CAB203 Problem solving

Shridhar Thorat: n10817239

June 17, 2022

1 Regular languages and finite state automata

In order to censor the articles: '*a*', '*an*' and '*the*', and append a student's number in their emails, a Python function **censor(s)** was developed. This function applies the required censoring to a string **s**, replacing each letter in an article with the pound sign: #, and appends a student number (including a single whitespace) if the string was censored.

This function took advantage of the Python package **re**, which provides regular expression syntax and functions.

Syntax and patterns:

The syntax used were; \b and ( ). The syntax \b defines the beginning and ending of a word boundary, while ( ) defines a capture group.

The regex patterns for the three articles in Python were chosen as below. Where *r* simply defines a raw string in Python.

- *r*"\b(a)\b" only matches the word "*a*"
- *r*"\b(an)\b" only matches the word "*an*"
- *r*"\b(the)\b" only matches the word "*the*"

These regex patterns are checked in the string s from left to right. The patterns can be interpreted as below.

- The first \b sets the starting word boundary (Foundation 2022).
- The article in () defines the character capture group. Where only characters in this group are matched (Foundation 2022).
- The last \b sets the end of the word boundary (Foundation 2022).

Thus the patterns only match characters in the capture group - and thus, the articles. However, the articles must be matched regardless of capitalisation - which isn't done by the pattern. Instead of modifying the pattern, the flag re.IGNORECASE was used in the regular functions.

Flags used:

The flag re.IGNORECASE (re.I) was used as a function parameter. As the name suggests, it performs case-*insensitive* matching for a pattern (Foundation 2022). For instance, the pattern *r*"\b(the)\b" would match "*The*", "*ThE*", etc, when the flag is used in any function.

Now we require functions which can determine if a match (article) exists in a string and all the matches for each of the patterns.

Functions used:

In order to find the first location where a pattern produces a match, the regular function **re.search(*pattern, string, flags=re.I*)** was used. As the name suggests, this function finds a match to the pattern in the string, regardless of capitalisation (as defined by the flag) (Foundation 2022). If any article pattern exists, it can then be censored. However, this requires determining their location within the string - specifically, the start and end. Another function **re.finditer(*pattern, string, flags=re.I*)** was used to determine this information.

It scans the string from left-to-right and returns an iterator that yields *match* objects which match the regex pattern (Foundation 2022).

I.e, it returns every match to the pattern as a regex *match* object. These objects have 2 attributes useful in this problem; the **match.start()** and **match.end()** which return the starting and ending index of a match respectively (Foundation 2022). Given the start and end indices, the string can be censored with a pound # between these indices - thus censoring the article. Applying this for each match (i.e. article), the whole string can be censored.

2 Linear algebra

Before implementing the problem in Python, it needs to be defined using mathematical terminology.

From the premise, 2 simultaneous equations that solve for nitrogen **n**, and phosphate **p** can be created.

$$n = 0.3A + 0.1B$$

$$p = 0.2A + 0.1B$$

Where A and B represent 1 kilogram of fertiliser A and B respectively.

If we consider that the ratios of nitrogen and phosphate in the two fertilisers are unknown, where

- a_n and b_n are the amounts of nitrogen in 1kg of fertiliser A and B respectively
- a_p and b_p are the amounts of phosphate in 1kg of fertiliser A and B respectively

Then the equations can be re-written as

$$n = a_nA + b_nB$$

$$p = a_pA + b_pB$$

In matrix form:

$$\begin{bmatrix} n \\ p \end{bmatrix} = \begin{bmatrix} a_n & b_n \\ a_p & b_p \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$$

Since the amount of fertiliser A and B is unknown, we can rearrange the equation as such

$$\begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} a_n & b_n \\ a_p & b_p \end{bmatrix}^{-1} \begin{bmatrix} n \\ p \end{bmatrix} \tag{1}$$

The inverse of $\left[\begin{smallmatrix} a_n & b_n \\ a_p & b_p \end{smallmatrix}\right]$ can be written using the rule for inverses of 2-by-2 matrices:

$$\begin{bmatrix} a_n & b_n \\ a_p & b_p \end{bmatrix}^{-1} = det(C) \cdot \begin{bmatrix} b_p & -b_n \\ -a_p & a_n \end{bmatrix}$$

$$Where\ det(C) = \frac{1}{a_nb_p - b_na_p}$$

From the above formulae, it can be observed that the inverse will not exist if the determinant; $det(C)$ is 0. If $det(C) = 0$, then referring to equation (1), vector $\left[\begin{smallmatrix} A \\ B \end{smallmatrix}\right]$ would not have a unique solution.

This problem was implemented in python using the function fertiliser(an, ap, bn, bp, n, p), which returns (a,b), where a and b represent the amount of fertiliser A and B respectively. Using the Python package **numpy**, the function returns None when the determinant of matrix C is 0. Additionally, since the amount of fertiliser cannot be negative, it returns None if a or b are negative.

References

Foundation, P. S. (2022). Re — regular expression operations — python 3.10.5 documentation. *docs.python.org*. Retrieved June 16, 2022, from https://docs.python.org/3/library/re.html

teaching team, C. (2022). Cab203 problem solving. Retrieved June 16, 2022, from https://blackboard.qut.edu.au/bbcswebdav/pid-9577305-dt-content-rid-51686171_1/courses/CAB203_22se1/problem_solving_example.pdf