Natural Language Processing

Lab 2: Tokenization and Sentence Segmentation

---

1 Introduction

In this lab, we are going to refine a tokenizer to handle phenomena such as punctuation, abbreviations, contractions, and words containing non-alphabetic characters. We will also add a simple mechanism for sentence segmentation. For development of the tokenizer, we use the same text as in Lab 1 when we manually screened and corrected the output of the simple tokenizer. As soon as we are satisfied with the performance of the tokenizer on this text, we will then evaluate it on a previously unseen (but similar) text. All the files needed for the lab are in `/local/kurs/nlp/basic2/`. Again, start by making copies into your home directory (see Lab 1 for details).

2 The gold standard

The file `dev1-gold.txt` contains the correct tokenization of the text in `dev1-raw.txt` according to the widely adopted Penn Treebank standard. Have a look at the tokenized text and make sure that you understand the principles by which it has been tokenized. Most of the tokens are either regular words or punctuation marks, but note the following:

- Numerical expressions like `3.8` are treated as single tokens.
- Logograms like `%` and `$` are treated as separate tokens.
- Contractions like `it's` and `don't` are split into two tokens – but how?
- Straight quotation marks (`"`) have been changed to opening (`` ` ` ``) or closing (`''`) quotation marks.

3 A pattern-matching tokenizer

The file `tokenizer1.py` contains a very simple tokenizer:

```
import re, sys

regex_pattern = "([,;:.!?\"]|\w+)"
for line in sys.stdin:
    for token in re.findall(regex_pattern,line.strip()):
        print(token)
```

The main difference compared to the whitespace tokenizer used in Lab 1 is that we are now trying to define what a valid token looks like, instead of just using whitespace to find token boundaries. Therefore, we use the Python method `re.findall` instead of `re.split`. This method takes as arguments a regular expression and a string, and returns a list of all the matches found in the string. The matching is done from left to right and greedily tries to find the largest possible match each time. Let us consider the regular expression used in `tokenizer1.py`:

```
"([,;:.!?\"]|\w+)"
```

The regular expression, enclosed in double quotes (`"`), is a simple union expression, where the first part is a character set containing the most common punctuation marks (`[,;:.!?\"]`), and the second part is an expression (`\w+`) matching any non-empty sequence of alpha-numeric characters. Note that the double quotes must be escaped in the set of punctuation marks. Why?

4 Refine the tokenizer

Run `tokenizer1.py` on `dev1-raw.txt`, and compare the output to `dev1-gold.txt` using the Linux command `diff`:

```
python tokenizer1.py < dev1-raw.txt > dev1-tok.txt

diff dev1-gold.txt dev1-tok.txt
```

Note: For better visualisation, you may want to try using vimdiff:
`vimdiff dev1-gold.txt dev1-tok.txt`
Note that vimdiff opens an editor which has different key bindings than you may be used to. To quit, press :q.

This gives you a list of the problems you have to fix. The left column corresponds to the gold standard token and the right column corresponds to the token given by the tokenizer. Your task is to modify the regular expression used for matching in order to eliminate as many of the problems as possible. A good strategy is to concentrate on one type of token at a time, and add a new subexpression to the disjunction, specifically designed to handle that token type. Note that Python processes the regular expression from left to right, which means that earlier subexpressions will take priority over later subexpressions if they have overlapping matches. (The two subexpressions in the original expression never have overlapping matches. Why?) If you need help, consult the Python documentation at `https://docs.python.org/3/howto/regex.html`. If you want to measure the exact precision and recall on the token level, you can use `score-tokens.py` as follows:

```
python score-tokens.py dev1-gold.txt dev1-tok.txt
```

Note on quotation marks: In order to match the gold tokenization exactly, you would not only have to segment the text correctly but in addition replace any token of the form (`"`) by either (`` ` ` ``) or (`' '`). For the purpose of this lab, you can ignore any errors that only concern the form of quotation marks, which is also what the evaluation script `score-tokens.py` does.

Note on Python regular expression matching: The original regular expression is surrounded by a single pair of round brackets, and each match returned by `findall` (in the variable `token`) is a string matching the entire regular expression. When you refine the expression, you may have to add internal brackets to get the right grouping of subexpressions. This will change the behavior of the program and each match will now return a tuple of strings matching the different bracketings, where the first element is always the string matching the entire regular expression. In this case, you therefore have to change `print(token)` to `print(token[0])` to get the right output.

5 Add sentence segmentation

The final task is to add sentence segmentation to the tokenization process. In general, this is a rather hard problem, but for the sentences in `dev1-raw.txt` it should not be too difficult to come up with something reasonable. It is customary to mark sentence boundaries by a blank line, so you just need to come up with rules for inserting a blank line after each sentence (including the last one). When you have done this, you can compare your output to `dev1-gold-sent.txt`, which is identical to `dev1-gold.txt` except that sentence boundaries have been inserted.

Note on sentence-final abbreviations: When an abbreviation such as `e.g.` occurs at the end of a sentence, orthographic conventions prescribe that there is no final stop (`.`) to end the sentence. According to the Penn Treebank standard, however, the final stop should be restored as part of the sentence segmentation process, which means that an additional token is inserted in this case. For the purpose of this lab, you can ignore this complication.

Try to reach at least 95% precision and recall.