Natural Language Processing

Lab 1: Simple Tokenization

1 Introduction

In many languages, word boundaries are marked by spaces. However, relying on whitespace alone seldom provides adequate tokenization. In this lab, you will inspect the output of a tokenizer that only segments tokens separated by whitespace, discuss the shortcomings of this approach, and start to think about how they can be overcome. After that, you will improve another tokenizer, using regular expressions.

Start by making a sound file structure for the course. For example, create a directory called `NLP_5LN701` with a subdirectory `labs`, and a subdirectory in `labs` called `lab01`. Then copy the files for today's lab to `lab01`. This is because the work must be done under your own home directory. For example:

`/home/user1234/NLP/labs/lab01/`

All the files needed for the lab can be found on Futurum by following this path:

`/common/student/courses/NLP-5LN701/labs/lab01`

2 Tokenize some text

The file `nlp5LN701tokenizer0.py` contains the simplest possible tokenizer — replace the comments with adequate Python code.

```
import re, sys

# Define a function that takes as input a filepath and that returns
#the lines of that file. Use the method .readlines()

def main():
    # Add code for a function call here. Pass in sys.argv[1] to the function.
    for line in lines:
        for token in re.split("\s+", line.strip()):
            print(token)

main()
```

The first line imports the `re` module (for regular expressions) and the `sys` module (for reading from stdin). The main function consists of two nested `for` loops: For each line in the input file, split the line into tokens at (one or more) whitespace characters (`\s+`), and print each token. To run the tokenizer on the file `dev1-raw.txt` and print the result to the (new) file `dev0-tok.txt` by making use of the redirection arrow.

Do you need help with the command? Here it is:

- `python3` will be the program
- The arguments to that program will be: `nlp5LN701tokenizer0.py` and `dev1-raw.txt`
- The redirection arrow > tells the Terminal not to print to the standard output but to redirect the output to somewhere else.
- Where is the somewhere else? Well, it's the file `dev0-tok.txt` that, in other words, will be created.

When you have done this, check that `dev0-tok.txt` contains a sequence of tokens, one on each line. Use any command you see fit (e.g., `cat`, `head`, `less`, `vim`, `nano`, etc.) Use `man` if you are unsure how, or any cheat-sheet from the course Programming for Language Technologists.

2.1 Analyze the result

Identify problems in the tokenization. Use your intuition. What units are convenient for further linguistic analysis? Find arguments for why a particular segmentation may lead to problems, for example, when we perform a grammatical analysis or try to look up words in a lexicon. Most problems you encounter will be **either** cases of *oversplitting*, where the tokenizer has split what should have been one token into several tokens, **or** *undersplitting*, where the tokenizer has failed to split a string into multiple tokens. Try to categorize different cases of oversplitting and undersplitting. How could they be eliminated?

2.2 Correct the tokenization

When you have identified the tokenization errors, you should correct them manually using a text editor. You should correct at least the first ten lines of the text in `dev1-raw.txt` (ending just before the word "Pacific"). Compare with some of your classmates: do you agree on how to tokenize the text?

3 Refinement

We are now going to refine a tokenizer to handle phenomena such as punctuation, abbreviations, contractions, and words containing non-alphabetic characters.

$We will also add a simple mechanism for sentence segmentation.$

3.1 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.

3.2 A pattern-matching tokenizer

The file `nlp5LN701tokenizer1.py` contains a very simple tokenizer:

```
import re, sys

# Add code here as you did in nlp5LN701tokenizer0.py

def main():
    # And here
    pattern = "[,;:.!?\"]|\w+"
    for line in lines:
        tokens = re.findall(pattern, line.strip())
        for token in tokens:
            print(token)

if __name__ == "__main__":
    main()
```

The main difference compared to the whitespace tokenizer 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()`. The method `re.findall()` 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 `nlp5LN701tokenizer1.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?

3.3 Punctuation handling in tokenization

Breaking off punctuation as separate tokens while preserving word-internal punctuation is a common task in natural language processing (NLP) and text processing. Use basic regular expressions to keep the punctuation that occurs internally within words, like:

- Abbreviations, e.g. U.S.A.
- Words that may have internal hyphens, such as mother-in-law
- Monetary values and percentages, like $15.75 or 75.5%
- Ellipses, such as '...'

3.4 Refine the tokenizer

Run `nlp5LN701tokenizer1.py` on `dev1-raw.txt` and redirect the output to a new file. Use the name `dev1-tok.txt`. In other words, the command is `python3` and its arguments are `nlp5LN701tokenizer1.py` and `dev1-raw.txt`.

Compare the output to `dev1-gold.txt` using the Linux command `diff`:

```
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`

If you have not used `vim`: 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?)[1]

3.5 Measure precision, recall and F-score:

To measure the exact precision, recall and F_1-score on the token level, use `score-tokens.py` as follows shown below. Make sure you understand what is the command and what the arguments are as far as the Terminal is concerned.

```
python3 score-tokens.py dev1-gold.txt dev1-tok.txt
```

As you can see when you run this, something is missing – you are not getting the F_1-score. To fix this, write code for a function `f_score()` in a Python file called `f_score.py`. The function should take two parameters, precision and recall. F_1-score in this case is the harmonic mean of precision and recall. To test whether your function is correct, you can pass in 89 and 111, which should result in 98.79. Here is the formula that you will turn into code:

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{1}$$

For the tokenizer, try to get at $precision$, $recall$ and F_1-score to at least 0.95

[1] If you need help consult the Python documentation at:
`https://docs.python3.org/3/howto/regex.html`

3.6 Notes

A 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.

A 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.

4 Using `egrep`

The Linux program `egrep` (see `man egrep`, use the arrows to move down, and press `Q` to quit) is a powerful — and **extremely useful** — tool for searching in text.[2] The command `egrep` allows you to use regular expressions. In the following, you will use the regular expressions that you used in Python. In this way, you can tokenize the same file but with `egrep` instead of with Python. Explore as much as you want, and compare the results to what you achieved when using Python. Optimally, you will get the same results. Remember to use the redirection arrow and to name the output file to something different than before.

Bellow are a few introductory steps. The single quotation marks ' are incorrectly formatted in this PDF, so don't copy them from here. In the Terminal, an quotation marks should appear as a single straight apostrophe.

```
egrep 'the' dev1-raw.txt
```

If you run this command, you will see a wall of text, feel free to be surprised! This is how `egrep` works: It returns the **lines** on which the matches are found, if any. Try this instead:

```
egrep 'the' dev1-raw.txt --color
```

This time, you hopefully see how it is working. However, notice `egrep` also found things like **the**se, **the**y and ano**ther**. Can we avoid such cases? Try the previous command on the file `the.txt`. Remember to use `--color`.

As you can see, three instances of the string **the** were returned, but only two seem relevant (unless we were looking for any string containing **the**). Let's say we wanted to find only **the** (and not things like **they**)? What if we could tweak the command `egrep` to better suit our needs? It turns out we can. Use the manual `man` to find a suitable **flag**. What is a flag? Flag are additional options to the commands. They might look the same (dash - followed by a single character) for some commands, but often they don't mean the same thing but are unique to each command.

Before giving you the answer to what you need for `egrep` to avoid things like **they**, let's check out a few more flags for other commands.

- `wc -l file_name.txt`
- `cp -i file1.txt file2.txt`
- `mv -n file1.txt file2.txt`
- `ls -a`
- `sort -f file1.txt`

[2] You can also use `grep` only, which is almost the same. The **e** stands for *extended*.

Did the flags make sense? Remember you can use `man` when you forget what a flag does. Now let's turn back to `egrep` and find out what flag is relevant. Try this command:

```
egrep -w 'the' the.txt --color
```

It should match only whole words. Now let's combine it with another flag, which can be done in two equivalent ways:

```
egrep -wo 'the' the.txt --color
```

```
egrep -w -o 'the' the.txt --color
```

What does the flag `-o` do for egrep? Use the manual. Now you should be ready to start using your regexp knowledge that you attained in the earlier parts of this lab and apply it to what you have just learned about `egrep`!

5 Extras

If you are done with everything and feel you want more, don't dispair! Here is a cool **command chain**. A command chain can be recognized by the pipe `|`, which tells the terminal something like: "The command to the left of the pipe gave some output that I want the command to the right of the pipe to take as input." In other words, you can combine commands! Here is a simple example:

```
egrep -o '\w+' dev1-raw.txt | egrep -w -c 'the'
```

The flag `-o` says only the matched parts of a matching line should be printed, with each match on a separate line. The regexp `\w+` matches any alpha-numeric character (including underscore). The pipe `|` feeds the output of `egrep -o '\w+' dev1-raw.txt` to the command `egrep -wc 'the'`. The flag `-w` you have already seen, and the flag `-c` counts the matches. To check that the count is correct, try the same command but on the file `the.txt`.

Here is another useful command chain to know:

```
egrep -o '\w+' dev1-raw.txt | sort | uniq -c | sort -rn | head
```

Try to figure out all the parts yourself. **Hint:** Start with the leftmost command and add one link in the chain at a time.