Assignment 5: Random Indexing (Clarifications & Tips)

Generated by Doxygen 1.8.14

Contents

1 Class Documentation

1.1 RandomIndexing Class Reference

Class for creating word vectors using Random Indexing technique.

Public Member Functions

- def __init__ (self, filenames, dimension=2000, non_zero=100, non_zero_values=list([-1, 1]), left_window_size=3, right_window_size=3)

  Object initializer Initializes the Random Indexing algorithm with the necessary hyperparameters and the textfiles that will serve as corpora for generating word vectors.
- def clean_line (self, line)

  A function cleaning the line from punctuation and digits.
- def text_gen (self)

  A generator function providing one cleaned line at a time.
- def build_vocabulary (self)

  Build vocabulary of words from the provided text files.
- def vocabulary_size (self)

  Get the size of the vocabulary.
- def create_word_vectors (self)

  Creates word embeddings using Random Indexing.
- def find_nearest (self, words, k=5, metric='cosine')

  Function returning k nearest neighbors with distances for each word in `words`
- def get_word_vector (self, word)

  Returns a vector for the word obtained after Random Indexing is finished.
- def vocab_exists (self)

  Checks if the vocabulary is written as a text file.
- def read_vocabulary (self)

  Reads a vocabulary from a text file having one word per line.
- def write_vocabulary (self)

  Writes a vocabulary as a text file containing one word from the vocabulary per row.
- def train (self)

  Main function call to train word embeddings.
- def train_and_persist (self)

  Trains word embeddings and enters the interactive loop, where you can enter a word and get a list of k nearest neighours.

1.1.1 Detailed Description

Class for creating word vectors using Random Indexing technique.

Author

Dmytro Kalpakchi dmytroka@kth.se

Date

November 2018

1.1.2 Constructor & Destructor Documentation

1.1.2.1 __init__()

```
def __init__ (
              self,
              filenames,
              dimension = 2000,
              non_zero = 100,
              non_zero_values = list([-1, 1]),
              left_window_size = 3,
              right_window_size = 3 )
```

Object initializer Initializes the Random Indexing algorithm with the necessary hyperparameters and the textfiles that will serve as corpora for generating word vectors.

The `self.__vocab` instance variable is initialized as a Python's set. If you're unfamiliar with sets, please follow this link to find out more: https://docs.python.org/3/tutorial/datastructures.html#sets.

Parameters

self	The RI object itself (is omitted in the descriptions of other functions)
filenames	The filenames of the text files (7 Harry Potter books) that will serve as corpora for generating word vectors. Stored in an instance variable self.__sources.
dimension	The dimension of the word vectors (both context and random). Stored in an instance variable self.__dim.
non_zero	The number of non zero elements in a random word vector. Stored in an instance variable self.__non_zero.
non_zero_values	The possible values of non zero elements used when initializing a random word. Stored in an instance variable self.__non_zero_values. vector
left_window_size	The left window size. Stored in an instance variable self__lws.
right_window_size	The right window size. Stored in an instance variable self__rws.

1.1.3 Member Function Documentation

1.1.3.1 clean_line()

```
def clean_line (
              self,
              line )
```

A function cleaning the line from punctuation and digits.

The function takes a line from the text file as a string, removes all the punctuation and digits from it and returns all words in the cleaned line.

Parameters

line	The line of the text file to be cleaned

Returns

A list of words in a cleaned line

1.1.3.2 text_gen()

```
def text_gen (
              self )
```

A generator function providing one cleaned line at a time.

This function reads every file from the source files line by line and returns a special kind of iterator, called generator, returning one cleaned line a time.

If you are unfamiliar with Python's generators, please read more following these links:

- https://docs.python.org/3/howto/functional.html#generators
- https://wiki.python.org/moin/Generators

Returns

A generator yielding one cleaned line at a time

1.1.3.3 build_vocabulary()

```
def build_vocabulary (
              self )
```

Build vocabulary of words from the provided text files.

Goes through all the cleaned lines and adds each word of the line to a vocabulary stored in a variable `self.__vocab`. The words, stored in the vocabulary, should be unique.

Note: this function is where the first pass through all files is made (using the `text_gen` function)

1.1.3.4 vocabulary_size()

```
def vocabulary_size (
              self )
```

Get the size of the vocabulary.

Returns

The size of the vocabulary

1.1.3.5 create_word_vectors()

```
def create_word_vectors (
              self )
```

Creates word embeddings using Random Indexing.

The function stores the created word embeddings (or so called context vectors) in `self.__cv`. Random vectors used to create word embeddings are stored in `self.__rv`.

Context vectors are created by looping through each cleaned line and updating the context vectors following the Random Indexing approach, i.e. using the words in the sliding window. The size of the sliding window is governed by two instance variables `self.__lws` (left window size) and `self.__rws` (right window size).

For instance, let's consider a sentence: I really like programming assignments. Let's assume that the left part of the sliding window has size 1 (`self.__lws` = 1) and the right part has size 2 (`self.__rws` = 2). Then, the sliding windows will be constructed as follows:

```
I really like programming assignments.
^   r      r
I really like programming assignments.
l   ^      r       r
I really like programming assignments.
    l      ^       r           r
I really like programming assignments.
           l       ^           r
I really like programming assignments.
                   l           ^
```

where "^" denotes the word we're currently at, "l" denotes the words in the left part of the sliding window and "r" denotes the words in the right part of the sliding window.

Implementation tips:

- make sure to understand how generators work! Refer to the documentation of a `text_gen` function for more description.
- the easiest way is to make `self.__cv` and `self.__rv` dictionaries with keys being words (as strings) and values being the context vectors.

Note: this function is where the second pass through all files is made (using the `text_gen` function). The first one was done when calling `build_vocabulary` function. This might not the most efficient solution from the time perspective, but it's quite efficient from the memory perspective, given that we are using generators, which are lazily evaluated, instead of keeping all the cleaned lines in memory as a gigantic list.

1.1.3.6 find_nearest()

```
def find_nearest (
              self,
              words,
              k = 5,
              metric = 'cosine' )
```

Function returning k nearest neighbors with distances for each word in `words`

We suggest using nearest neighbors implementation from scikit-learn (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html). Check carefully their documentation regarding the parameters passed to the algorithm.

To describe how the function operates, imagine you want to find 5 nearest neighbors for the words "Harry" and "Potter" using cosine distance (which can be computed as 1 - cosine similarity). For that you would need to call 'self.find_nearest(["Harry", "Potter"], k=5, metric='cosine')`. The output of the function would then be the following list of lists of tuples (LLT) (all words and distances are just example values):

```
[[('Harry', 0.0), ('Hagrid', 0.07), ('Snape', 0.08), ('Dumbledore', 0.08), ('Hermione', 0.09)],
 [('Potter', 0.0), ('quickly', 0.21), ('asked', 0.22), ('lied', 0.23), ('okay', 0.24)]]
```

The i-th element of the LLT would correspond to k nearest neighbors for the i-th word in the `words` list, provided as an argument. Each tuple contains a word and a similarity/distance metric. The tuples are sorted either by descending similarity or by ascending distance.

Parameters

words	A list of words, for which the nearest neighbors should be returned
k	A number of nearest neighbors to be returned
metric	A similarity/distance metric to be used (defaults to cosine distance)

Returns

A list of list of tuples in the format specified in the function description

1.1.3.7 get_word_vector()

```
def get_word_vector (
              self,
              word )
```

Returns a vector for the word obtained after Random Indexing is finished.

Parameters

word	The word as a string

Returns

The word vector if the word exists in the vocabulary and None otherwise.

1.1.3.8 vocab_exists()

```
def vocab_exists (
              self )
```

Checks if the vocabulary is written as a text file.

Returns

True if the vocabulary file is written and False otherwise

1.1.3.9 read_vocabulary()

```
def read_vocabulary (
              self )
```

Reads a vocabulary from a text file having one word per line.

Returns

True if the vocabulary exists was read from the file and False otherwise (note that exception handling in case the reading failes is not implemented)

1.1.3.10 train()

```
def train (
              self )
```

Main function call to train word embeddings.

If vocabulary file exists, it reads the vocabulary from the file (to speed up the program), otherwise, it builds a vocabulary by reading and cleaning all the Harry Potter books and storing unique words.

After the vocabulary is created/read, the word embeddings are created using Random Indexing.

The documentation for this class was generated from the following file:

- random_indexing.py

Index