Lab 1

Status: Completed
Your identity is visible during marking.
Marks: 7 / 7

Submission deadline: 30 Aug 2019 23:55, 45 days left

Deadline: 27 November 2018

Contact: Chatrine Qwaider (Kathrein Abu Kwaik)

(This lab was designed by Peter Ljunglöf and updated by Simon Dobnik. Minor updates by Gerlof Bouma, Kathrien Abu Kwaik)

The goal of this assignment is to create a simple corpus from raw text. The task is to write regular expressions for word tokenization. You will use the NLTK Treebank corpus, which has a raw text version that you can work on. This corpus contains ca. 5% of the Penn Treebank, (c) LDC 1995, which means around 100,000 words in 4,000 sentences.

Submitting your answers

Please submit the lab as a single Python file with a name following the pattern `lab1-surname.py`. For example, if I were to submit, I would use `lab1-dobnik.py`.

Submit the file in GUL.

The file should run from the command line without arguments, and print out all answers on the terminal, for example:

```
$ python lab1-ljunglof.py
"""Assignment 1: WordNet (deadline: 2018-11-27)
Name 1: NAME
Name 2: NAME
"""
 Part 1
------------------
(...)
Precision: 91.83%
Recall: 93.63%
```

```
F-score: 92.72%

Part 2
------------------
Nr. tokens: 66666
Nr. types: 66666
(...)
```

The code must be well documented, and all functions (no exceptions!) must have docstrings. All computations must be done in functions, the only things that are allowed on the top-level are, in this order:

1. module imports
2. definitions of constants
3. function and/or class definitions
4. a final run-time clause `if __name__ == '__main__'`

This is the structure, and it's strict:

```
# module imports
import nltk
import another_possible_module
(...)

# constants
corpus_size = (...)
token_regexp = r"""(...)"""
(...)

# function/class definitions
def a_function(with, some, arguments):
    """A mandatory docstring"""
    (...)
    return some_return_value

def another_function(more, arguments):
    """This docstring is also compulsory"""
    (...)

# command line interpreter
if __name__ == '__main__':
    do_this(...)
    then_do_that(...)
    # don't do too much here; call functions instead!
```

Acquiring the corpus

First you need to get the corpus containing raw text and the tokenised corpus which will serve as our gold standard for comparison. They are in different NLTK corpora, `treebank_raw` and `treebank_chunk`, respectively. Furthermore, there are some differences that we need to fix. For example, there are two different quotations in the gold standard (`''` and ``` `` ```), whereas there is only one in the raw text (`"`). To make the data comparable we must translate the gold standard quotes into the ones found in raw text. You can use these two functions to get the raw and gold corpus:

```
def get_corpus_text(nr_files=199):
    """Returns the raw corpus as a long string.
    'nr_files' says how much of the corpus is returned;
    default is 199, which is the whole corpus.
    """
    fileids = nltk.corpus.treebank_raw.fileids()[:nr_files]
    corpus_text = nltk.corpus.treebank_raw.raw(fileids)
    # Get rid of the ".START" text in the beginning of each file:
    corpus_text = corpus_text.replace(".START", "")
    return corpus_text

def fix_treebank_tokens(tokens):
    """Replace tokens so that they are similar to the raw corpus text."""
    return [token.replace("''", '"').replace("``", '"').replace(r"\/", "/")
            for token in tokens]

def get_gold_tokens(nr_files=199):
    """Returns the gold corpus as a list of strings.
    'nr_files' says how much of the corpus is returned;
    default is 199, which is the whole corpus.
    """
    fileids = nltk.corpus.treebank_chunk.fileids()[:nr_files]
    gold_tokens = nltk.corpus.treebank_chunk.words(fileids)
    return fix_treebank_tokens(gold_tokens)
```

Tokenize the corpus

Create a function that tokenizes a given text:

```
def tokenize_corpus(text):
    """Don't forget the docstring!"""
    (...)
    return tokens
```

Implement your own tokenizer and apply Regular Experssion on it (**DON'T** use NLTK tokenizer**)**. See the documentation of the python module re for more information about regular expressions. Use the following evaluation function to test the result to the gold standard tokenization:

```
def evaluate_tokenization(test_tokens, gold_tokens):
    """Finds the chunks where test_tokens differs from gold_tokens.
    Prints the errors and calculates similarity measures.
    """
    import difflib
    matcher = difflib.SequenceMatcher()
    matcher.set_seqs(test_tokens, gold_tokens)
    error_chunks = true_positives = false_positives = false_negatives = 0
    print(" Token%30s  |  %-30sToken" % ("Error", "Correct"))
    print("-" * 38 + "+" + "-" * 38)
    for difftype, test_from, test_to, gold_from, gold_to in
matcher.get_opcodes():
        if difftype == "equal":
            true_positives += test_to - test_from
        else:
            false_positives += test_to - test_from
            false_negatives += gold_to - gold_from
            error_chunks += 1
            test_chunk = " ".join(test_tokens[test_from:test_to])
            gold_chunk = " ".join(gold_tokens[gold_from:gold_to])
            print("%6d%30s  |  %-30s%d" % (test_from, test_chunk,
gold_chunk, gold_from))
    precision = 1.0 * true_positives / (true_positives + false_positives)
    recall = 1.0 * true_positives / (true_positives + false_negatives)
    fscore = 2.0 * precision * recall / (precision + recall)
    print()
    print("Test size: %5d tokens" % len(test_tokens))
    print("Gold size: %5d tokens" % len(gold_tokens))
    print("Nr errors: %5d chunks" % error_chunks)
    print("Precision: %5.2f %%" % (100 * precision))
    print("Recall:    %5.2f %%" % (100 * recall))
    print("F-score:   %5.2f %%" % (100 * fscore))
    print()
```

Run the tokenizer and evaluate. Look through the report and make appropriate changes to your regexp. Iterate this until you are satisfied. Note that you will not be 100% correct since this is virtually impossible! (But you should at least be able to raise above 98%). Start with a small `nr_files` such as 20. When you have completed that, you can increase the size to 50, then 100, and finally 199.

To make life easier, you can do something like this at the end of your file when you are working on your regexp:

```
if __name__ == "__main__":
    nr_files = 20
    corpus_text = get_corpus_text(nr_files)
    gold_tokens = get_gold_tokens(nr_files)
    tokens = tokenize_corpus(corpus_text)
    evaluate_tokenization(tokens, gold_tokens)
```

Corpus statistics

Use the tokenized corpus to answer the following questions:

1. How big is the corpus in terms of the number of word tokens and in terms of word types?
2. What is the average word token length?
3. What is the longest word length and what words have that length?
4. How many hapax words are there? How many percent of the corpus do they represent? (For this question and the questions below, by corpus size, we mean the number of tokens)
5. Which are the 10 most frequent words? How many percent of the corpus do they represent?
6. Divide the corpus in 10 slices of equal sizes: s[0]...s[9].
    - How many hapaxes are there in each of the slices in terms of percentage of the subcorpus?
    - Now look at subcorpora of increasing size: s[0], s[0]+s[1], s[0]+s[1]+s[2], and so on, until you have reconstructed the complete corpus. How many hapaxes are there in each of these subcorpora? How much is it in each case in terms of a percentage of the subcorpus?
7. Draw the results from question 6 in a graph.
8. How many unique word bigrams are there in the corpus? How many percent do they represent of all bigrams?
9. How many unique trigrams are there? How many percent of all trigrams do they represent?

Implemented each of these questions as a function that takes the corpus as its argument and returns the answer:

```
def nr_corpus_words(corpus):
    """Don't forget to docstring me!"""
    nr_of_corpus_words = (...)
    return nr_of_corpus_words
```

Write a function that takes the tokenized corpus as its argument and prints in terminal its statistics:

```
def corpus_statistics(corpus):
    """Docstring, docstring, docstring!"""
    do_some_calculations
    print("Here is an answer: %5.2f %%" % (100.0 * nr_occurrences /
total_nr))
    print("And here is another: %s" % (", ".join(a_list_of_strings)))
    (...)
```

Finally, make the Python file callable from the command line so that it first tokenizes the raw corpus, prints the error report and prints the answers to the questions.

What is a word?

We also count punctuation symbols as words. Hence, in the following example there are 16 words and 8 word forms:

En såg såg en såg en såg såg, en annan sågade sågen sågen såg.

Here are the word forms:

, . En annan en såg sågade sågen

"En" and "en" count as different word forms, but "såg" (verb) and "såg" (noun) are taken as the same word. "," and "." are also words and word forms in this sense. The most common words are "såg" (6 occurrences), "en" (3), and "sågen" (2).

The bigrams in this corpus are (En såg), (såg såg), (såg en), (en såg), (såg en), (en såg), (såg såg), (såg ,), (, en), (en annan), (annan sågade), (sågade sågen), (sågen sågen), (sågen såg), (såg .), hence there 15 of them. But since some of them occur more than once (en såg, såg en, såg såg), the number of unique bigrams is 12. The number of possible bigrams is (the number of word forms)^2 = 8x8 = 64.

Hemanth Kumar Battula , 27 Nov 2018 23:29

File name: lab1-battula.py (12,1 KB)

Status set to: To be marked

Chatrine Qwaider , 8 Dec 2018 19:26

Status set to: Completed

Chatrine Qwaider , 18 Jan 2019 09:21

Grade set to: VG

Chatrine Qwaider , 29 Jan 2019 13:40

Mark set to: 7