12/22/2019

Arapaut Sivaprasad, Ph.D.

WebGenie Software Pty Ltd.

[avs2904@webgenie.com](mailto:avs2904@webgenie.com); <https://www.linkedin.com/in/dr-siva/>

m: 0411 132 646

**STLP Project**

**- Process Flow in nlp\_parallel.py** **-**

# INTRODUCTION

Commercial in Confidence

# DISCLAIMER

This document is a high-level design of the project and is based on the current understanding of the requirements and methodology. It primarily deals with analysing the content of web pages for similarity, which is applicable for locally stored text files as well. It may not be useful in classifying documents of dissimilar contents into the same category. Changes in the algorithm may become necessary as we progress.

# INTRODUCTION

This is a description of the process flow in the Python program, nlp\_parallel.py, to make it clear how the comparisons are done. Such info can be used in modifying the code to deal with other types of data input (e.g. the NLP output from Gavin’s documents).

## Fig.1 – High level process flow

8 Workers

ReadUrls()

GetDocSegments()

CalculateSimilarity()

C\_1D array

main()

par\_compare()

2D matrix to file

par\_ReadUrls()

Compare Ref Matrix

Result as %age

# DETAILS

## Overview

To run faster the program is parallelised for the comparison steps, using as many workers as there are logical processors (typically 8 on a PC) on the machine. Since each worker can do the comparison independently of others, this is an “embarrassingly parallel” algorithm.

Each worker is assigned a set of pairs to compare. These are arrived at by taking non-repetitive pairs between all, much like in a cricket match where every team plays once against all opponents. The total number of pairs will grow exponentially with the number of files/URLs in the set. The total number is divided by the number of workers and each given an array of the pairs.

The pairwise comparisons produce a “Cosine Similarity Score (CS)” which is recorded in a one-dimensional array (“C1\_D”) that is accessible to all workers. This is a special array via Python’s multiprocessing module, as a normal array is only accessible to the same process.

The main function, ‘par\_compare’, starts the workers and waits for all to finish. When all have finished, the C1\_D array is read and added to a 2-dimensional matrix. This matrix is compared against a reference matrix where a 1 represents pair that should have a score above a threshold (empirically determined) and a 0 represents a pair not expected to score above threshold. This matrix is manually generated.

The comparison with the reference matrix will give a score of matches versus non-matches. The score expressed as a percentage of total is the success rate. For example, for 30 files there will be 435 pair‑wise comparisons. Of these, if 75 are marked as 1 in the reference table, then if all scores in these cells are above threshold, and every cell with 0 has a lower than threshold score, the success rate is 100%. If some of these cells score below threshold, they are false negatives. Similarly, if some of the other cells score above threshold, they are false positives. The sum of false positives and false positives expressed as a percentage of total cells in the matrix gives the “failure rate” which, when subtracted from 100 gives the “success rate”.

Obviously, the preparation of the reference matrix is critical and must not wrongly mark a cell. This could be a challenge in the case of thousands of files in a matrix.

## Summary of Functions

These are the functions in the program and a short description of what they do. Details of code logic will follow.

* def par\_compare():
  + Sets up multi-processing.
  + Read CS scores in C\_1D and add to a 2-dimensional matrix
  + Compare the matrix against reference matrix.
  + Calculate and display the success rate.
* def par\_ReadUrls(m,n,pairs,w,C\_1D):
  + ReadUrls(i, j):
    - Read the files.
      * If first time, read the URL and write the content to file.
    - Takes these components:
      * text: visible text in body
      * title: Page title in the <head>
      * tw: Keywords in the <head>
      * desc: Page description in the <head>
    - Split text into words and Lemmatise them (t1[]).
    - Do. with title.
  + GetDocSegments():
    - Split the list of lemmatised words into chunks of 100
  + CalculateSimilarity(l, m, i, j, k):
    - Calculate the cosine similarity (CS) between pairs of segments in the docs.
    - Return the average of the CS scores.
  + Add the CS score into C\_1D
  + Return

## Code details

### Modules

**import** spacy  
**import** urllib.request  
**from** bs4 **import** BeautifulSoup  
**from** bs4.element **import** Comment  
**import** urllib.request  
**import** re  
**import** time  
**from** nltk.corpus **import** stopwords  
**import** multiprocessing **as** mp

### **def par\_compare():**

This function sets up multi-processing.  
 1. Create an array of the pairs of comparisons. This is in the form of a list as below, where the numbers are the indices of the URLs...  
 ['0,1', '0,2', '0,3', '0,4', '1,2', '1,3', '1,4', '2,3', '2,4', '3,4']

pairs = []  
**for** i **in** range(nurls):  
 **for** j **in** range(i + 1, nurls):  
 k = str(i) + **','** + str(j)  
 pairs.append(k)

This code creates an array of pairs that excludes self‑comparisons. Thus, i-i are avoided and only one of i-j and j-i is included. The total number of items in the array will be N = ((nurls\*nurls+1)/2) – nurls. Thus, for 30 URLs it will be ((30\*31)/2)-30 = **435**

2. Create a flat array that works across multiple processes. It will hold the results from every sub-process and finally compiles them into one list.

C\_1D = mp.RawArray(**'d'**, len(pairs))

# This RawArray is an 1D array, or a chunk of memory that will be used to hold the data matrix. Each worker is given a specific chunk of it as ‘b’ to ‘e’, where b is the starting index and e is the ending index number. When all workers have re-joined the parent function the C\_1D will hold the data in a linear array. It will then be split into a 2D array.

3. Find the number of CPUs on the system. These many workers will be launched.

num\_workers = mp.cpu\_count()

4. Determine the chunk\_size based on the length of pairs and num\_workers.

Each worker will be given the chunk\_size pairs to process. For example, if the number of pairs is 64 and there are 8 workers, then each worker will get 8 pairs to process.

chunk\_size = len(pairs) // num\_workers + 1  
n\_chunks = len(pairs) // chunk\_size

In this code, the ‘chunk\_size’ is the next integer from total number of pairs divided by the num\_workers. These many pairs will be assigned to the first ‘num\_workers-1’ workers. The remainder will be assigned to the last worker. e.g. 435 // 55 = 7, which means 7 workers will get 55 pairs each and the remainder, 50, will go to worker 8.

e = 0

**for** w **in** range(n\_chunks):  
 b = w \* chunk\_size  
 e = b + chunk\_size  
workers.append(mp.Process(target=par\_ReadUrls, args=(b, e, pairs, w, C\_1D)))

The first 7 workers will get 55 pairs as range ‘b to e’.

**try**:  
 r = 0  
 **if** e:  
 r = len(pairs) - e  
 **if** r > 0:  
 w += 1  
workers.append(mp.Process(target=par\_ReadUrls, args=(e, len(pairs), pairs, w, C\_1D)))

The remainder, 50, will go to worker 8 as range ‘e to len(pairs)’. So, theoretically, worker 8 should finish first in most cases.

**for** w **in** workers:  
 w.start()

As many processes are started as there are workers.

num\_workers,workers: 8 [<Process(Process-1, initial)>, <Process(Process-2, initial)>,...]

*# Wait for all processes to finish***for** w **in** workers:  
 w.join()

*Initialise a 2-dimensionsl array of nurls\*nurls items initialised with 0s to hold the cosine similarity scores.*matrix = [[0] \* nurls **for** i **in** range(nn)]  
  
*Print the top row values as we populate the matrix. It is for visual display alone.*print(**"#,"**, end=**''**)  
**for** i **in** range(nn):

Print the first row showing the indices of URLs as e.g.,

#,0,1,2,3,4,5,…,27,28,29,  
 print(**"{},"**.format(i),end=**''**)  
  
*C\_1D is a 1-dimensional array that holds the values. It gets the values from all workers together. Split it into rows and columns in matrix.* **for** j **in** range(i+1,nn):  
 matrix[i][j] = C\_1D[k]  
 k += 1  
 print(**''**)

In the above code the top row of the display is created and the 2D matrix is populated with data from C\_1D array.

Now, display the matrix. The leftmost column is the index of URLs and a row is the matrix scores as e.g.

#,0,1,2,3,4,

0,0,0.97,0.96,0.95,0.94,

**for** i **in** range(nn):  
 print(**"{},"**.format(i),end=**''**)  
 **for** j **in** range(nn):  
 print(**"{},"**.format(matrix[i][j]), end=**''**)  
 print(**''**)

This code is to display the matrix for visual examination by copying and pasting into an Excel sheet. It is not required for the program to calculate the success rate.

*# Write the results into a file. This will be used to compare against a reference matrix for expected matches*

**with** open(**'Files/matrix.txt'**, **'w'**) **as** f:  
 **for** i **in** range(nurls):  
 **for** j **in** range(nurls):  
 f.write(**"{},"**.format(str(matrix[i][j])))  
 f.write(**"\n"**)

This code stores the matrix in a file. It is not required for the program to calculate the success rate.

*# Read in the ref\_table for expected matches*ref\_table = [[0] \* 30 **for** i **in** range(30)]  
j = 0  
**with** open(**'ref\_table.txt'**,**'r'**) **as** f:  
 lines = f.readlines()  
 **for** i **in** range(len(lines)):  
 lines[i].rstrip(**'\n\r'**)  
 la = lines[i].split(**'\t'**)  
ref\_table[j] = la  
 j += 1

This code reads in the reference matrix to check against the results matrix.

**STILL TO DO:** Compare the reference matrix with results matrix and calculate the success rate.

### def par\_ReadUrls (m, n, pairs, w, C\_1D):

**for** k **in** range(m, n):  
 pair = pairs[k]  
 i\_j = pair.split(**','**)  
 i = int(i\_j[0])  
 j = int(i\_j[1])  
 ReadUrls(i, j)  
 ld1, ld2 = GetDocSegments()  
 sc = CalculateSimilarity(ld1, ld2, nurls)  
 s = round(sc, 2)  
 C\_1D[k] = s

1. Each worker gets a range of pairs to process. These are done sequentially

2. Split the pairs[k] into two indices. e.g. '0,1' to 0 and 1. These are the two URLs to be compared

3. Send these to 'ReadUrls' to get the file contents.

4. Call 'GetDocSegments' to split the content into smaller chunks.

5. Call 'CalculateSimilarity' to calculate the cosine similarity.

6. Record the value in C\_1D array.

### def ReadUrls(i, j):

This function reads the URLs to get the body text (“text”), title, keywords and description. Only the text is currently used in the comparison, but others may be used in future.

Text is split into tokens, lemmatised and the common words removed. Also removed are all words of 3 chars or less. Repeated words are NOT squeezed. The list (‘t1’) is global and is used for cosine similarity score.

**try**:  
 **with** open(file1, **"rb"**) **as** f:  
 filecontent = bytes(f.read())  
 text1, title1, kw1, desc1 = text\_from\_html(filecontent,file1)  
 *# Take the lemmas of words within the text in pages. The 'text1' comes from the <body> and omits title* t1 = Lemmatise(text1)  
t1t = Lemmatise(str(title1))  
**if** err:  
 *# If stored file does not exist, then get the content from the URL and write it out in stored file. Since the same URL is used in many comparisons, faster to store it and read from it for subsequent calls.* response = urllib.request.urlopen(url1)  
 html1 = response.read()  
 **with** open(file1, **'wb'**) **as** f:  
 f.write(html1)  
 text1, title1, kw1, desc1 = text\_from\_html(html1,url1)  
 t1 = Lemmatise(text1)  
 t1t = Lemmatise(str(title1))

Firstly, check and read the page contents of the URLs previously retrieved and written out. If not present, get the contents from the URLs and analyse as well as write out. Since the same URL will be compared multiple times with others, it is necessary to write out the content for speed.

text\_from\_html:Reads the filecontent and parse it as HTML. Omit the text within several HTML tags and take only the visible text in the body.

Lemmatise: The text returned by the above function is lemmatised and removed of common words and those with 3 or less chars.

*# Make a list of words and their counts. We will take the top 10 for further analysis. The word counts are made from the lemmas of 'text1'. The most popular words are compared between the URLs*CountWords(t1)  
t1w = []  
**for** key, val **in** counts.items():  
 item = str(val) + **':'** + str(key.lower())  
 t1w.append(item)  
t1w.sort(key=**lambda** fname: int(fname.split(**':'**)[0]), reverse=**True**)

The items in the t1 and t2 arrays are counted for word frequencies. This is not used at present, but may be used in future.

### def GetDocSegments():

This function splits the ‘t1’ and ‘t2’ arrays into smaller chunks of 100 words (arbitrary number) and keep in two arrays (‘doc1\_segs’ and ‘doc2\_segs’). These segments will be compared in all possible combinations.

c\_sz = 100 # Arbitrary number of words  
lt1 = len(t1) # Length of the array of lemmatised words.  
n = lt1 // c\_sz # Number of full segments. e.g. n == 4 if 459 words  
r = lt1 % c\_sz # Remainder of the arrays. e.g. r == 59

**for** i **in** range(n):  
 b = i \* c\_sz  
 e = b + c\_sz  
 **for** j **in** range(b, e):  
 text += t1[j] + **' '** doc1\_segs.append(text)  
 text = **''**

From ‘t1’ array 100 words each are cocatenated into ‘n’ items in ‘doc1\_segs’. The same is done with ‘t2’ array.

**if** r > 0:  
 b = e  
 e = lt1  
 **for** j **in** range(b, e):  
 text += t1[j] + **' '** doc1\_segs.append(text)

After the full segments are appended to ‘doc1\_segs’, the remainder is appended as the last item.

**return** len(doc1\_segs), len(doc2\_segs)

### def CalculateSimilarity(l, m, k):

\_\_\_\_\_\_\_\_\_\_\_END OF DOCUMENT\_\_\_\_\_\_\_\_\_\_\_