

# Natural Language Processing – Assignment I

Reykjavik University – School of Computer Science

September 2010

## 1 Part I – 16%

In this part, you experiment with the corpus *eng.sent*<sup>1</sup>. This corpus contains one English sentence per line where each word is tagged with a part-of-speech (PoS) tag. The tagset used is the Penn Treebank tagset (see [http://www.ling.upenn.edu/courses/Fall\\_2003/ling001/penn\\_treebank\\_pos.html](http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html)).

**For each item below, show the exact command that you use.**

1. (10%) Use *grep* or *egrep* to display information from *eng.sent* that match certain patterns. Note: Chapter 2.3.4 in the text book might help as well as the *man* page for *grep* or *egrep*.

- (a) Display all sentences that include the exact word forms “German” and “car” (not necessarily adjacent, but in that order).

```
grep "German .*car " eng.sent
```

- (b) Display the number of sentences that include the exact word form “tennis”.

```
grep -c "\btennis " eng.sent
```

```
or
```

```
grep -c -w "tennis" eng.sent
```

- (c) Display plural English nouns (not proper nouns) of length 12. Here we don’t want the whole line, only the nouns in question.

```
egrep -o "\b[A-Za-z]{12} NNS" eng.sent
```

```
or
```

---

<sup>1</sup>This is a Reuters corpus, available from <http://www.ru.is/faculty/hrafn/Data/eng.zip>. Due to copyright reasons, please make sure you do not distribute this corpus.

```
egrep -o "(^| ) [A-Za-z]{12} NNS" eng.sent
```

```
egrep -o "\b[^\ ]{12} NNS" eng.sent  
gives a little bit more results,  
because it does allow characters not in A-Za-z
```

- (d) Display all sentences that start with a Wh-pronoun.

```
egrep "^[^\ ]+ WP " eng.sent
```

- (e) Display all sentences that include the exact word form “offer”, either as a verb or as a noun, and either preceded by a word tagged as “DT” or a word tagged as “TO”.

```
egrep "^[^\ ]+ (TO|DT) offer (VB|NN) " eng.sent
```

2. (16%) Use *sed* to perform the following tasks on *eng.sent* and write the output to *eng.out*. Note: The web page <http://www.grymoire.com/Unix/Sed.html> might help.

- (a) Substitute a number at **the beginning** of a line with the number itself surrounded by brackets (e.g. 123 becomes [123]).

```
sed 's/^[0-9][0-9]*/[&]/' < eng.sent > eng.out
```

- (b) Make each <token,tag> pair appear on a separate line. For example, the first three lines in *eng.out* should appear as:

```
EU NNP  
rejects VBZ  
German JJ
```

```
sed 's/[^\ ]* [^\ ]* /&\n/g' < eng.sent > eng.out
```

- (c) Remove the PoS-tags such that only the tokens remain. For example, the first line in *eng.out* should look like this:

```
EU rejects German call to boycott British lamb .
```

```
sed 's/\([^\ ]*\ \)[^\ ]* /\1/g' < eng.sent > eng.out
```

## 2 Part II – 34%

This part consists of two Perl programs. Note that the amount of code that you have to write is about 20 lines (or less) for each of the two programs.

1. (14%) Write a Perl program, *findLongest.pl*, accepting a text file as an argument. The program finds the longest word, that only contains lower case English letters<sup>2</sup>, in the file and prints it out along with its length. Note that the program should work for any text file, regardless of its format (i.e. whether the file has one word per line or multiple words per line). To invoke the program the user should type in:

```
perl findLongest.pl <filename>
```

- Test your program on the corpus *eng.sent* and return your program code along with the output of your program when running against this corpus.

Longest word found is characteristically of length 18

```
# @ARGV contains the arguments
open(FILE, "$ARGV[0]") || die "Could not open file $ARGV[0].";

$maxLen = 0;           # the length of the longest word found so far.
$longest_word = "";   # the longest word found so far.

# The while-loop reads the input file line by line.
while (<FILE>) {
    @array = split; # Here we split the line (on space using the $_ variable)
    foreach $word (@array) {
        if ($word =~ m/^[a-z]+$/) { # If the current word contains only lower case
            $len = length($word);
            if ($len > $maxLen) { # If a longer word is found
                $longest_word = $word; # store the longest word
                $maxLen = $len; # and the longest length
            }
        }
    }
}
# Print the result.
print "Longest word found is $longest_word of length $maxLen\n";
close(FILE);
```

---

<sup>2</sup>Use a regular expression for enforcing this condition.

2. (20%) Write a Perl program, *tagFrequency.pl*, accepting a tagged corpus as an argument (one word/tag per line) and a number *t*. The program prints out, in descending order, the *t* tags occurring most often in the corpus. Hint: Consult, for example, <http://www.perlfect.com/articles/sorting.shtml> for a discussion on Perl sorting.

- Test your program on the corpus *eng.train* (which is included in the *eng.zip* file) and return your program code along with the output of your program when running against this corpus using the number *10*, i.e.

```
perl tagFrequency.pl eng.train 10
```

```
use warnings;
# Sort definition subroutine.
sub by_number { $hash_freq{$b} <=> $hash_freq{$a} }

# Main program starts here
# This program takes a corpus file and number t as input and
# prints out the t tags in the corpus with highest frequency

($input,$number) = shift(@ARGV);
open(INFILE, "$input");

# Loop through the corpus file
while ($line = <INFILE>) {
    if ($line !~ m/^\s+$/) { # don't process empty lines
        # split the line using space as a delimiter
        @words = split /\s+/, $line;
        $tag = $words[1];    # get the tag
        $hash_freq{$tag}++; # increment the hash table for the tag
    }
}

# Output
$count=1;
foreach $key (sort by_number keys %hash_freq) {
    if ($count > $number) {last;}
    $value = $hash_freq{$key};
    print "$key => $value\n";
    $count++;
}
```

```
close(INFILE);
```

Test:

```
perl tagFrequency.pl eng.train 10
```

```
NNP => 34392
NN  => 23899
CD  => 19704
IN  => 19064
DT  => 13453
JJ  => 11831
NNS => 9903
VBD => 8293
.   => 7389
,   => 7291
```

### 3 Part III - 38%

In this part, you develop a *tokeniser* and a *sentence segmentiser* for a language of your choice. Indeed, it would be best if your program could handle a family of languages (instead of a single language), for example some of the Germanic languages like English, German, Icelandic, Danish and Swedish. With language independence in mind, regular expressions for matching abbreviations are preferred to an implementation using a list.

You should use **JFlex** and/or **Perl** as implementation languages. Note that it may be appropriate to use both JFlex and Perl. One might, for example, use JFlex to do “simple” tokenisation and then read its output into a Perl program which does more complicated tokenisation/sentence segmentation, not easily solved by using regular expressions.

Hence, you could implement your solution by constructing a set of units running in a sequence. In that case, each unit reads as input the output from the preceding unit in the sequence. However, the exact division of units should be hidden from the user, i.e he/she should only need to enter a command like:

```
tokenise input.txt output.txt
```

where *input.txt* is the input file and *output.txt* is the corresponding tokenised (and sentence segmentised) output text.

This project is in two parts, described below in Sections 3.1 and 3.2.

### 3.1 Word tokeniser

- The input to this program is a text file with a “free format”, i.e. the file can contain one sentence per line, many sentences per line, one token per line, several tokens per line, etc.
- The output should be one token per line.

### 3.2 Sentence segmentiser

- The input to this program is a text file with one token per line, i.e. the input is a file generated by the word tokeniser.
- The output is one token per line along with empty lines denoting sentence boundaries.

### 3.3 Testing and what to return

Your goal should be to develop a tokeniser/segmentiser which is reasonably accurate, but it does not have to be perfect (which indeed is a difficult task!).

For testing you should gather text in your own language (and related languages) from the web, for example, text from newspapers, personal pages, university pages, etc. The text should contain at least 10,000 tokens and make sure that it contains **different types of tokens**, e.g. words, personal names, numbers and abbreviations.

You need to return your program code, your test file and the output generated by your tokeniser when processing the test file.

**A solution is not provided for this part.**

## 4 Part IV - 12%

In this part, you develop an English trigram language model based on *eng.sent* by only using Unix/Linux tools (not Perl) as discussed in the

lecture on N-grams. Note that here we are only interested in word (token) trigrams (including punctuations), not PoS trigrams.

- (a) *eng.sent* is pre-tokenised even though the <token,tag> pairs do not appear on a separate line. However, in order to construct the language model you need a file with one token (word) per line without any empty lines. Show the sequence of command that you use for constructing this file, *eng.tok*.

```
# Remove the PoS tags and make sure we have only
# one token per line without any empty lines
sed 's/\([^ ]* \)[^ ]* /\1\n/g' < eng.sent | sed '/^$/d' > eng.tok
```

- (b) Now show the sequence of commands you use to construct a trigram frequency file *engTri.freq* (from *eng.tok*), sorted in descending order of frequency.

```
# Now construct the language model, a trigram frequency file
tail --lines=+2 < eng.tok > eng2.tok
tail --lines=+3 < eng.tok > eng3.tok
paste eng.tok eng2.tok eng3.tok > engTri.tok
sort engTri.tok | uniq -c | sort -nr > engTri.freq
```

- (c) Finally, show the command you use for displaying the 10 most frequent trigrams from *engTri.freq*, as well as the output from this command.

```
# Finally, show the 10 most frequent trigrams
head -10 engTri.freq
```