Tutorial: Statistical Language Processing (CL III)
Summer 2024

Assignment 1: Due Monday, May 13, 2024 - 17:00

Submission

See the general rules which apply to all assignments. Submit the following files to Moodle (only 1 submission per group):

- a1.py with your implementation
- your discussion in pdf format
- Please **do not** submit data, embedding, or unit test files.

Vector Semantics and Embeddings

As you see in Chapter 6 of Jurafsky and Martin's textbook, dealing with word meaning is a central task in computational linguists. Encoding meanings of words (at any level) is far from trivial and is a task that has kept the experts occupied for a long time. By far the most common way of computationally representing words is representing them as vectors - lists of numbers such that each number in the list carries some information about the meaning of the word.

In this assignment, you will use pretrained word embeddings to build a recommender system that makes suggestions based on a user query.

Dataset

Your system will process user queries and recommend several TED Talks based on the query.

We can use the following dataset from Kaggle. Download **ted_main.csv** to the same directory as the starter code. You don't need the transcripts for this assignment.

You will create a single vector for each entry in the *description* column in **ted_main.csv**.

You will also create a single vector for the user query, which can contain multiple words.

You will then find the description vectors that are most similar to the query vector, and recommend those talks to the user.

1 Load and Preprocess Data (Total: 3 pts)

First load the spaCy English small model and the GoogleNews word2vec embeddings as instructed in the **ToDo**s. Please save the word2vec embeddings in the same directory as the script - which will make testing for us easier.

1.1 *load_data()* (1 pt)

Implement the function load_data() that takes the name of the file containing the TED Talk data and loads the *description* and *url* columns into a dictionary of dictionaries containing 2550 entries:

```
data = {
  0: {
    "description": "Sir Ken Robinson makes an entertaining ...",
    "url": "https://www.ted.com/talks/ken_robinson_says_schools_kill_creativity"
    },
  1: {
    "description": "With the same humor and humanity he exuded in ...",
    "url": "https://www.ted.com/talks/al_gore_on_averting_climate_crisis"
    }
    ...
}
```

1.2 *preprocess_text()* (1 pt)

Implement the function preprocess_text() that takes any text as the argument and returns a list of preprocessed tokens, where preprocessing consists of the following steps:

- Tokenization - split each sentence into individual words (tokens)
- Punctuation removal - get rid of punctuation symbols (commas, periods, quotation marks, etc.)
- Stopword removal - remove stopwords (words that arguably do not contribute to the semantic meaning of a text, such as articles, interjections, etc.)
- Url removal - remove urls
- Whitespace removal - get rid of whitespace in the text (tabs, newlines, etc)
- Lowercasing - turn all words to lower case letters (e.g. "Dog" becomes "dog")

All the above steps can be easily accomplished with the spaCy library. Please use the **en_core_web_sm** model.

After implementing all the above steps, your preprocess_text() should work like this:

```
preprocess_text("This is a simple file. It contains: Two Sentences.")
["simple", "file", "contains", "sentences"]
```

1.3 *preprocess_texts()* (1 pt)

Implement the function preprocess_texts() that takes a dictionary, as returned by load_data(), and preprocesses all of the *description* values in the inner dictionaries, and adds the key *pp_text* whose value is the preprocessed tokens. The updated dictionary is returned.

The returned dictionary should have the following structure:

```
data = {
    0: {'description': 'Description of a very interesting talk.',
        'url': 'https://example.com',
        'pp_text': ['description', 'interesting', 'talk']
    },
    1: {'description': 'Another one.',
        'url': 'https://example.com',
        'pp_text': []
    }
}
```

2 Combining Embeddings (Total: 4 pts)

A standard way of combining multiple word vectors into one vector is to simply take the mean of the vectors.

2.1 *get_vector()* (2 pts)

Implement the get_vector() function which takes a list of preprocessed token strings, and returns the mean of the word2vec vectors.

2.2 *get_vectors()* (2 pts)

Implement the function get_vectors() that takes a dictionary, as returned by preprocess_texts(), and preprocesses all of the *pp_text* values in the inner dictionaries, and adds the key *vector* whose value is the mean of vectors for the words in *pp_text* (or None if *pp_text* is an empty list). The updated dictionary is returned.

The returned dictionary should have the following structure:

```
data = {
    0: {'description': 'Description of a very interesting talk.',
        'url': 'https://example.com',
        'pp_text': ['description', 'interesting', 'talk'],
        'vector': [-0.08374023,  0.09472656, -0.02270508, ...]
    },
    1: {'description': 'Another one.',
        'url': 'https://example.com',
        'pp_text': []
        'vector': None
    }
}
```

3 Similarity (Total: 6 pts)

3.1 *cosine_similarity()* (2 pts)

Implement the cosine_similarity() function as described in chapter 6 the J&M textbook, formula 6.9:

$cos(v, w) = \frac{v \cdot w}{|v||w|}$

There are several numpy imports in the starter code that you can use for this.

3.2 *k_most_similar()* (4 pts)

Implement the k_most_similar() function, which takes a user query, the dictionary that was built from the input data, and k that specifies the number of results to return. Return a list of tuples with the result id and the similarity score of the top k results.

4 Recommender Application (Total: 5 pts)

4.1 *recommender_app()* (2.5 pts)

Implement your recommendation system in the *recommender_app()* function according to the instructions.

Note that there are no tests for this function.

4.2 *main()* (2.5 pts)

Implement the *main()* function, which is the entry point of the code.

Build the dictionary and start the application here.

Note that there are no tests for this function.

5 Discussion (2 pts)

Evaluate your system by entering a series of queries and observe the results. Based on the output, how do you rate your system? Do the vectors manage to capture the meaning of the descriptions and queries? Why or why not? Note that you will need to do quite a few queries in order to make a well-based judgement.

Write a **brief (max 1 page)** report of your findings. There are no right or wrong answers, but include examples to back up your statements. Submit the discussion in a separate **PDF file**.