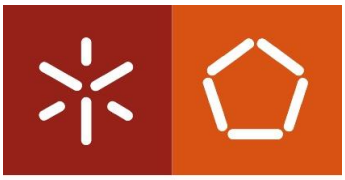

Universidade do Minho

Departamento de Informática

Processing the CETEMPublico's corpus

Pedro Mendes (a79003)

July 8, 2019

Abstract

This project's goal is to analyse the *CETEMPublico*'s corpus, compiled from publications of the *Publico* newspaper [**?**]. This document's purpose is to assist researchers in the development of software that needs to process the Portuguese language, as well as, anyone studying the Portuguese language.
Simple examples of such processing are demonstrated in this paper.

Contents

Chapter 1

Introduction

This project aims to parse the *CETEMPublico*'s corpus to produce some statistics using the *gawk* programming language. *Gawk* is a domain specific language designed for parsing CSV and CSV-like files making it a very competent choice for the proposed problem, it does have it's limitations, as will be shown in the on the last chapter.

First we'll analyse the problem, see what needs to be implemented and what challenges need to be overcome to implement said features.

Next we'll look at the solutions to the proposed problems, dedicating a section to the solution of each problem.

Chapter 2

Problem

The following analysis must be made:

- Count the number of sentences, paragraphs, extracts and multi word expressions.
- Find and count the multi word expressions (MWE).
- Find and count the verbs.
- Build the implicit dictionary in the corpus.

The input file uses XML tags to annotate it's contents, therefore the gawk scripts will have to look for these during parsing.

Multi Word Expressions span more then one line, thus the line based parsing will have to be adapted.

As is usual with these kinds of file processing tasks, the input is millions of lines long which means that care must be taken to not accumulate too much data in program memory.

Chapter 3

Solution

3.1 Requirement 1: Counting

The first requirement is relatively simple, the corpus contains a tag for each of the items that need to be counted, therefore my job was simply to count these tags.

```
BEGIN { FS="\t" }
$1 ~ /<s[^>]*>/ { s++ }
$1 ~ /<p[^>]*>/ { p++ }
$1 ~ /<ext[^>]*>/ { e++ }
$1 ~ /<mwe[^>]*>/ { mwe++ }
END {
    print "#Extracts:\t"e
    print "#Paragraphs:\t"p
    print "#Sentences:\t"s
    print "#MWExpressions:\t"mwe
}
```

Figure 3.1: **count.awk:** Gawk script to count sentences, paragraphs, extracts and MWE

As demonstrated in Figure 3.6 each tag has a distinct regular expression to capture it which maps to a simple counting action. At the end the program simply prints the number of occurrences of each tag.

3.2 Requirement 2: Multi Word Expressions

The second requirement entails counting the MWEs in the file. Because each MWE uses more than one line there needs to be a variable tracking the context the parser is in.

```
BEGIN { FS="\t"; mwe = 0 }
$1 ~ /<mwe[^>]*>/               { mwe = 1; current = "" }
$1 ~ /<\/mwe[^>]*>/             { mwe = 0; expressions[current]++ }
mwe == 1 && $1 !~ /<mwe[^>]*>/ { current = current " " $1 }
END { for(e in expressions) { print expressions[e] "\t" e } }
```

Figure 3.2: **mwe.awk:** Gawk script to find and count MWEs

As can be seen the `mwe` variable tracks whether the parser is inside a MWE. In addition to this, the `current` variable accumulates the MWE to be stored in the `expressions` associative array for counting. The script finishes by printing every expression along with it's occurrence count.

3.3 Requirement 3: Verbs

Verbs in the corpus are distinguished by the fifth column, if it's value starts with a `V` then the associated word is a verb. Afterwords the script stores the field in the forth column since this is the verb in it's infinitive form, as opposed to the conjugated form that is in the first field.

```
BEGIN { FS="\t" }
$5 ~ /^V/ { verbs[tolower($4)]++ }
END { for(v in verbs) { print verbs[v]"\t"v } }
```

Figure 3.3: **verbs.awk:** Gawk script to find and count all verbs in their infinitive form

The script finishes by printing out all the verbs and their occurrences.

3.4 Requirement 4: Dictionary

A dictionary in this context is list of words group with their lemmas. The lemmas, on the other hand, have associated with them their respective part of speech.

```
$0 !~ /^</ && $4 !~ /[0-9¡ª´¹º©«».,&?;:'"{}=_~|`!@#$%^*()+\-\/\[\]\\]/ {
    word = toupper(substr($4, 1, 1)) substr($4, 2)
    dict[word][$1][$5]++
}
```

Figure 3.4: Excerpt of a gawk script to filter and store valid words

The script used uses a single filter to obtain all the valid words (see Figure 3.4). This filter will first remove lines started by XML tags, and second filter any words that include punctuation, numbers, and other non-letter symbols. Then in the action part of this filter the first letter is capitalized and inserted into the dictionary associating the lemma found (`$1`) and the lemma's part of speech (`$5`).

For the `END` action, there are two versions of the script: One that outputs a JSON string and one that creates a series of html files to make browsing the dictionary easier.

3.4.1 JSON

The JSON string output by this script groups words with their lemmas and the lemmas with their parts of speech. It aims to be more or less direct translation of the way they are stored in memory (see Figure 3.4).

In the example in Figure 3.5 we can see represented an object containing a dictionary which contains a words field with the list of words, each one containing a list of lemmas. Each lemma with a list of parts of speech it was found in. This is a more or less direct translation of the way the dictionary is stored in memory.

```
{
  "dictionary": {
    "words": [
      {
        "word": "Avançado",
        "lemas": [
          {"avançados": [
            "ADJ"
          ]},
          {"avançada": [
            "ADJ"
          ]},
          {"avançado": [
            "N",
            "ADJ"
          ]}
        ]
      },
```

Figure 3.5: Excerpt of the output JSON string

3.4.2 HTML

The second version of the script outputs the dictionary to html files with the following organization. An *index.html* file that lists all the words in the dictionary as links to other *word.html* files that list the different lemmas and their parts of speech.

The list in the index file is ordered alphabetically and includes links at the top to jump the different letters of the alphabet.

This implementation, however, is very slow due to the sorting used by *gawk* not being intended for use case. For example, to process the largest file provided the script takes 40 minutes to complete. To solve this issue a implementation of the sorting and generation of the html files was made in Rust a system's programming language that has performance comparable to that of *C* and *C++*. With this the time it takes to process the largest file provided was cut to 30 seconds.

To achieve this the *gawk* script opens a named pipe and then starts the Rust program in the background, setting it up to read from the pipe, afterwords it simply writes the filtered and valid words to the pipe. On the other side, the Rust code is very similar to the previous, *gawk* only code, keeping the same data structure, only using a binary tree to store the words instead of an associative array.

Both versions of the program were kept for evaluation purposes.

```
BEGIN {
    FS = "\t"
    system("mkfifo pipe")
    system("./dictionary_html_fast pipe &")
}
$0 !~ /^</ && $4 !~ /[0-9¡ª´¹º@«».,&?;:'"{}=_~|`!@#$%^*()+\-\/\[\]\\]/ {
    word = toupper(substr($4, 1, 1)) substr($4, 2)
    print word"\t"$1"\t"$5 >> "pipe"
}
END { system("rm pipe") }
```

Figure 3.6: **dictionary_html_fast.awk:** Gawk script to build a dictionary out of the corpus, using a second program to process the filtered words

Chapter 4

Listing of the produced files

- **count.awk** Explained in Section 3.1
- **mwe.awk** Explained in Section 3.2
- **verbs.awk** Explained in Section 3.3
- **dictionary_json.awk** Explained in Subsection 3.4.1
- **dictionary_html.awk** and **dictionary_html_fast.awk** Explained in Subsection 3.4.2

Chapter 5

Conclusion

In conclusion, *gawk* is a very powerful tool for filtering and reorganizing text that gets a lot of work done in very few lines of code. On the other hand it is not intended for complex data processing and reorganization. Because of this it is my opinion that when using it one has to know it's limits or risk wasting more time then needed.

The tool also fits incredibly well with the rest of the Unix system ecosystem, reading from the standard input by default and writing to the standard output, it can be used to process in conjunction with other common tools through the use of pipes.

In regards to the work done in this assignment, most tasks were relatively easy to complete due to *gawk*'s powerful syntax, only ever falling short on the aforementioned performance and complex processing, which the tool was never meant to be used for in the first place.