06-Exercises

November 27, 2024

1 06 - Exercises: file input/output

This week we saw: - How to deal with files in input and output in Python - Different file formats for our files - What is text mining - NLTK for natural language processing

Here are some exercises to help you get comfortable with these concepts :)

1.1 0. Some setup

Often, text is not stored plainly, but in some formatted structure, such as Word documents, HTML web pages, or XML files. In this exercise, we will learn: 1. How to install third-party packages with Anaconda 2. How to preprocess such data through the example of reading a Word document. For this you are provided with the first chapter of Mary Shelley's novel *Frankenstein* in the file `Frankenstein.docx`.

NOTE: a *.docx* file is a **compressed archive** that contains a set of different XML documents; each of them can contain content, but also theme details, settings and style instructions:

```
Browsing zipfile Frankenstein.docx:
[Content_Types].xml
_rels/.rels
word/_rels/document.xml.rels
word/document.xml
word/theme/theme1.xml
word/settings.xml
word/fontTable.xml
word/webSettings.xml
docProps/app.xml
docProps/core.xml
word/styles.xml
```

(if you're curious how to do it on your own laptop, try googling "How to browse a .docx file as a compressed archive")

1.2 1. Installing third-party packages with Anaconda - 1 point

The third-party package `python-docx` provides a **Word Document parser** that you will use to load the contents of `Frankenstein.docx` into your Python program. 1. Open Anaconda Navigator and switch to the "Environments" view. 2. Add *channel* "conda-forge" by clicking on the "channels" button, and then "add". 3. Update the package index afterwards. 4. Switch to the "Not installed"

list and search for “python-docx”, then select the package for installation and hit the “Apply” button

[ ]:

1.3 2. Preprocessing the `Frankenstein.docx` file - 3 points

Now that we have our `python-docx` package installed, we can turn to the task of preprocessing the `Frankenstein.docx` file: 1. Read up on the package and its API (Application Programming Interface) at https://python-docx.readthedocs.io/en/latest/. 2. Load the `Frankenstein.docx` file. 3. Iterate over its paragraphs, and extract the content in a single, long string. Delimit paragraphs by a newline (`\n`) character. - You can use a `for` loop, the `join()` function, a list comprehension… 4. Print the contents of the string.

[ ]:

1.4 3. Improving the lemmatization of the Frankenstein test

1.4.1 3.1. Repeating some analyses - 3 points

During the lecture, we have noted that lemmatization without Part-Of-Speech (POS) information will lead to a poor performance. We will now investigate the issue further in this exercise. 1. Repeat the analysis of the lecture in which the WordNetLemmatizer is applied to the Frankenstein text: - Create a list of filtered words by removing punctuation marks and ignoring stop words. - Print the 10 most common words after this filtering - Instantiate the WordNetLemmatizer - Lemmatize words in the filtered words list - Print the 10 most common words after lemmatization of filtered words

[ ]:

1.4.2 3.2. Tagging a text: using nested lists - 3 points

The process of obtaining POS information is called **tagging**. You can find some info about tagging with NLTK here: https://www.nltk.org/book/ch05.html. The following step of the exercise requires some additional data from NLTK’s database, which can be obtained by executing the following commands: - `import nltk` - `nltk.download('average_perceptron_tagger')` - `nltk.download('universal_tagset')`

Now we are ready to continue our analyses! Let’s use NLTK’s functionality to **tag** the Frankenstein text: 1. Tagging can only be performed sentence by sentence. Therefore, alter the analysis shown in the lecture to create a list of sentences, where each sentence is again represented by a list of its contained words that preserves their original order. Use NLTK’s sentence and word tokenizers for this task: - Import necessary packages - Tokenize each sentence in the text by using the `sent_tokenize()` function we used together - Tokenize each word of each sentence by using the `word_tokenize()` function we used together (you need to maintain the nested list structure!) - Print the results

[ ]:

1.4.3 3.3. Tagging a text: using NLTK's `pos_tag()` - 3 points

Now we are ready to tag each sentence of the text using NLTK's `pos_tag()` function! 1. When calling this function, use the parameter setting `tagset='universal'` to indicate the use of the more commonly used *universal* POS annotation (called **tagset*), instead of the NLTK's default one: - Use the `pos_tag()` function on the list containing each tokenized sentence in a list - Print the result

[ ]:

1.4.4 3.4. Tagging a text: Prepare the POS information for the WordNetLemmatizer - 2 points

Before we can supply the newly obtained POS information to the WordNetLemmatizer, the **tagset** must be mapped to the WordNetLemmatizer's tagset dialect. 1. The WordNetLemmatizer can only lemmatize *nouns*, *verbs*, *adverbs*, and *adjectives*, so we need to create a mapping: - Create a dictionary to map between the universal tagset and WordNetLemmatizer's tagset dialect using the following table:

```
NOUN  n
VERB  v
ADV   r
ADJ   a
```

[ ]:

1.4.5 3.5. Lemmatize the sentences - 5 points

1. Create a function `lemmatizeSentences(text, universal_to_wordnet_tagset)` that lemmatizes the given text (represented as a **list of lists of word-tokenized sentences**) and our previously created mapping between the universal and WordNetLemmatizer's tagset. Discard any word whose tag does not appear in the mapping:
   - Create the function `lemmatizeSentences(text, universal_to_wordnet_tagset)`
   - Iterate through your tagged text
   - Discard any word whose tag does not appear in the mapping
   - "Translate" your tagset used the dictionary we previously created
   - Lemmatize each word in your sentence and store it

[ ]:

1.4.6 3.6. Analyze results - 5 points

Finally! After all of our analyses, let's print out the results and compare them. 1. Print the 10 most frequent lemmatized words after tagging 2. Compare the **sets** of lemmatized words before and after tagging.

Are they different? How?

[ ]: