(documents)
```

   The code generates the following output:

```
32922
```

10. Now that you have your documents, train the model by making use of the `Word2Vec` class provided by the gensim package. Insert a new cell and add the following code to implement this:

```
from gensim.models import Word2Vec
# build vocabulary and train model
model = Word2Vec(
        documents,
        size=100,
        window=10,
        min_count=2,
        workers=10)
model.train(documents, total_examples=len(documents), \
            epochs=50)
```

The code generates the following output:

```
(27809439, 37551450)
```

Now make use of the `most_similar()` function of the `model.wv` instance to find the similar words. The `most_similar()` function takes `positive` as a parameter and returns a list of strings that contribute positively. Insert a new cell and add the following code to implement this:

```
model.wv.most_similar(positive="worse")
```

The code generates the following output:

```
[('kinder', 0.6370856761932373),
 ('older', 0.616365909576416),
 ('more', 0.616054892539978),
 ('narrower', 0.6126840710639954),
 ('better', 0.5983243584632874),
 ('larger', 0.5939739346504211),
 ('stronger', 0.5806257128715515),
 ('happier', 0.5734115839004517),
 ('less', 0.5692222714424133),
 ('handsomer', 0.5489497184753418)]
```

```
Note:

You may get a slightly different output as the output depends on the
model training process, so you may have a different model than the
one we have trained here.
```

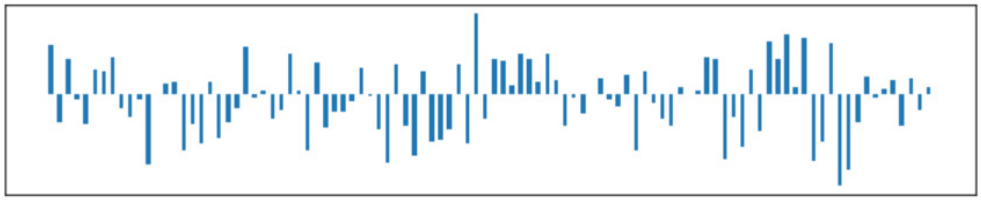
11. Create a `show_vector()` function that will display the vector using `pyplot`, a plotting framework in Matplotlib. Insert a new cell and add the following code to implement this:

```
%matplotlib inline
import matplotlib.pyplot as plt
def show_vector(word):
    vector = model.wv[word]
    fig, ax = plt.subplots(1,1, figsize=(10, 2))
```

```
    ax.tick_params(axis='both', \
                   which='both',\
                   left=False, \
                   bottom=False, \
                   top=False,\
                   labelleft=False, \
                   labelbottom=False)
    ax.grid(False)
    print(word)
    ax.bar(range(len(vector)), vector, 0.5)
show_vector('sad')
```

The code generates the following output:

```
sad
```



Note

In the preceding figure, we can see the vector representation when the word provided to the `show_vector()` function is "sad". We have learned about training word vectors and representing them using `pyplot`. In the next section, we will focus more on using **pre-trained word vectors**, which are required for NLP projects.

Using Pre-Trained Word Vectors

`Word2Vec` has recently proved to be state-of-the-art for tasks including checking for word analogies and word similarities, as follows:

- *vector('Paris') - vector('France') + vector('Italy')* results in a vector that is very close to *vector('Rome')*.
- *vector('king') - vector('man') + vector('woman')* is close to *vector('queen')*.

To better understand how we can use pre-trained word vectors in Python, let's walk through a simple exercise.

Exercise 6.05: Using Pre-Trained Word Vectors

In this exercise, we will load and use pre-trained word embeddings. We will also show the image representation of a few word vectors using the pyplot framework of the Matplotlib library. We will be using `glove6B50d.txt`, which is a pre-trained model.

Note

The pre-trained model being used for this file can be found at https://www.kaggle.com/watts2/glove6b50dtxt/download. Download this file and place it in the `data` folder of *Lab 6, Vector Representation*.

Follow these steps to complete this exercise:

1. Open a Jupyter notebook.

2. Add the following statement to import the `numpy` library:

```
import numpy as np
import zipfile
```

3. Move the downloaded model from the preceding link to the location given in the following code snippet. In order to extract data from a ZIP file, use the `zipfile` Python package. Add the following code to unzip the embeddings from the ZIP file:

```
GLOVE_DIR = '../data/'
GLOVE_ZIP = GLOVE_DIR + 'glove6B50d.txt.zip'
print(GLOVE_ZIP)
zip_ref = zipfile.ZipFile(GLOVE_ZIP, 'r')
zip_ref.extractall(GLOVE_DIR)
zip_ref.close()
```

4. Define a function named `load_glove_vectors()` to return a model Python dictionary. Insert a new cell and add the following code to implement this:

```
def load_glove_vectors(fn):
    print("Loading Glove Model")
    with open( fn,'r', encoding='utf8') as glove_vector_file:
        model = {}
        for line in glove_vector_file:
            parts = line.split()
            word = parts[0]
            embedding = np.array([float(val) \
                        for val in parts[1:]])
            model[word] = embedding
        print("Loaded {} words".format(len(model)))
    return model
glove_vectors = load_glove_vectors(GLOVE_DIR +'glove6B50d.txt')
```

Here, `glove_vector_file` is a text file containing a dictionary. In this, words act as keys and vectors act as values. So, we need to read the file line by line, split it, and then map it to a Python dictionary. The preceding code generates the following output:

```
Loading Glove Model
Loaded 400000 words
```

If we want to view the values of `glove_vectors`, then we insert a new cell and add the following code:

```
glove_vectors
```

You will get the following output:

```
{'players': array([-9.1399e-01,  8.4152e-02,  6.4889e-02,  3.8138e-01,  1.0120e-01,
        -4.1253e-01, -1.0374e+00,  5.8694e-01, -1.0811e+00,  4.1565e-01,
         7.5995e-01,  5.8594e-01, -1.0692e+00,  2.8048e-01,  1.0978e+00,
        -2.2174e-02,  5.0837e-01, -1.1568e-02, -7.8902e-01, -1.2340e+00,
        -1.1572e+00,  3.1983e-01,  4.2662e-01,  5.2228e-01, -2.8263e-01,
        -1.1629e+00,  3.8899e-01, -5.7561e-01, -3.0536e-01, -1.0698e+00,
         3.4031e+00,  1.2970e+00,  4.0442e-01, -5.0792e-01,  8.6177e-01,
         7.7060e-01,  4.8023e-01,  4.9316e-01, -4.2102e-01, -8.6115e-01,
        -1.3608e-02,  7.4204e-02,  3.6231e-02,  1.1018e+00,  1.3154e-01,
         2.0627e-01, -2.9658e-03,  6.5953e-01, -7.2998e-01, -1.9931e-01]),
 '1.3277': array([-3.2147e-01, -7.2671e-01,  1.9024e-01, -9.0861e-01, -1.4139e+00,
        -7.5414e-01, -1.0425e+00,  1.2735e-01, -5.3525e-01,  6.2581e-01,
        -9.0084e-01, -5.6066e-01,  1.4616e+00,  2.9404e-01,  6.9728e-01,
        -6.9704e-01, -2.4583e-01, -7.3691e-01, -1.0720e-03,  1.7516e+00,
         1.5047e+00, -3.2935e-02, -5.5113e-01,  1.7669e-01, -1.1550e-01,
         6.8808e-01,  1.2447e+00, -1.2322e+00, -7.0814e-01,  1.5486e-02,
        -1.1982e+00,  3.3936e-01,  2.7709e-01, -5.5773e-01, -7.4283e-01,
         1.0060e+00, -3.6225e-01,  2.9963e-01, -7.9166e-01, -6.2066e-01,
        -1.3871e-01, -1.2618e-01,  8.4840e-01,  2.9157e-01, -8.4433e-01,
         6.1656e-01, -3.3926e-01,  1.1435e-01, -1.3091e-01, -1.3911e+00]),
```

The order of the result dictionary can vary as it is a Python dict.

5. The `glove_vectors` object is basically a dictionary containing the mappings of the words to the vectors, so you can access the vector for a word, which will return a 50-dimensional vector. Insert a new cell and add the code to check the vector for the word `dog`:

```
glove_vectors["dog"]
```

```
array([ 0.11008  , -0.38781  , -0.57615  , -0.27714  ,  0.70521  ,
        0.53994  , -1.0786   , -0.40146  ,  1.1504   , -0.5678   ,
        0.0038977,  0.52878  ,  0.64561  ,  0.47262  ,  0.48549  ,
       -0.18407  ,  0.1801   ,  0.91397  , -1.1979   , -0.5778   ,
       -0.37985  ,  0.33606  ,  0.772    ,  0.75555  ,  0.45506  ,
       -1.7671   , -1.0503   ,  0.42566  ,  0.41893  , -0.68327  ,
        1.5673   ,  0.27685  , -0.61708  ,  0.64638  , -0.076996 ,
        0.37118  ,  0.1308   , -0.45137  ,  0.25398  , -0.74392  ,
       -0.086199 ,  0.24068  , -0.64819  ,  0.83549  ,  1.2502   ,
       -0.51379  ,  0.04224  , -0.88118  ,  0.7158   ,  0.38519  ])
```

In order to see the vector for the word `cat`, add the following code:

```
glove_vectors["cat"]
```

```
array([ 0.45281 , -0.50108 , -0.53714 , -0.015697,  0.22191 ,  0.54602
,
       -0.67301 , -0.6891  ,  0.63493 , -0.19726 ,  0.33685 ,  0.7735
,
        0.90094 ,  0.38488 ,  0.38367 ,  0.2657  , -0.08057 ,  0.61089
,
       -1.2894  , -0.22313 , -0.61578 ,  0.21697 ,  0.35614 ,  0.44499
,
        0.60885 , -1.1633  , -1.1579  ,  0.36118 ,  0.10466 , -0.78325
,
        1.4352  ,  0.18629 , -0.26112 ,  0.83275 , -0.23123 ,  0.32481
,
        0.14485 , -0.44552 ,  0.33497 , -0.95946 , -0.097479,  0.48138
,
       -0.43352 ,  0.69455 ,  0.91043 , -0.28173 ,  0.41637 , -1.2609
,
        0.71278 ,  0.23782 ])
```

6. Now that you have the vectors, represent them as an image using the pyplot framework of the Matplotlib library. Insert a new cell and add the following code to implement this:

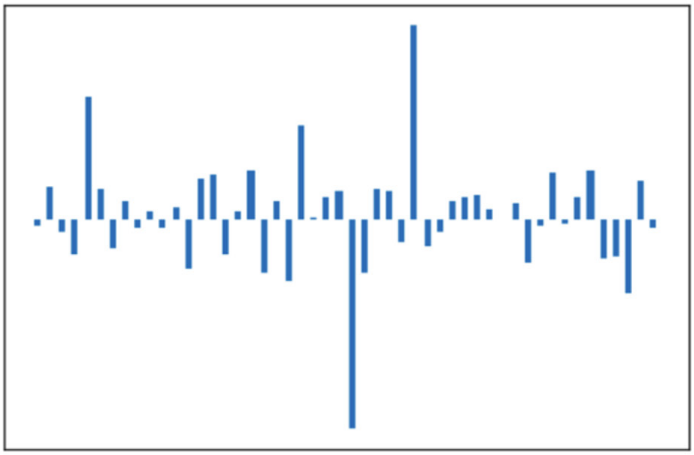
```
%matplotlib inline
import matplotlib.pyplot as plt
def to_vector(glove_vectors, word):
    vector = glove_vectors.get(word.lower())
    if vector is None:
        vector = [0] * 50
    return vector
def to_image(vector, word=''):
    fig, ax = plt.subplots(1,1)
    ax.tick_params(axis='both', which='both',\
                   left=False, \
                   bottom=False, \
                   top=False,\
                   labelleft=False,\
                   labelbottom=False)
    ax.grid(False)
    ax.bar(range(len(vector)), vector, 0.5)
    ax.text(s=word, x=1, y=vector.max()+0.5)
    return vector
```

   In the preceding code, you defined two functions. The `to_vector()` function accepts `glove_vectors` and `word` as parameters. Here, the `get()` function of `glove_vectors` will find the word and convert it into lowercase. The result will be stored in the `vector` variable.

7. The `to_image()` function takes `vector` and `word` as input and shows the image representation of `vector`. To find the image representation of the word `man`, type the following code:

```
man = to_image(to_vector(glove_vectors, "man"))
```

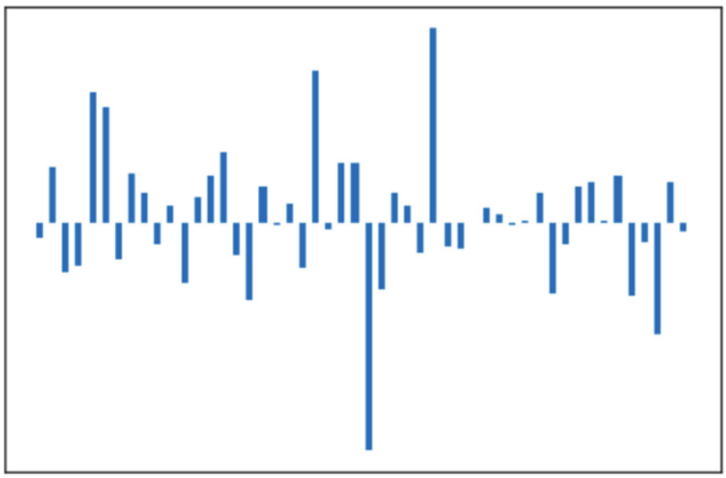
The code generates the following output:



8. To find the image representation of the word `woman`, type the following code:

```
woman = to_image(to_vector(glove_vectors, "woman"))
```

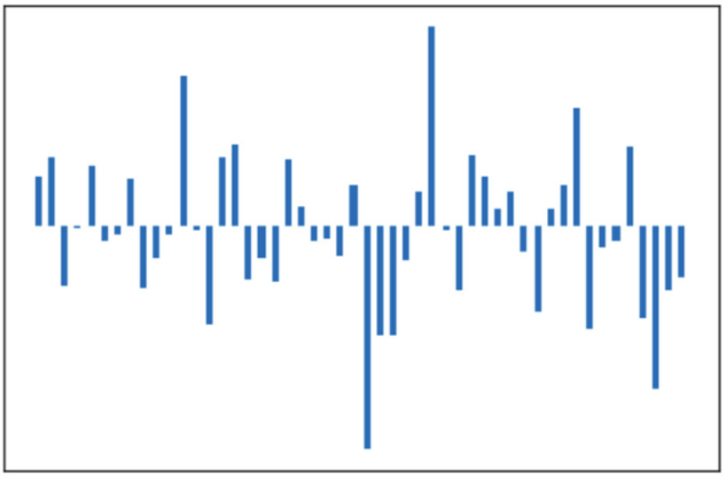
This will generate the following output:



9. To find the image representation of the word `king`, type the following code:

```
king = to_image(to_vector(glove_vectors, "king"))
```

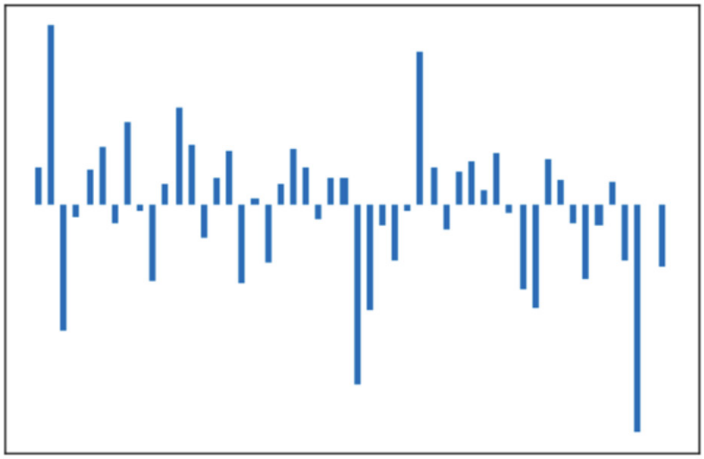
This will generate the following output:



10. To find the image representation of the word `queen`, type the following code:

```
queen = to_image(to_vector(glove_vectors, "queen"))
```

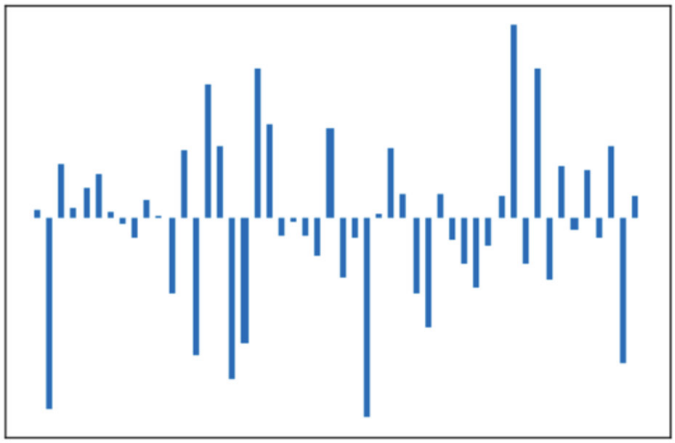
This will generate the following output:



11. To find the image representation of the vector for `king - man + woman - queen`, type the following code:

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

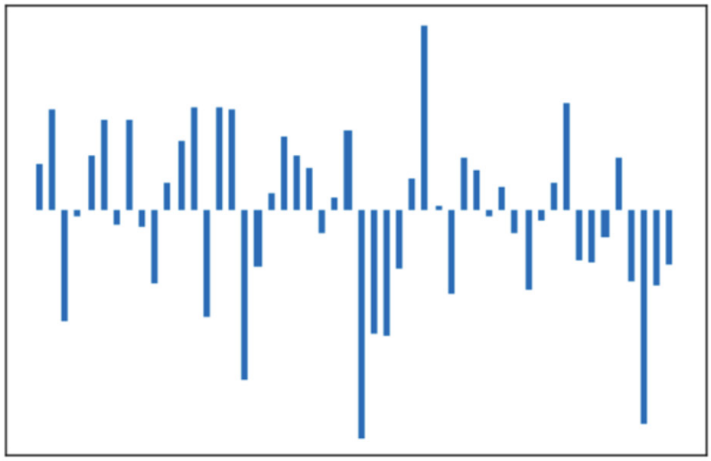
This will generate the following output:



12. To find the image representation of the vector for `king - man + woman`, type the following code:

```
nd = to_image(king - man + woman)
```

This will generate the following output:



Note

This section does not currently have an online interactive example, and will need to be run locally.

The preceding results are the visual proof of the example we already discussed. We've learned how to load and use pre-trained word vectors and view their image representations. In the next section, we will focus on document vectors and their uses.

Document Vectors

Word vectors and word embeddings represent words. But if we wanted to represent a whole document, we'd need to use document vectors. **Note** that when we refer to a document, we are referring to a collection of words that have some meaning to a user. A document can be a single sentence or a group of sentences. A document can consist of product reviews, tweets, or lines of movie dialogue, and can be from a few words to thousands of words. A

document can be used in a machine learning project as an instance of something that the algorithm can learn from. We can represent a document with different techniques:

- Calculating the mean value: We calculate the mean of all the constituent word vectors of a document and represent the document by the mean vector.
- `Doc2Vec`: `Doc2Vec` is a technique by which we represent documents by a fixed-length vector. It is trained quite similarly to the way we train the `Word2Vec` model. Here, we also add the unique ID of the document to which the word belongs. Then, we can get the vector of the document from the trained model using the document ID.

Similar to `Word2Vec`, the `Doc2Vec` class contains parameters such as `min_count`, `window`, `vector_size`, `sample`, `negative`, and `workers`. The `min_count` parameter ignores all the words with a frequency less than that specified. The `window` parameter sets the maximum distance between the current and predicted words in the given sentence. The `vector_size` parameter sets the dimensions of each vector.

The `sample` parameter defines the threshold that allows us to configure the higher-frequency words that are regularly down-sampled, while `negative` specifies the total amount of noise words that should be drawn and `workers` specifies the total number of threads required to train the model. To build the vocabulary from the sequence of sentences, `Doc2Vec` provides the `build_vocab` method. We'll be using all of these in the upcoming exercise.

Uses of Document Vectors

Some of the uses of document vectors are as follows:

- **Similarity**: We can use document vectors to compare texts for similarity. For example, legal AI software can use document vectors to find similar legal cases.
- **Recommendations**: For example, online magazines can recommend similar articles based on those that users have already read.
- **Predictions**: Document vectors can be used as input into machine learning algorithms to build predictive models.

In the next section, we will perform an exercise based on document vectors.

Exercise 6.06: Converting News Headlines to Document Vectors

In this exercise, we will convert some news headlines into document vectors. Also, we will look at the image representation of the vector. Again, for image representation, we will be using the pyplot framework of the Matplotlib library. Follow these steps to complete this exercise:

1. Open a Jupyter notebook.
2. Import all the necessary libraries for this exercise. You will be using the gensim library. Insert a new cell and add the following code:

```
import pandas as pd
from gensim import utils
from gensim.models.doc2vec import TaggedDocument
from gensim.models import Doc2Vec
from gensim.parsing.preprocessing \
import preprocess_string, remove_stopwords
import random
import warnings
warnings.filterwarnings("ignore")
```

In the preceding code snippet, other than other imports, you imported `TaggedDocument` from gensim, which prepares the document formats used in `Doc2Vec`. It represents the document along with the tag. This will be clearer from the following code lines. `Doc2Vec` requires each instance to be a `TaggedDocument` instance.

3. Move the downloaded file to the following location and create a variable of the path as follows:

```
sample_news_data = '../data/sample_news_data.txt'
```

4. Now load the file:

```
with open(sample_news_data, encoding="utf8", \
          errors='ignore') as f:
    news_lines = [line for line in f.readlines()]
```

5. Now create a DataFrame out of the headlines as follows:

```
lines_df = pd.DataFrame()
indices  = list(range(len(news_lines)))
lines_df['news'] = news_lines
lines_df['index'] = indices
```

6. View the head of the DataFrame using the following code:

```
lines_df.head()
```

This will create the following output:

	news	index
0	Top of the Pops leaves BBC One The BBC flagshi...	0
1	Oscars race enters final furlong The race for ...	1
2	US TV special for tsunami relief A US televisi...	2
3	Williamson lauds bowlers for adapting to atypi...	3
4	Housewives lift Channel ratings The debut of U...	4

7. Create a class, the object of which will create the training instances for the `Doc2Vec` model. Insert a new cell and add the following code to implement this:

```
class DocumentDataset(object):

    def __init__(self, data:pd.DataFrame, column):
        document = data[column].apply(self.preprocess)
        self.documents = [ TaggedDocument( text, [index]) \
                           for index, text in \
                           document.iteritems() ]

    def preprocess(self, document):
        return preprocess_string(\
            remove_stopwords(document))

    def __iter__(self):
        for document in self.documents:
            yield documents

    def tagged_documents(self, shuffle=False):
        if shuffle:
            random.shuffle(self.documents)
        return self.documents
```

In the code, the `preprocess_string()` function applies the given filters to the input. As its name suggests, the `remove_stopwords()` function is used to remove `stopwords` from the given document. Since `Doc2Vec` requires each instance to be a `TaggedDocument` instance, we create a list of `TaggedDocument` instances for each headline in the file.

8. Create an object of the `DocumentDataset` class. It takes two parameters. One is the `lines_df_small` DataFrame and the other is the `Line` column name. Insert a new cell and add the following code to implement this:

```
```
documents_dataset = DocumentDataset(lines_df, 'news')
```
```

9. Create a `Doc2Vec` model using the `Doc2Vec` class. Insert a new cell and add the following code to implement this:

```
```
docVecModel = Doc2Vec(min_count=1, window=5, vector_size=100, \
                      sample=1e-4, negative=5, workers=8)
docVecModel.build_vocab(documents_dataset.tagged_documents())
```
```

10. Now you need to train the model using the `train()` function of the `Doc2Vec` class. This could take a while, depending on how many records we train. Here, `epochs` represents the total number of records required to train the document. Insert a new cell and add the following code to implement this:

```
```
docVecModel.train(documents_dataset.\
                  tagged_documents(shuffle=True),\
                  total_examples = docVecModel.corpus_count,\
```
```

```
                  epochs=10)
```

11. Save this model for future use as follows:

```
docVecModel.save('../data/docVecModel.d2v')
```

12. The model has been trained. To verify this, access one of the vectors with its index. To do this, insert a new cell and add the following code to find the `doc` vector of index `657`:

    ```
    docVecModel[657]
    ```

    You should get an output similar to the one below:

    ```
    array([-0.20929255, -0.28592077,  0.07170248, -0.08378136,  0.08677434,
           -0.08534106, -0.10685636,  0.00576654, -0.03399038,  0.31707773,
           -0.01177737, -0.01255067,  0.08325162, -0.00340701,  0.18105389,
            0.21334894, -0.06321663,  0.12173653,  0.06913093,  0.26549008,
           -0.13100868, -0.18686569,  0.15937229, -0.2514969 , -0.04796949,
           -0.16842327,  0.00436876, -0.33365294, -0.28871712,  0.03756697,
            0.04344342,  0.00543962, -0.04138061,  0.18812971, -0.09889945,
            0.04105975, -0.10300889,  0.23657063,  0.13254939, -0.00466744,
            0.11880156,  0.5763065 , -0.13349594,  0.0199004 , -0.20860918,
            0.25577313,  0.19562072, -0.07257853, -0.06559091,  0.10523199,
            0.02009419,  0.16713172, -0.01012599,  0.1330115 ,  0.01896849,
            0.17950307, -0.17225678,  0.12428728,  0.29377568, -0.07040878,
           -0.02737776, -0.08043538,  0.14898372, -0.13726163,  0.19797257,
            0.21860234, -0.09122779, -0.20461401,  0.11374234,  0.26827952,
            0.20443173, -0.06511806, -0.08415387,  0.03433008,  0.12155177,
           -0.2339348 ,  0.13126448,  0.25857013,  0.1554307 ,  0.07587516,
            0.07074967,  0.20744652, -0.1130043 , -0.3087442 , -0.13767944,
            0.02682412, -0.07112097,  0.15238564, -0.05019746, -0.12936442,
            0.12461095, -0.04515083, -0.11713583, -0.11419832, -0.14024146,
           -0.00401924,  0.04734296,  0.03295911, -0.11777124,  0.04519369],
          dtype=float32)
    ```

13. To check the image representation of any given vector, make use of the pyplot framework of the Matplotlib library. The `show_news_lines()` function takes a line number as a parameter. Based on this line number, find the vector and store it in the `doc_vector` variable. The `show_image()` function takes two parameters, `vector` and `line`, and displays an image representation of the vector. Insert a new cell and add the following code to implement this:

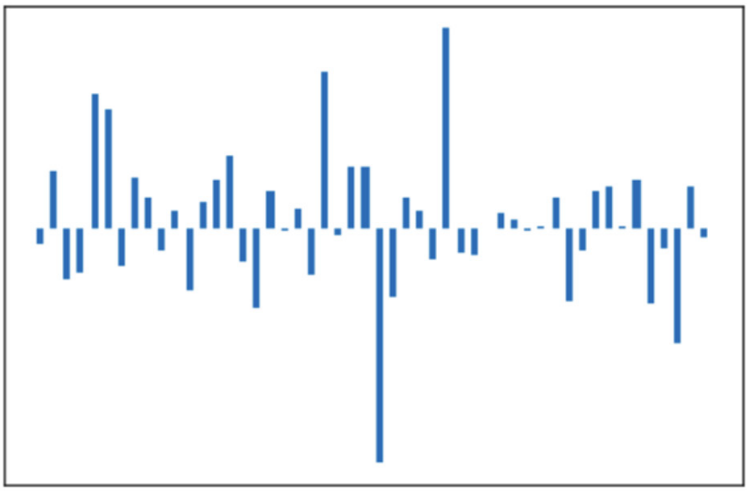
```
import matplotlib.pyplot as plt
def show_image(vector, line):
    fig, ax = plt.subplots(1,1, figsize=(10, 2))
    ax.tick_params(axis='both', \
                   which='both',\
                   left=False, \
                   bottom=False,\
                   top=False,\
                   labelleft=False,\
                   labelbottom=False)
```

```
    ax.grid(False)
    print(line)
    ax.bar(range(len(vector)), vector, 0.5)
def show_news_lines(line_number):
    line = lines_df[lines_df.index==line_number].news
    doc_vector = docVecModel[line_number]
    show_image(doc_vector, line)
```

14. Now that you have defined the functions, implement the `show_news_lines()` function to view the image representation of the vector. Insert a new cell and add the following code to implement this:

    ```
    show_news_lines(872)
    ```

    The code generates the following output:



Activity 6.01: Finding Similar News Article Using Document Vectors

To complete this activity, you need to build a news search engine that finds similar news articles like the one provided as input using the `Doc2Vec` model. You will find headlines similar to "US raise TV indecency US politicians are proposing a tough new law aimed at cracking down on indecency." Follow these steps to complete this activity:

1. Open a Jupyter notebook and import the necessary libraries.
2. Load the new article lines file.
3. Iterate over each headline and split the columns and create a DataFrame.
4. Load the `Doc2Vec` model that you created in the previous exercise.
5. Create a function that converts the sentences into vectors and another that does the similarity checks.
6. Test both the functions.

   Note: The full solution to this activity in the current directory.

So, in this activity, we were able to find similar news headlines with the help of document vectors. A common use case of inferring text similarity from document vectors is in text paraphrasing, which we'll explore in detail in the next

lab.

Summary

In this lab, we learned about the motivations behind converting human language in the form of text into vectors. This helps machine learning algorithms to execute mathematical functions on the text, detect patterns in language, and gain an understanding of the meaning of the text. We also saw different types of vector representation techniques, such as character-level encoding and one-hot encoding.

In the next lab, we will look at the areas of text paraphrasing, summarization, and generation. We will see how we can automate the process of text summarization using the NLP techniques we have learned so far.