PROGRAMMING ASSIGNMENT 4

Issue Date : 02.05.2025 - Friday
Due Date : 11.05.2025 - Sunday (23:59)
Advisor: Res. Asst. İsmail Furkan Atasoy
Programing Language : Python

The objective of this project is to explore selected applications of word embedding vectors and to develop hands-on experience through their implementation. To this end, the Gensim library will be employed to work with Word2Vec representations. The pre-trained `word2vec-google-news-300` model will be utilized. You can start the model with the following code part:

```python
import gensim.downloader
model = gensim.downloader.load("word2vec-google-news-300")}
```

1 Replace with Similar Words

In this section, given a sentence and a list of indices, the words at the specified indices will be randomly replaced with their most similar alternatives according to the Word2Vec model.

For this purpose, the function `replace_with_similar(sentence, indices)` should first be defined. The `sentence` parameter will contain a single string, while the `indices` parameter will be a list of indices. The indices provided to the function will always be valid, and it is also guaranteed that each word corresponding to the given indices exists in the training set of the model.

The sentence will be tokenized based on whitespace. For each token specified by the indices, the top 5 most similar words along with their similarity scores will be retrieved from the model. For this purpose, the `most_similar()` method of the `model` object should be used.

These words and their scores will be stored in a dictionary, where each original token maps to a list of tuples. Each tuple contains a similar word and its corresponding similarity score.

Assume that a sentence containing 10 words and a list of two indices are given as parameters, the structure of the resulting dictionary should be as follows:

```
{
word_1: [(similar_word_1, score), ..., (similar_word_5, score)],
word_2: [(similar_word_2, score), ..., (similar_word_5, score)]
}
```

`word_1` and `word_2` are the words corresponding to the given indices.

Once this dictionary is completed, one of the five words from the list of similar words for each token will be randomly selected and used to replace the original word, thereby creating a new sentence.

The `replace_with_similar()` function should return both the dictionary and the new sentence as follows:

`(new_sentence, similar_words_dict)`

Some usage examples are provided below:

```
sentence = "I love AIN442 and BBM497 courses"
indices = [1, 5]
new_sentence, most_similar_dict = replace_with_similar(sentence, indices)

print(most_similar_dict.keys(), end="\n\n")
print(most_similar_dict["love"], end="\n\n")
print(most_similar_dict["courses"], end="\n\n")
print(new_sentence)
```

Output:

```
dict_keys(['love', 'courses'])

[('loved', 0.6907791495323181), ('adore', 0.6816873550415039),
('loves', 0.661863386631012), ('passion', 0.6100708842277527),
('hate', 0.600395679473877)]

[('Courses', 0.7718060612678528), ('course', 0.6795483231544495),
('coursework', 0.6573367714881897), ('classes', 0.6370522379875183),
('noncredit_courses', 0.6368249654769897)]

I adore AIN442 and BBM497 classes
```

Note: Since the replacement word is selected randomly in the example, your `new_sentence` may differ. However, the values in the `most_similar_dict` should be exactly the same (Due to floating point precision issue, slight differences in the last few decimal places are expected).

2 Find Similar Sentences

The purpose of this section is to identify and list the sentences that are most similar to a given query using cosine similarity of Word2Vec vectors. All vector operations in this section will be performed using NumPy.

You will have two different function definitions. These are `sentence_vector(sentence)` and `most_similar_sentences(file_path, query)`.

2.1 sentence_vector(sentence)

The `sentence_vector()` function will take only a sentence string and will store the Word2Vec vectors of each word in the string in a dictionary. This dictionary should look like the following:

```
{
word_1: [vector with 300 dimensions],
word_2: [vector with 300 dimensions],
...
}
```

If the word is not found in the model, then a 300-dimensional NumPy array filled with zeros should be added to the dictionary.

In order to obtain the sentence vector, the vectors of each word in the dictionary will be summed up and divided by the number of words. In other words, the mean of each word vectors will be taken, resulting in a single 300-dimensional sentence vector representation.

The `sentence_vector()` function should return both the dictionary of each word's vectors and the calculated sentence vector as follows:

`(vector_dict, sentence_vec)`

Here are some examples of how to use the `sentence_vector()` function:

```
vector_dict, sentence_vec = sentence_vector("This is a test sentence")

print(vector_dict.keys(), end="\n\n")
print(vector_dict["This"][:5], end="\n\n")
print(vector_dict["a"][:5], end="\n\n")
print(len(vector_dict["test"]))
```

Output:

```
dict_keys(['This', 'is', 'a', 'test', 'sentence'])

[-0.2890625   0.19921875  0.16015625  0.02526855 -0.23632812]

[0. 0. 0. 0. 0.]

300
```

2.2 most_similar_sentences(file_path, query)

The `most_similar_sentences()` function should take a file path and a query sentence as parameter. The query sentence will be a string, and the file path will be `sentences.txt` which contains 20 different sentences.

First, read the file and treat each line as a separate sentence (Don't make any changes on sentences). Then, Using the `sentence_vector()` function defined in the previous subsection, generate sentence vectors for both the query and the 20 different sentences. Each sentence should be paired with the query, and the cosine similarity between their sentence vectors should be calculated. Use `numpy.dot()` and `numpy.linalg.norm()` for your calculations.

For each sentence, create a tuple that contains the sentence itself and its cosine similarity with the query. Then, collect all these tuples into a list. This list should contain 20 tuples in the following format:

```
[
(sentence1, cosine_similarity),
(sentence2, cosine_similarity),
...
(sentence20, cosine_similarity)
]
```

Finally, sort this list in descending order of cosine similarity, and have the function return the sorted list.

Here are some examples of how to use the `most_similar_sentences()` function:

```python
file_path = "sentences.txt"

query = "Which courses have you taken at Hacettepe University ?"

results = most_similar_sentences(file_path, query)

for sentence, score in results[:3]:
    print(f"{score:.5f} -> {sentence}")
```

Output:

```
0.82187 -> I am very happy to have taken the AIN442 and BBM497 courses of
Hacettepe University .

0.64230 -> Students have the chance to gain practice with the homework
given in lab classes of universities .

0.58392 -> I forgot my backpack on campus and when I got there it was in
the same place .
```

Submission

- You will submit your programming assignment using the **HADI** system. You have to upload a single **zip file** (`.zip`, `.gzip` or `.rar`) holding a Python program (`.py` file).
- The name of your solution file should be: `hw04_NameLastname.py` replacing the `NameLastname` with your actual first name and last name. The name of your zip file should match `hw04_NameLastname` with a corresponding extension (`.zip`, `.gzip`, or `.rar`).

Late Policy:

- You must submit your programming assignment **before its due date**.

- You can also submit your assignment up to **three days late**, but with a penalty:
  - **1 day late:** 10% penalty
  - **2 days late:** 20% penalty
  - **3 days late:** 30% penalty

DO YOUR PROGRAMMING ASSIGNMENTS YOURSELF!

- **Do not share your programming assignments with your friends.**
- **Do not complete your entire assignment using AI tools. Comment on the parts where you have received support from AI tools.**
- **Cheating will be punished.**