Information Processing and Retrieval

Instituto Superior Técnico 2020

Lab 1: Introduction to text processing and indexing with Python

1 Programming simple text utils

Let us start with some some warm up exercises.

1.1. Define a function that receives a list of objects and sorts the list in place using the *quicksort* algorithm[1]. If needed, use the following pseudocode as a guide.

```
quicksort(A as array, low as int, high as int)
    if (low < high)
        pivotlocation = Partition(A,low,high)
        Quicksort(A,low, pivotlocation -- 1)
        Quicksort(A, pivotlocation + 1, high)

partition(A as array, low as int, high as int)
    pivot = A[low]
    leftwall = low
    for i = low + 1 to high
        if (A[i] < pivot) then
            leftwall = leftwall + 1
            swap(A[i], A[leftwall])
    swap(A[low],A[leftwall])
    return (leftwall)
```

1.2. Implement a script that reads a list of numeric values from a file (containing one value per line) and prints the same values in ascending order.

1.3. Implement a script that reads a text file, containing natural language text, and prints each word it contains and the number of times the word occurs.

1.4. Implement a script that reads two text files and counts the number of words in common.

2 Processing text using nltk

The python extension package named nltk[2] provides a set of facilities for processing natural language text. For example, you can use the following methods:

[1] https://en.wikipedia.org/wiki/Quicksort
[2] http://www.nltk.org

- nltk.sent_tokenize(d), which splits a document d into a list of sentences;
- nltk.word_tokenize(s), which splits a sentence s into a list of words;
- nltk.pos_tag(w), which tags the words in list w according to their part-of-speech (i.e., tag words according to morphosyntactic classes such as noun, verb or adjective);
- nltk.ne_chunk(p, binary=True), which tags the words in list p as named entities or not (where each word in p was previously tagged with a part-of-speech tag).

2.1. Use the *nltk* package to solve word counting problems 1.3 and 1.4.

2.2. Count how many words of each syntactic class (noun, verb, etc.) occur in a document.

2.3. Print all the named entities found for the selected text document.

2.4. nltk[3] documentation further provides multiple algorithmic variants for parts-of-speech tagging and named entity recognition.

Test the senna[4] part-of-speech tagger, named-entity-recognition tagger and chunk tagger on the targeted text.

2.5. Compare the results produced by the senna taggers against the default nltk taggers. Consider using text from different contexts (such as news and blogs).

3 Processing text using SCIKIT-LEARN

scikit-learn is a machine learning library for python[5], which also contains useful functions to map unstructured data into structured data. For example, consider the following classes:

- sklearn.feature_extraction.text.CountVectorizer, which transforms a list of texts into a vector of word counts;
- sklearn.feature_extraction.text.TfidfVectorizer, which transforms a list of texts into a vector of TF-IDF values. TF-IDF values score words in accordance with their relevance for a given text taking into account all texts from the inputted list;

3.1. Solve the word-counting problems using scikit-learn.

Note that vectorizers work by first learning the vocabulary (using method fit) and then transforming the documents into vectors (using method transform).

[3] http://www.nltk.org/api/nltk.tag.html
[4] http://www.nltk.org/api/nltk.tag.html#module-nltk.tag.senna
[5] http://scikit-learn.org/stable/

3.2. Notice that the method transform returns a *sparse matrix*, defined in the numpy[6] package. Search and discuss the major advantages of sparse versus dense matrices.

3.3. Transform two documents into TF-IDF vectors using scikit-learn.

3.4. Compute the cosine similarity between the produced TF-IDF vectors. To this end, consider using cosine_similarity from sklearn.metrics.pairwise.

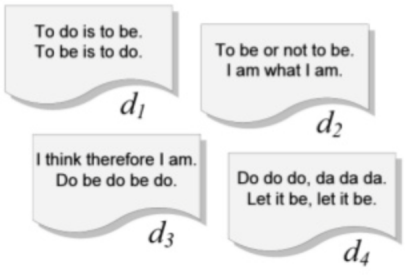
4 **Indexing** (homework)

In the next class we will develop a simple IR system that will be able to index a collection of documents and process search queries. To this end, our first step is to index a collection of documents. Please consider advancing the next lab by solving the following exercises at home.

4.1. Implement a function that reads text documents from a disk directory and creates a simple, in-memory *inverted index* for the given collection.

The inverted index essentially corresponds to a dictionary that contains, for each term, the documents where it occurs and the corresponding document frequency. The following figure schematically illustrates an inverted index created over a collection of four documents.

Vocabulary	n_i	Occurrences as inverted lists
to	2	[1,4],[2,2]
do	3	[1,2],[3,3],[4,3]
is	1	[1,2]
be	4	[1,2],[2,2],[3,2],[4,2]
or	1	[2,1]
not	1	[2,1]
I	2	[2,2],[3,2]
am	2	[2,2],[3,1]
what	1	[2,1]
think	1	[3,1]
therefore	1	[3,1]
da	1	[4,3]
let	1	[4,2]
it	1	[4,2]



Note: do not worry with time and space concerns associated with the inverted index for now.

4.2. Using the programmed inverted index, implement a function to print some statistics on the collection, including: i) the total number of documents; ii) the total number of terms; and iii) the total number of individual term occurrences.

[6] http://www.numpy.org/