7

Vector Representations

Present 1-3: The Title slide and the Learning Objectives slide. An overview of what we will achieve in this course.

Lesson Objectives

By the end of this lesson, you will be able to:

* Learn why text needs to be represented as vectors for natural language processing
* Understand the various ways that text can be represented as vectors
* Understand Word Vectors and the various forms
* Learn how to create document vectors
* Learn how to perform vector arithmetic i.e. King – Man + Woman = Queen

Introduction

Announce 4: Welcome the class. Introduce yourself and discuss what the lesson will cover. Talk about the topics that will be covered in this lesson.

The previous lessons gave you a very good grounding on natural language processing but now we will go deeper into a key topic – one that also give us surprising insights into how language works and how some of the key advances in human computer interaction is facilitated.

At the heart of natural language processing is the simple trick. If you can represent text as numbers, then software algorithms can perform the sophisticated computation require to understand the meaning of the text.

Text representation can be as simple as encoding each word as an integer number but it can also include using an array of numbers for each word. Each of these representations help machine learning programs to function.

Some of the techniques we will learn are – at the time of speaking – close to cutting edge. A lot of the recent advances in natural language come from representing text as vectors. Word2Vec for example has been used by Google as the main text representations in some of their deep learning models – and the models support their consumer products that use natural language processing such as Google Echo.

This lesson begins by trying to understand how text can be represented as vectors, what vectors are, and how they can be composed in such a way as to represent really complex speech. We will also work through the various representations in both directions – learning how to encode text as vectors and how to retrieve the text.

Why Vector Representations

Present 5: Slide introducing the topic of why we represent text as numbers and specifically as vectors

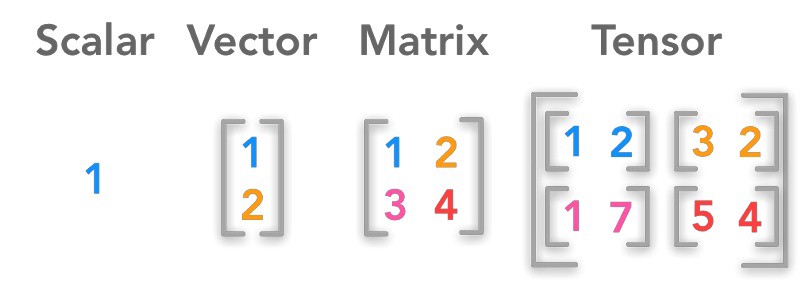
Vector Definitions

Present 5 and 6: Slide introducing the new topic

To understand how to represent text as numbers we first have to understand these concepts – scalars, vectors and matrixes. The simplest way to understand these are as follows

1. A scalar is a number like 3, 5 or 168
2. A vector is a row or list of numbers
3. A matrix is a rectangular array of numbers – one or more rows, one or more columns. You can also think of a vector as a row in a matrix.

Figure showing the difference between scalars, vectors, matrices and tensors



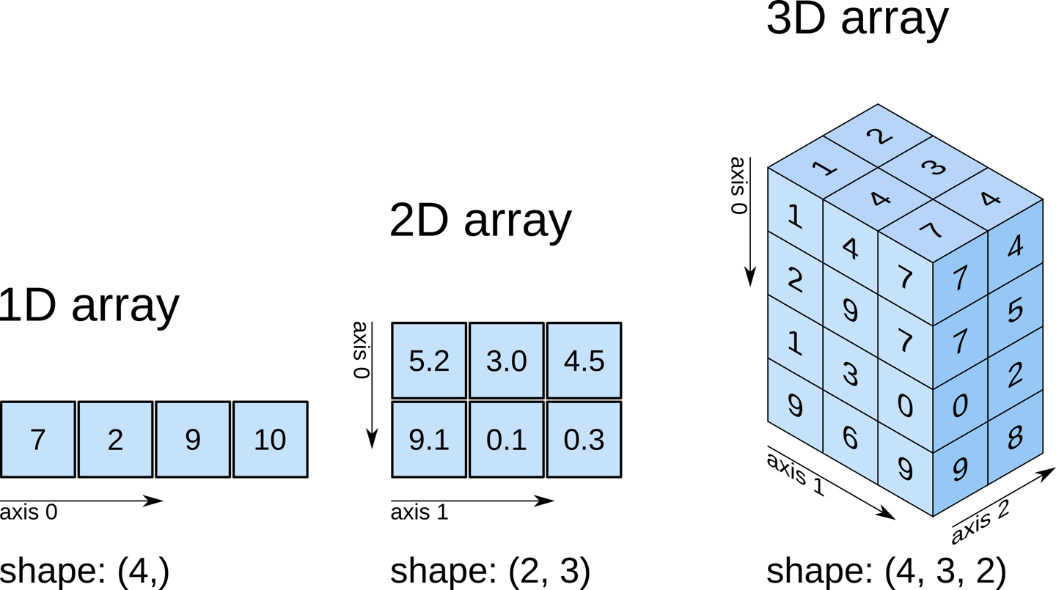
Now try to think about text

1. A letter is a single unit of an alphabet
2. A word is a list of letters
3. A paragraph is a collection of words. Note that as document is also a collection of words possibly broken down into paragraphs.

This illustrates that there can be a representation of text as numbers since there is a for each unit of text a similar unit of numbers that ca be used to represent the text. There are a few existing numeric representations of text used in computing – most notably ASCII and Unicode – but those are general purpose. In machine learning, the representations are centered around the use of vectors and matrices. The operations are based on performing calculations on the data using linear algebra.

Note: Throughout this lesson we will be using the **numpy** library. **Numpy** uses what it calls arrays, which is a structure that can represent scalars, vectors and matrices. We will use the term array interchangeably with vectors and matrices

Figure showing 1, 2 and 3D numpy arrays



Discuss 6: Discuss ways that text can be represented as numbers and based around the Unicode value for the student’s language

The distinction between vectors and matrices should be emphasized. You can think of a vector as a row or column of a matrix in the same way as a sentence is a row in a document.

Thus, if you needed to read a text document into a machine learning program you could think about creating an array of from the lines in the document.

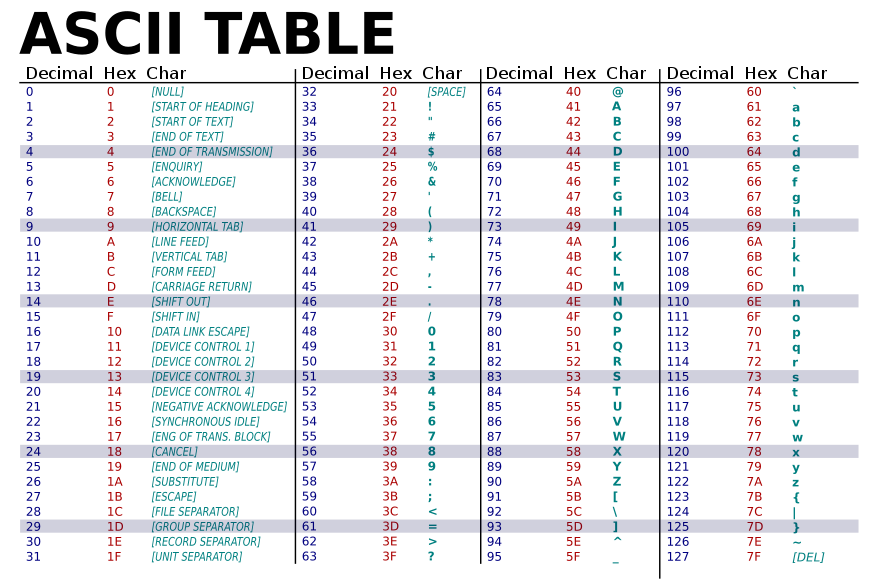
To see examples of vectors, see Vector Examples on WolframAlpha <https://www.wolframalpha.com/examples/mathematics/linear-algebra/vectors/>

Announce 7: The library Tensorflow was named after tensors – which are a generalization of scalars, vectors and matrices

Character Level Encoding

At the lowest level text is simply a series of characters. Characters usually start with the letters of the alphabet plus punctuation characters plus other lesser used characters. When it relates to natural language processing the characters that are encoded are the ones that contain the linguistic meaning of the text and it is common practice in preprocessing to remove characters that do not assist the NLP program to gain the meaning of the text. There are exceptions though where the punctuation marks, spaces, exclamations etc. give additional hints as to the meaning of the text and so it would be a judgement call by the data scientist whether to include and encode these characters.

The Roman alphabet is the most widely used alphabet and is the one on which the Ascii encoding is based. So, one simple way to do character level encoding is to simply use the ASCII code for that character.



The illustration above shows the ASCII charset.

Note

All exercises and activities will be primarily developed in Jupyter Notebook. There should be separate notebooks for most exercises, but there will be some exercise that will be a continuation of a previous exercise and so will share the same notebook. The instructions for each exercise will state the notebook that will be used for that exercise.

The code in the exercises may also depend on pre-installed libraries such as Tensorflow or Keras. So, for the code run make sure that the conda environment for the exercises is running and the notebook is launched into that environment. The following command should do this

**General Instructions for Launching Notebooks**

1. Start 🡪 Run 🡪 Type “Anaconda Prompt”
2. Select Anaconda Prompt. In the console enter the following command
3. conda activate packt
4. jupyter notebook

Activity 1: Character Encoding Using Ascii Values

In this activity we will try a very simple method of encoding text characters. Basically, we will write code that will encode text characters using their ascii values.

Python has a function called ord() which returns the ascii code of a provided character.

1. Follow the instructions above to open Anaconda prompt, activate the packt environment, and navigate to the **notebooks** directory.
2. Run the command jupyter notebook
3. In the directory listing select the **CharacterLevelEncoding.iynb** notebook
4. You can take a look at what the ord function returns. Insert a new cell and enter the following code. For example, you can enter the following

ord('A'), ord('a'), ord(',')

1. Now we are going to create a function that will take a text string and return a vector of the ascii characters for that text. For example, if given a string ‘quick brown fox’ the function should return a python list of length 15 containing the ascii value of each character.

Loop through each character and add its ascii value to the list

def to\_vector\_ascii(text):

return [ord(a) for a in text]

1. Now test the function using a phrase

to\_vector\_ascii('quick brown fox')

You can try other phrases

1. Our program outputs a list of the ascii characters in supplied phrases. Now we want to change this from a list to a numpy array. At the top of the cell enter

import numpy

1. Modify the to\_vector\_ascii function to

def to\_vector\_ascii(text):

return numpy.array([ord(a) for a in text])

Test your function

1. Modify the function again to

def to\_vector\_ascii(text):

return numpy.array([ord(a) for a in text]).reshape(1,1)

1. Now test the function by providing a test sentence. Do it in two lines – by first assigning to a variable, then printing

my\_vector = to\_vector(“The world is round”)

my\_vector

1. Now check the shape of the vector using the **numpy** array shape function

Discuss 6: Explain to the class that we used numpy arrays instead of because most machine learning algorithms operate on numpy arrays. Also explain that reshape converts the array into a shape that some algorithms e.t.c. those used in scikit-learn - prefer

Positional Character Level Encoding

Our function works to convert text into a vector of numbers but has an obvious limitation. Because it is based on ascii there are some characters that cannot be encoded using it and it is also incapable to encoding text in languages that have characters that fall outside of the range of the Ascii charset.

An improvement over ASCII encoding of text is to encode it by position – independent of a character set. One way would be to give each character a positional integer value based on the position of when that character is first encountered. For example, in the phrase ‘*sneeze epidemic*’ the letter *s* would be 0, *n* would be 1, *e* would be 2 and so on. This means we need to keep of when the character is first encountered. Repeated letters get the same integer value as it would when it was first encountered so the letter e is always encoded as 2

**Positional Encoding a phrase**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **s** | **n** | **e** | **e** | **z** | **e** |  | **e** | **p** | **i** | **d** | **e** | **m** | **i** | **c** |
| 0 | 1 | 2 | 2 | 3 | 2 | 4 | 2 | 5 | 6 | 7 | 2 | 8 | 6 | 9 |

Exercise 2: Character Level Encoding using Positions

In this exercise we will implement character encoding using positions.

The objective is, given a string we return a dict that contains the position where the character is first encountered.

1. Here is the algorithm.
2. Create a dict that will map characters to indices
3. Loop through the characters in the string. If it is not in the dict then add a mapping between the character and an index
4. Increment the index

**Solution**

1. Use the **CharacterLevelEncoding.ipynb** notebook as before
2. Add the code below to create a function

from collections import **OrderedDict**

def positional\_encode\_chars(text):

char\_to\_index = OrderedDict()

index = 1

for character in text:

if character not in char\_to\_index:

char\_to\_index[character] = index

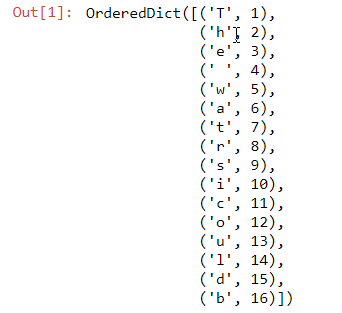
index +=1

return char\_to\_index

1. Test the function

positional\_encode\_chars(‘The water was as wet as it could be’)

The above input produces the following output



Displays a python dictionary of each charactar encoded as an integer

Advantages of Positional Encoding

1. It is a simple and easy way to encode characters. Once could use it for simple programs do not need the complexity of one-hot encoding
2. It can encode any character. If you recall that using specific character sets such as ASCII to encode characters

Disadvantages of Positional Encoding

1. Because the numerical values are increasing i.e. 1,2,3,4,5 some machine learning algorithms assume that some values are greater than others. For example if the letter ‘A’ is encoded as 1, and ‘B’ is encoded as 2 the algorithms might treat the letter ‘B as greater than ‘A’ which is not what we want it to learn.
2. Machine learning algorithms are more efficient when values are within a certain range. For example, deep learning algorithms perform better when the values are in the range 0 to 1 or -1 to 1. For this reason, a lot of ML libraries include utilities that scale the values to fit within these ranges.

These disadvantages also apply to natural language processing algorithms which are usually built on other machine learning algorithms.

Note about Positional Encoding

Positional encoding is rarely used as a final encoding step in machine learning. Instead algorithms first positional encode characters(or words) and then convert to one-hot or embedding representations

Announce 7: We are going to look at One-Hot Encoding – a very common technique used in machine learning. It is also heavily used in Natural Language Processing and so we will try to understand how one-hot encoding is used to create vectors that can represent text

One Hot Encoding

In Natural Language processing a one-hot vector is a 1 X N matrix (vector) used to represent each word in a vocabulary. The vector is made up of all zeros except for a 1 in a single cell.

Each word in the vocabulary has a 1 in a different position and that is used to distinguish each word

Figure showing one-hot enoding of three items

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Food** | **Categorical** |  | **Apple** | **Orange** | **Broccoli** |
| Apple |  | 🡪 | 1 | 0 | 0 |
| Orange |  |  | 0 | 1 | 0 |
| Broccoli |  |  | 0 | 0 | 1 |

So, when one-hot encoding is used to encode characters the value of N for the vector will be the total number of characters that exist in the text. If we are interested in encoding only ascii characters for example we would create vectors the same length as the number of ascii characters. If we are using a different character encoding – e.g. a Unicode charset we would alternatively create one-hot vectors of the length of the number of characters in that charset. It is also possible to enumerate the total number of characters in a give document or set of documents in order to determine the one-hot vector size but it is more common to use utilities to do so.

One hot encoding is such an accepted practice in machine learning and natural language processing that almost all ML and NLP libraries either provide utilities for one-hot encoding or accept input as one-hot encoded vectors.

Discussion 7: Discuss how what the size of a vector would be needed to represent 30 words. Or 100 words. Or 100,000. Try to get the students to understand that to create one-hot vectors ypu need the entire vocabulary

Figure showing one-hot encoding of words

Liverpool

Milan

Word N

|  |  |  |
| --- | --- | --- |
| **Milan** | **=** | [**1**, 0, 0, 0, 0, 0, 0, …, 0 ] |
| **Liverpool** | **=** | [0, **1**, 0, 0, 0, 0, 0, …, 0 ] |
| **Rome** | **=** | [0, 0, **1**, 0, 0, 0, 0, …, 0 ] |
| **Belgrade** | **=** | [0, 0, 0, **1**, 0, 0, 0, …, 0 ] |

One Hot Encoding and Categorical Variables

One-hot encoding is only applicable for categorical variables but not continuous variables. Text can be considered a type of categorical variable i.e. each character is drawn from a finite set of characters.

Key Steps in One Hot Encoding

To create one hot encoding for text the following steps are required

1. Decide whether you want to encode characters or words
2. Choose how to tokenize the text and which tokens are irrelevant for the project and can be removed
3. Determine the total number of characters or words. This is simply the total number of unique tokens

|  |  |
| --- | --- |
| **Library** | **On-Hot Encoding Method** |
| **Keras** | keras.preprocessing.text.one\_hot  keras.utils.to\_categorical |
| **Scikit-Learn** | sklearn.preprocessing.OneHotEncoder |
| **Tensorflow** | tf.one\_hot |
| **pandas** | pandas.get\_dummies |

Exercise: Character One Hot Encoding – Manual

In this exercise we will create a function that can one-hot encode the character of words.

1. Launch Jupyter notebook and select and open the notebook **OneHotEncodingText.ipynb**
2. Add a new Cell and change the type to Markdown
3. Add the following content

## Manual One Hot Encoding

1. Add the following code

def onehot\_word(word):

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

print(lookup)

word\_vector = []

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

Here is a detailed explanation

* 1. First, we create a lookup table for each of the characters in the word. The function enumerate() take the word and split into the unique characters then we map each character to a positional index
  2. Next, we loop through the characters in the word and create a vector the same size as the number of characters. This vector is filled with 0’s
  3. Finally, we use the lookup table to find the position of the character and set that value to 1

1. Now we can test by adding a cell that contains different values. For example create a new code cell and run with the value

onehot\_word(‘data’)

Note that while this function works for individual words it does not work for multiple words across separate function calls.

Discussion 7: Discuss how the function could be changes so that if we repeatedly call if for different words it could operate correctly. Perhaps we need to keep a lookup table of all the characters we might look at over all the calls? This introduces the concept of a vocabulary – and in natural language processing this is something we have to take into consideration

One Hot Encoding in Different Libraries

The **keras** library provides a method called keras.utils.to\_categorical that allows you to convert sequence of ints into one-hot vectors. To use it for text you first

1. Convert the text to integer sequences. You can use the same approach for positional encoding that we learnt in the previous section
2. Convert the integer sequences to categorical

Note that keras also has a function called keras.preprocessing.text.one\_hot. For this lesson we will stick to the to\_categorical method

Exercise: Character One Hot Encoding with Keras

In this exercise we will use the **keras** library to perform character encoding.

keras is a machine learning library that works along with Tensorflow to create deep learning models. Deep learning is being used more often nowadays in natural language processing and so there is some benefit to learning how to use the common tools like keras.

1. In the notebooks directory open the notebook OneHotEncodingWithKeras.ipynb
2. Import the **keras** packages that we will use for this exercise

from keras.preprocessing.text import Tokenizer

1. Create an instance of a Tokenizer. Because we are encoding at the character level in the constructor, we need to set the variable char\_level=True. This would be set to False (by default) if we were encoding words

char\_tokenizer = Tokenizer(char\_level=True)

1. Create a test sentence or sequence of characters with which to test the Tokenizer

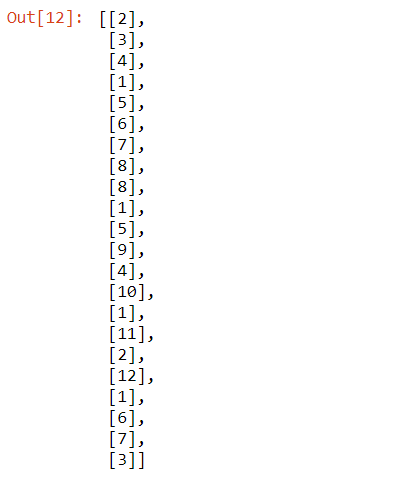
sentence = 'The quick brown fox jumped over the lazy dog'

1. Now fit the char\_tokenizer on a sentence. Under the hood the Tokenizer will break the text into characters and internally keep track of the tokens, the indices and everything else needed to perform one-hot encoding

char\_tokenizer.fit\_on\_texts(sentence)

1. Now that the char tokenizer has been fit on the sentence, we can look at the possible output. One type of output is are the char sequences i.e. the integers assigned to each character in the text

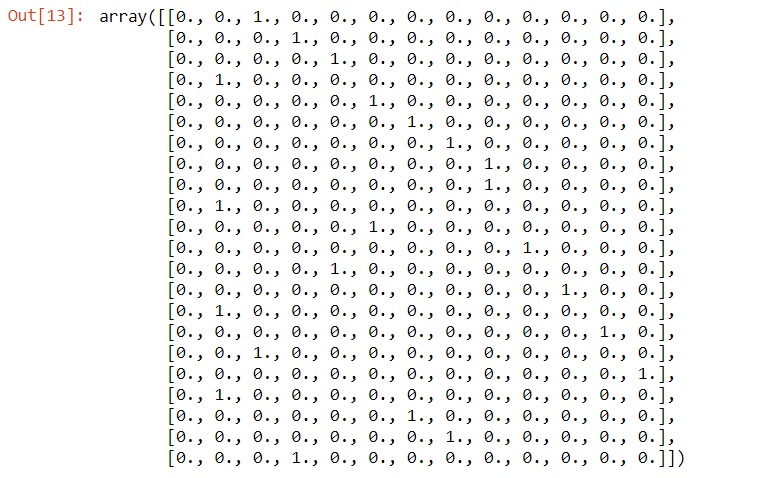
char\_tokenizer.texts\_to\_sequences(sentence)



1. Now we can look at the actual one-hot encoded values. For this we can call tokenizer.texts\_to\_matrix()

char\_vectors = char\_tokenizer.texts\_to\_matrix(text)

char\_vectors



1. As we can see this function produces a numpy array. Let’s investigate the shape

char\_vectors.shape



So char\_vectors is a numpy array with 44 rows and 27 columns. It has 27 columns – one column for each letter of the alphabet plus space and the sentence is 44 characters long. Enter the following code to get the full explanation

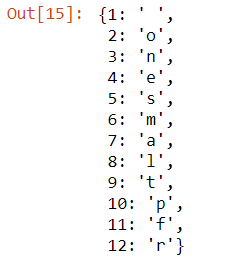
'char\_vectors has shape {} because there are {} characters and there are 26 letters of the alphabet plus space'.format(char\_vectors.shape, len(sentence))

1. Now we can compare the raw char value to the one-hot encoded vector. Note that the first char is ‘t’. The one-hot vector is

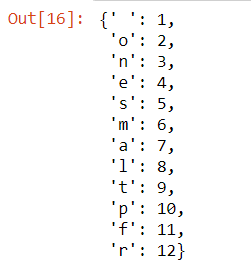
char\_vectors[0]

1. The tokenizer contains two dicts that keep track on the char to index mapping, and the index to char mapping. Let’s look at both

char\_tokenizer.index\_word



char\_tokenizer.word\_index



1. You can retrieve the original character from the one-hot vectors using the char\_tokenizer’s index\_word dict. You will make use of the numpy argmax function

char\_tokenizer.index\_word[np.argmax(char\_vectors[0])]

‘t’

Discussion 7: Discuss the steps in the previous exercise so that the student understands how the process of creating one hot vectors work

Advantages of One-Hot Encoding

1. It is a fairly simple concept and more importantly straightforward for algorithms to process
2. Feature values are automatically scaled in the range 0 and 1 which can make for efficient learning
3. Some types of operations on one-hot vectors are very fast e.g. checking whether two values are the same

Disadvantages of One-Hot Encoding

1. Can be resource intensive especially since each vector can potentially use memory proportional to the number of unique items – the number of characters or the number of words
2. Because of the high dimensional nature of one hot encoded vectors machine learning algorithms can suffer from the curse of dimensionality. When the number of dimensions become large a lot of that is empty space and so not enough information in each area for the algorithm to learn efficiently
3. Each one hot vector is effectively independent and bears no relationship to another. So, if you one-hot encode **auto** and **car** youcannot determine if there is any similarity between those two words. Later on we will look at other means of encoding that can represent words in such as way that we can determine the similarity

Word Level Encoding

Words are the basic units of grammar and meaning of human language and efficient vector representations allow for natural language algorithms to extract the most meaning from text. Many of the same types of representation that are done at the character level can also be done at the word level. You can also one-hot encode words but there are some other interesting ways of representing words that take advantage of the fact that words carry the basic meaning of language.

Word Vocabulary

When one-hot encoding words you need to understand the vocabulary of words that you are dealing with. A vocabulary is the total number of unique words in the text(s) sources for your project. So, if you have a large source e.g. from Wikipedia then you will end up with a huge vocabulary and large one-hot vectors sizes and therefore use a lot more memory.

Exercise X: Word Level One Hot Encoding

In this exercise we will be loading a file containing 100 movie lines. We will be one-hot encoding the words in the file.

1. In the notebooks directory open the notebook OneHotEncodingWords.ipynb
2. First, we need to load the movie lines file. For this we will use pathlib to specify the location of the file.

from pathlib import Path

data = Path('data')

movie\_lines\_file = data / '100lines.txt'

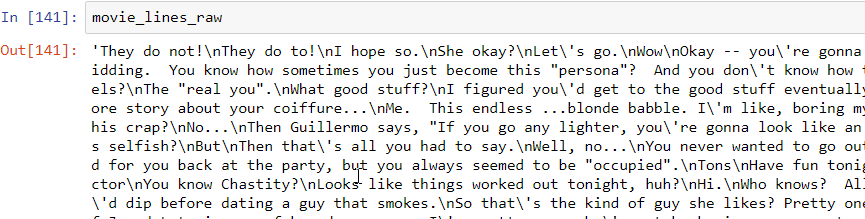
1. Now load the contents of the file. Since we need to create one-hot vectors for each word in the file we need to construct a vocabulary – which is the entire list of unique words in the file.

with movie\_lines\_file.open() as f:

movie\_lines\_raw = f.read()

If you take a look at the variable movie\_lines\_raw you will see a lot of newline characters. This happened because we loaded the entire contents at once into a single variable instead of separate lines. You will also see a lot of non-alphanumeric characters.

movie\_lines\_raw



So, we need to

* 1. Tokenize the string into words
  2. Remove newlines
  3. Remove non-alphanumeric characters

1. Add the following code to do these processing steps

import string

alpha\_characters = str.maketrans('', '', string.punctuation)

def clean\_tokenize(text):

text = text.lower()

text = re.sub(r'\n', '\*\*\* ', text)

text = text.translate(alpha\_characters)

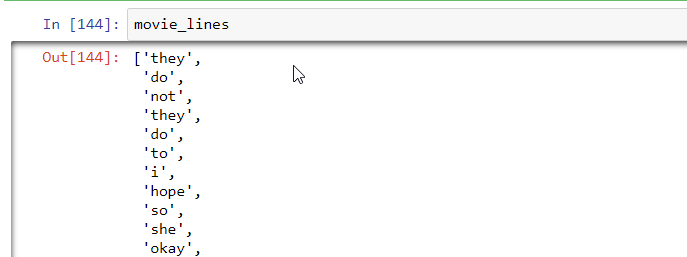
text = re.sub(r' +', ' ', text)

return text.split(' ')

movie\_lines = clean\_tokenize(movie\_lines\_raw)

1. Take a look at the movie lines now. It should now be a list

movie\_lines



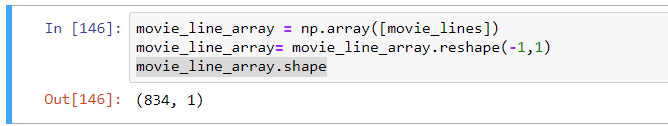
1. The next step is to convert the list into a **numpy** array. Numpy arrays are a form that is more specific to natural language processing than are Python lists, and it is the format that is required for **scikit-learn**, the library that we will use to one-hot encode the words.

Add the following code to convert the list to a numpy array and to print the shape.

movie\_line\_array = np.array([movie\_lines])

movie\_line\_array= movie\_line\_array.reshape(-1,1)

movie\_line\_array.shape



As you can see it is an array with 834 rows and 1 column. Each row is a word in the original movie lines file

1. Now we can use the encoders in the scikit-learn preprocessing package to convert the movie\_line\_array to one-hot encode format. We will use the **OneHotEncoder** to do this.

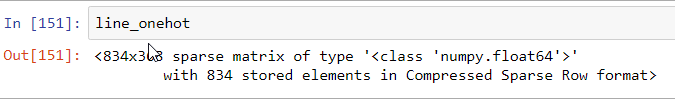
Create a **OneHotEncoder**, fit it to the movie\_lines\_array and then call transform to create a one hot encoded vector

wordOneHotEncoder = preprocessing.OneHotEncoder()

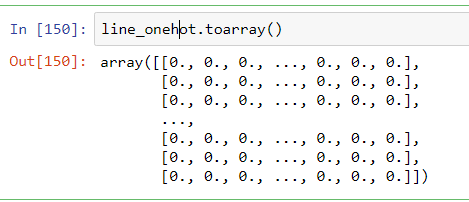
wordOneHotEncoder.fit(movie\_line\_array)

line\_onehot = wordOneHotEncoder.transform(movie\_line\_array)

1. Take a look at the resulting vector. It will be a sparse matrix



1. You can convert it to a dense vector using to array



Word Embeddings

Now we will look at the area of **word embeddings.** The main objectives are the same – we need to find a way to represent text as vectors so that it is useful for machine learning algorithms. One-hot encoding is one way to do this but we need to find a way around these limitations:

1. One hot vectors can get quite large
2. One-hot vectors are sparse
3. One-hot vectors contain no information about correlation between themselves. In other words you can choose two vectors and determine if they are close or far apart

With these limitations in mind the goal is to find a vector representation of words that is

1. Compressed in fewer dimensions
2. Contain more information
3. Each word vector contains some information in relation to other vectors. For example, you can do a correlation between the word vector for man and the word vector for woman.

This leads into the approach of word embeddings. Word embeddings are a special type of vector that try to achieve the objectives we set out above. Let’s look generally at what an embedding is.

Embeddings

An embedding is a mathematical structure that contains another structure. So, you can say an embedding vector contains or represents a value of something. An embedding is usually the result of a function f: X🡪 Y where

Word embeddings are a special type of embedding where the word (and the meaning of the word) is embedded into a vector.

Word Embeddings and the Meaning of Words

Intuitively an **embedding** is a structure into which we **embed** information. When we speak of a **word embedding** we mean that we have a structure into which we **embed** the meaning of a word.

So, each *individual* embedding vector contains information about the *meaning* of a single word. For example, we can embed the meaning of the word **king** into a 100-dimension vector. We can then use that vector to represent the word king in a machine learning programs or in search indexes.

Word meanings are determined by looking at the word and other words that tend to surround it in sentences. So, the word **king** tends to be in sentences such as

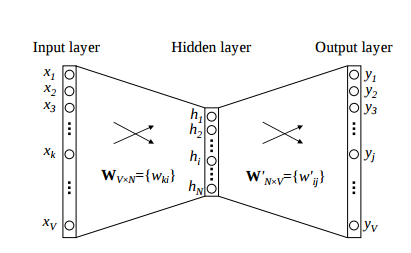
***James VI and I****King of Scotland as****James VI****from 24 July 1567 and King of England and Ireland as****James I****from the union of the Scottish and English crowns on 24 March 1603 until his death in 1625.*

and king tends to be in documents that also include the words **crown** or **reign**. But so does the word **queen**, and it turns out that the word embedding for **queen** should be similar to the one for king because they are used in similar contexts.

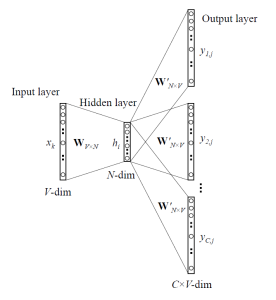
To train word embeddings we find a large source of words such as Wikipedia and Google News and use algorithms to learn the word embeddings. The word embeddings are trained to produce vectors that can predict which words are used in the same context. So, word embeddings are formed by predictive models.

**Continuous Bag of Words**

Continuous bag of words (CBOW) we try to predict the center word given the words that surround it.



**Skip Grams**



Advantages of the Skip Gram Model

1. You can have multiple target vectors for an input word e.g. for Apple one vector for the fruit and one for the company
2. Skip-gram with (negative subsampling) outperforms every other method currently
3. Word embeddings are usually trained using neural network models. We will look at one such model – Word2Vec.

Word2Vec

Word2Vec is an algorithm developed at Google for training word vectors. It takes a text corpus as input and produces word vectors as output. It learns the output word vectors by a predictive method – it tries to predict the word vectors that represent how words occur together.

Exercise: Training Word Vectors

In this exercise we create word vectors using the **gensim** library. To create word vectors, we need a source of documents. For this we will use books available on Project Gutenberg.

We will write code that will download books from Project Gutenberg as the training dataset so you should have access to the internet.

1. In the notebooks directory open the notebook **WordVectors.ipynb**
2. Add the following import statements. We will be using the **requests** library to load books from the Gurenberg website. We will be using the **json** library to load a book catalog

import requests

import json

1. Add the following code that will declare load a book from the Gutenberg website using its id. The code will also clean the text by removing newlines.

import re

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)

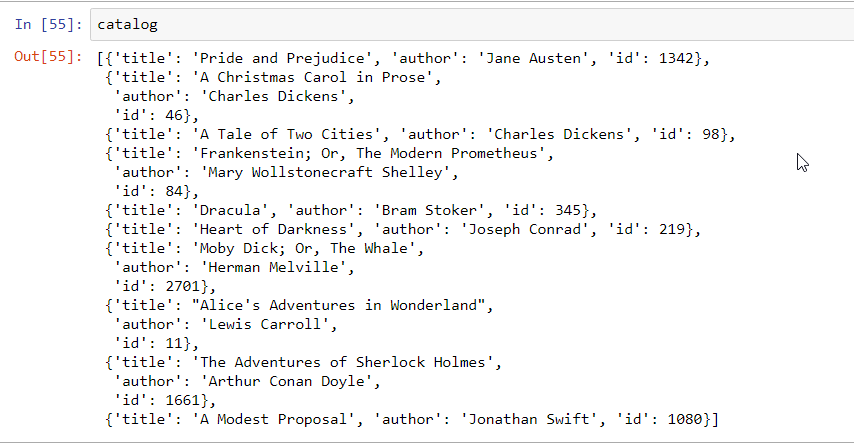
return cleaned\_contents

1. Load the book catalog file. It should be located in a file called ProjectGutenbergBooks.json in the directory where the notebook is running

with open('ProjectGutenbergBooks.json', 'r') as catalog\_file:

catalog = json.load(catalog\_file)

1. Print the catalog



1. Load the books

book\_ids = [ book['id'] for book in catalog ]

books = [ load\_book(id) for id in book\_ids]

1. Before we can train the word vectors we need to break the books into a list of document. Let’s think about what a document is. In this case we want to teach Word2Vec about words in the context of the sentences that they are in. So in our case a “document” is actually a sentence. So, we need to create a list of sentences from all 10 books.

The **gensim** library has a utility called textcleaner that can split text into sentences and it also has a utility function to split a sentence into words. Let’s use both

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

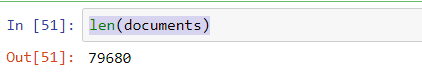
1. Now split the books into sentences and the sentences into documents

books\_sentences = [to\_sentences(book) for book in books]

documents = [sentence for book\_sent in books\_sentences for sentence in book\_sent]

1. Look at the number of documents

len(documents)



1. Now that we have our documents, we can train Word2Vec

from gensim.models import Word2Vec

# build vocabulary and train model

model = Word2Vec(

documents,

size=100,

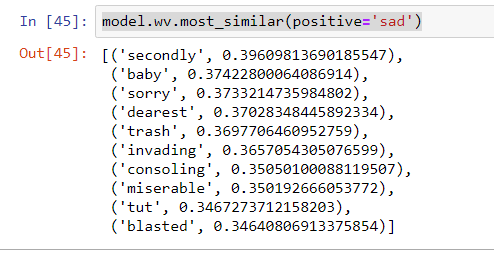
window=10,

min\_count=2,

workers=10)

model.train(documents, total\_examples=len(documents), epochs=50)

1. The word vectors are contained in a model.wv. Use it to find similar word



1. Add the following code to show the vector

import matplotlib.pyplot as plt

plt.style.use('fivethirtyeight')

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,

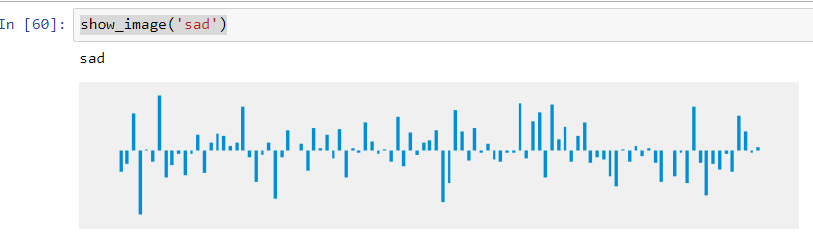
labelbottom=False)

ax.grid(False)

print(word)

ax.bar(range(len(vector)), vector, 0.5)

show\_image('sad')



Word Similarity and Cosine Distance

Word2Vec has a tool called distance that can tell you the closest words to a given word. For example here are the words that are most similar to france.

|  |  |
| --- | --- |
| **spain** | 0.678515 |
| **belgium** | 0.665923 |
| **netherlands** | 0.652428 |
| **italy** | 0.63313 |
| **switzerland** | 0.622323 |
| **luxembourg** | 0.610033 |
| **portugal** | 0.577154 |
| **russia** | 0.571507 |
| **germany** | 0.563291 |
| **catalonia** | 0.534176 |

Using Pretrained Word Vectors

Present 28: Slide introducing the new topic

So, as it turns out word vectors have become a pretty important input in natural language processing pipelines. Now that we have shown how to train them, we can look at a quicker way to get the word vectors required for NLP projects.

Once word vectors have been trained on a language source, and used in a project, they can actually be stored and reused for other related projects. Depending on the training source, for example Wikipedia, the number of words and the context in which they were used is large enough to be used for other projects.

This is part of what is known as transfer learning, which is the application of training from one project to another – using the artifacts that were trained in the first projects.

Note

Pretrained word vectors can get pretty large. For example, vectors trained on the Google News contain 3 million words and on disk its compressed size is 1.5GB

Sources of Word Vectors

There are quite a few different sources for pretrained word vectors.

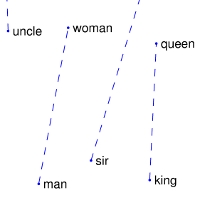
|  |  |
| --- | --- |
| **Word Vector** | **Link** |
| Word2Vec | [Google News vectors](https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing) |
| Glove Embeddings | [Glove Vectors](https://nlp.stanford.edu/projects/glove/) |
| FastText | [Fast Text Word Vectors](https://fasttext.cc/docs/en/pretrained-vectors.html) |

Loading Pretrained Vectors

Pretrained word vectors are usually Python or C++ based binary files. The most popular word vectors are Python based and so it’s possible and usually easy to load into Python programs.

Glove Embeddings

Glove which stands for **G**lobal **Ve**ctors for Word Representation, is a project from Stanford University for training vector representation of words.



Word vectors are trained on Wikipedia, Common Crawl and Twitter – and are available in different vector sizes – 50, 100, 300, depending on the needs of your machine learning project.

**Exercise: Loading Pretrained Word Vectors**

In this exercise we will load and use pretrained word embeddings from Stanford Glove.

1. In the notebooks directory open the notebook **PretrainedVectors.ipynb**
2. Add the following import statement

import numpy as np

1. Add the following code to unzip the Glove embeddings from the zip file

GLOVE\_DIR = '../data/glove/'

GLOVE\_ZIP = GLOVE\_DIR + 'glove.6B.50d.zip'

import zipfile

zip\_ref = zipfile.ZipFile(GLOVE\_ZIP, 'r')

zip\_ref.extractall(GLOVE\_DIR)

zip\_ref.close()

1. The Glove vector file is a text file containing a dictionary of word -> vector. So, we need to read the file line by line, split it and then map to a python dictionary

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:]])

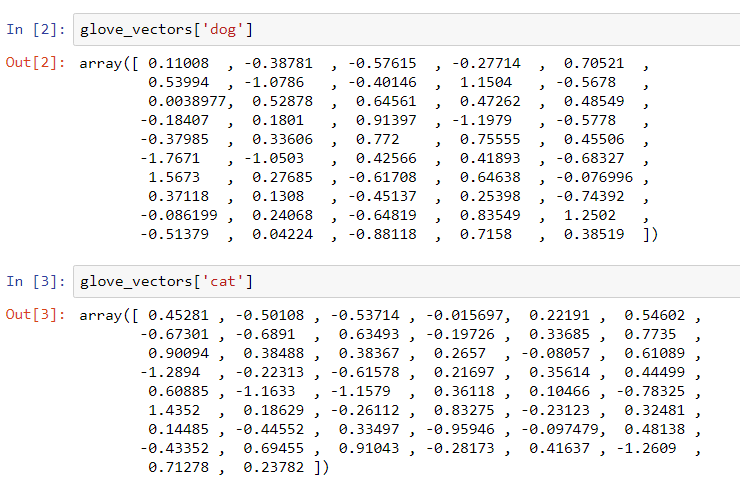
model[word] = embedding

print("Loaded {} words".format(len(model)))

return model

glove\_vectors = load\_glove\_vectors('../data/glove/glove.6B.50d.txt')

1. The glove vector object is basically a dictionary containing the mapping from words to vectors. So, you can access the vector for a word using the python syntax glove\_vectors[‘word’]. This will return a 50 dimensional vector – because we loaded the glove.6B.50d.txt file.



1. We can add some code to display the vector as an image. Add a new code cell and add the following code

import matplotlib.pyplot as plt

plt.style.use('fivethirtyeight')

%matplotlib inline

def to\_vector(glove\_vectors, word):

vector = glove\_vectors.get(word.lower())

if vector is None:

vector [0] \* 50

return vector

def show\_vector(glove\_vectors, word):

vector = to\_vector(glove\_vectors, word)

to\_image(vector, word)

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,

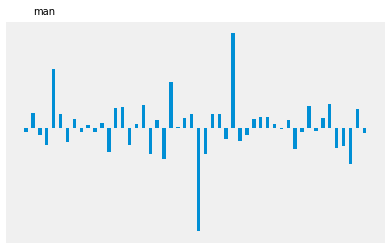
labelbottom=False)

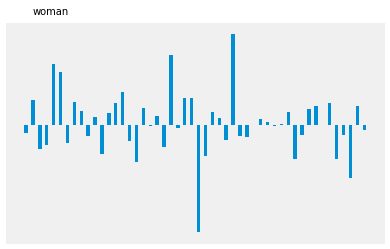
ax.grid(False)

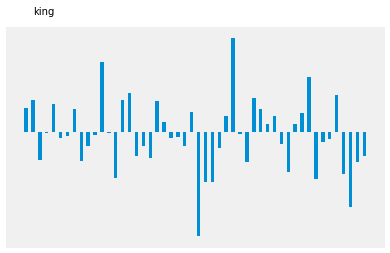
ax.bar(range(len(vector)), vector, 0.5)

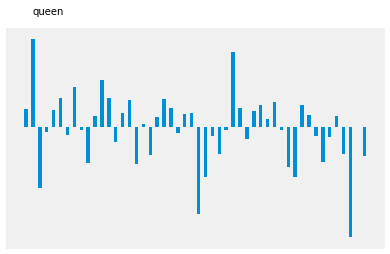
ax.text(s=word, x=1, y=vector.max()+0.5)

return vector









Document Vectors

Word vectors and word embeddings work or representing words but what if we wanted to represent whole documents. Document vectors is an extension of the same ideas a word vectors but for documents.

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 product review, a tweet or a line of movie dialogue and can be a few words or as many as thousands. What determines what we call a document is that we can identify it and use it in a machine learning project as an instance of something that the algorithm can learn from. More importantly, it depends on the objective of the project – so if the end goal is to read a tweet and predict something about it then that would consist a document.

Discussion 7: Discuss projects that students might want to do that involve analyzing text using document vectors. For example, could we use document vectors to predict events based on tweets, or news articles

Uses of Document Vectors

1. Similarity. We can use document vectors to compare texts for similarity. Legal AI software can use document vectors to find similar legal cases
2. Recommendations. For example, online magazines can recommend similar articles based on others that users have read
3. Predictions. Document vectors can be used as the input into machine learning algorithms in to build a predictive model

Exercise: From Movie Dialogue to Document Vectors

In this exercise we will convert movie dialogue into document vectors.

Each line of the movie will be converted to a vector.

Figure showing the required transformation of a line of text into a vector

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Looks like things worked out tonight, huh? | to |  |  |  |  |  |  |  |  |  |  |

For this exercise we will use a part of the Cornell Movie Dialogue Dataset.

You can find the full dataset at <https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html>

The dataset should be available locally in the lesson directory:

**vector-representations/data/cornell-movie-dialogs/**

1. Open the notebook called DocumentVectors.ipynb.
2. Add the import statements for the libraries we will use in this exercise. We will be using the **gensim** library.

The main **gensim** objects for Document Vectors are **Doc2Vec** and **TaggedDocument** and we will also need to import some utility and preprocessing 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

1. Set the display column width to be as wide as it needs to be in order to display the movie lines

pd.set\_option('display.max\_colwidth', -1)

1. Add the following declaration for the location of the movie lines file

movie\_lines\_file = '../data/cornell-movie-dialogs/movie\_lines.txt'

1. Load the movie dialogs. You will need to iterate over the lines in the file and split the columns. The columns are delimited by '+++$+++'

Then you will create a dataframe containing the movie lines.

with open(movie\_lines\_file) as f:

movie\_lines = [line.strip().split('+++$+++') for line in f.readlines()];

lines\_df = pd.DataFrame([{'LineNumber': d[0].strip(),

'Person': d[3].strip(),

'Line': d[4].strip(),

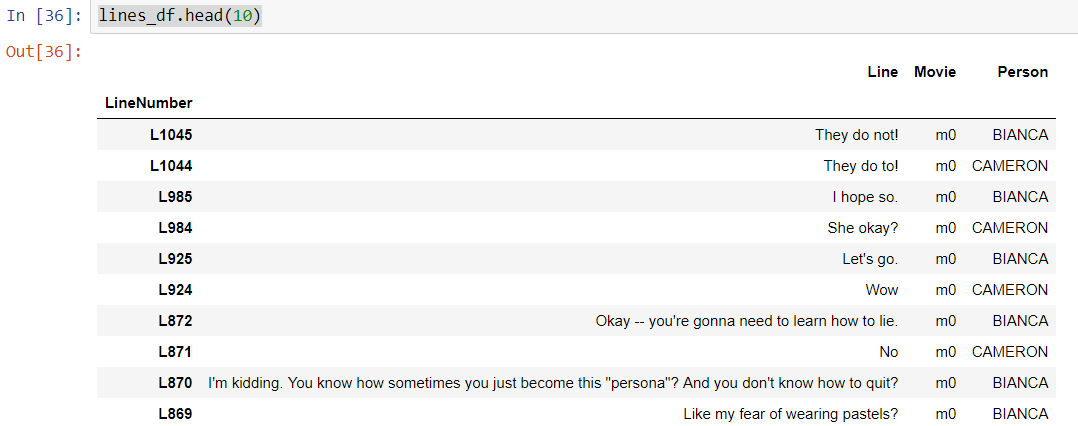
'Movie' : d[2].strip()}

for d in movie\_lines])

lines\_df = lines\_df.set\_index('LineNumber')

Take a quick look at the movie dialogue. You can use the following functions to look at the basic statistics of the **lines\_df** dataframe – **len, head, nunique**.

lines\_df.head(10)



Look at the basic stats of the movie lines file. Use len() and nunique()

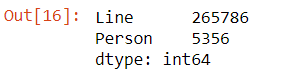
Figure showing the length of the movie\_lines dataframe

len(movie\_lines)



Figure showing the number of unique records in the movie\_lines dataframe

movie\_lines\_df.nunique()



1. Now because there are over 300,000 movie dialogue lines training might take a while. We can train on a subset of the movie lines. Let’s limit the training to 50000 rows

lines\_df\_small = lines\_df.head(50000)

1. We can now create the object that will create the training instances for the Doc2Vec model.

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

Doc2Vec requires each instance to be a TaggedDocument instance so internally we create a list of TaggedDocument for each movie line in the file.

1. Now create the document dataset object. Note that we specify which column contains the “document” – in this case it is the movie line

documents\_dataset = DocumentDataset(lines\_df\_small, 'Line')

1. Create the Doc2Vec model

model = Doc2Vec(min\_count=1, window=5, vector\_size=100, sample=1e-4, negative=5, workers=8)

model.build\_vocab(documents\_dataset.tagged\_documents())

1. Now train the model. It could take a while depending on how many records we train with. Note that we also set the number of epochs, or times through the training set to a reasonable number.

model.train(documents\_dataset.tagged\_documents(shuffle=True),

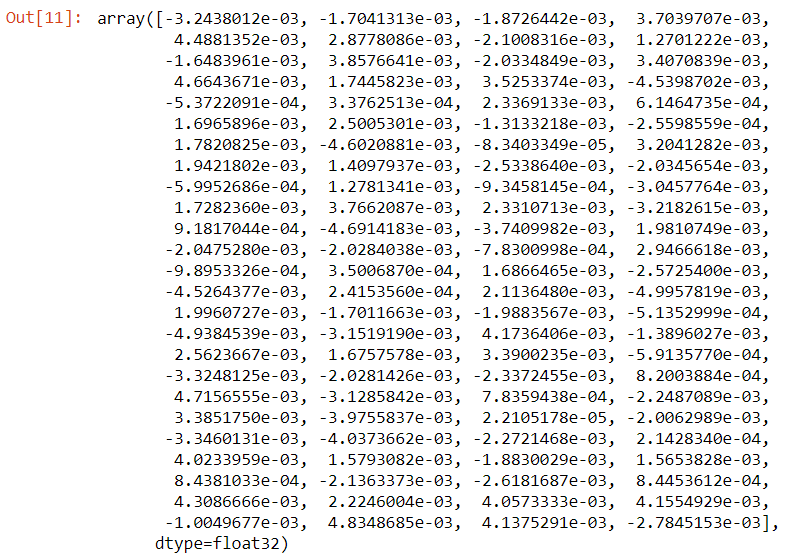
total\_examples = model.corpus\_count,

epochs=10)

1. Look at one of the vectors

model['L1045']

Figure showing a line of a movie represented as a vector of numbers



1. Add code to display each vector. We will define the following functions
   1. **show\_image** takes a vector and displays it as an image
   2. **show\_movie\_line** takes a line number e.g. L200 and returns the movie line and the vector for that line

import matplotlib.pyplot as plt

plt.style.use('fivethirtyeight')

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\_movie\_line(line\_number):

line = lines\_df\_small.ix['L1045'].Line

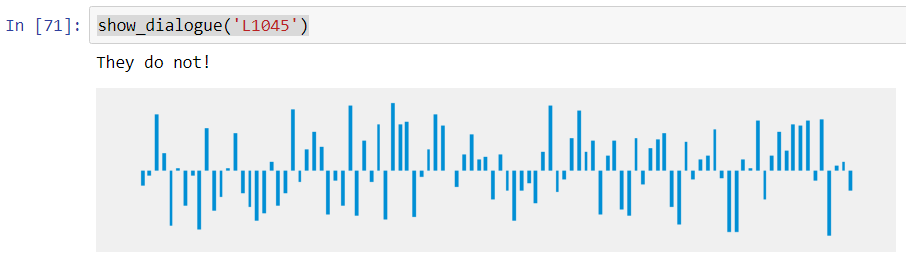
doc\_vector = model[line\_number]

show\_image(doc\_vector, line)

1. Now all the method we implemented above to display the movie line

show\_dialogue('L1045')

Figure showing a line of a movie represented as a vector



What can you do with document vectors

Now that we can turn movie lines into vectors, we can do interesting projects. One project that we can do is build a movie search engine that finds similar movie lines to any one that the user provides.

Summary

Present Slides 40 and 41 to summarize this lesson

During this lesson we learnt