Store Pre-processing Text Result Methods

Text Pre-processing has become a very important task in modern Computer Science fields such as Artificial Intelligent and Machine Learning. For example, in 2013, E. Haddi et. al have used multiples Text Pre-processing methods to analyze the sentiment of online movie reviews [1]. However, it is a very expensive process, and it becomes extremely expensive when the dataset becomes bigger. For example, In one of the research on extracting Named Entities from text, the data scientists have to process millions of online news articles from multiple online publishers including the Los Angeles Times, Routers, and the New York Times [2].

There are three main components of Pre-processing including *Tokenization*, *Normalization*, and *Substitution* [3] and their output results can be reused many times. However, those tasks require very high computational resources. In order to minimize the computing cost, we can save the output results to file and then we can reload them while needed. So that saving results to a file becomes a mandatory feature. There are some methods to address this problem and the methods' technique can be variable which depends on the data structure of the results. The methods can be put into two main categories which are *Human-readable* (text-based) and *Non-human-readable* (binary).

Since all the results of Text Pre-processing in this assignment is stored in Objects so the problem is narrowed down to "How to store an Java Object to file" and it can be solved using *Object Persistence Methods* [4], they are basically a process that transforms an Object to a format that can be stored on disk or database, however, we are only interested in storing in file due to the scope of this assignment.

- The first approach is to use the Build-in Java Serialization which makes use of object stream classes including *ObjectInputStream* and *ObjectOutputStream* [5] to convert and write an object to a binary file.
- The second approach is to transform the Objects to a human-readable string using the *Data Markup Language* such as XML or *Data Serialization Language* such as JSON and YAML and then those strings to a text file.
- The third approach is to convert the preprocessed data into *Comma-Separated Value* (CSV) format and then write to a text file.

I will use the third approach for my implementation, although it may not be as efficient as using Java Serialization in term of run time and space, however, it is a multi-platforms method that can be used with multiple programming languages and it is also human-readable that is easier for error checking. For each section, I will convert the Map of sanitized lines into a CSV file named *<section>.data.csv*, another index CSV named *<section>.index.csv* will store the Trie of words as well as the index tables of occurrences.

By loading data directly from the document, the *loadDocument()* method runs with the time complexity of **O(c)** with **c** is the number of characters in the documents. However, by loading the data from the stored CSV files, it only runs with the time-complexity of **O(w)** with **w** is the number of words in the document. That is a significant improvement because **c** >> **w**. The detailed time complexity analysis is described in the following pseudo-code fragments.

Time Complexity Analysis

Load directly from Document Pseudo Code	#Operations
Indexes <- GetIndexes()	#sections (s)
SectionMap <- Empty Map of Sections	1
LineNo <- 0	1
For each **Entry** in **Indexes**:	#sections (s)
Section <- New Section	1
Trie <- New Trie	1
Lines <- Empty Map of Lines	1
Files <- New File("document.txt")	1
While **LineNumber** < **Entry**.LastLine():	#lines in sect. (n)
Line <- **Files**.ReadLine()	1
Text <- Sanitize(**Line**)	#chars in line (c)
Lines.Add(**LineNo, Text**)	1 (expected)
Words <- Tokenize(**Text**)	1 (expected)
For each **Word, Pos** in **Words**:	#chars in line (m)
Trie.Insert(**Word**)	#chars in word (c)
Table <- **Trie**.Get(**Word**)	#chars in word (c)
If **Table** == Null:	1
Table <- New IndexTable	1
Trie.Set(**Word, Table**)	#chars in word (c)
Table.Add(**Pos**)	1 (expected)
Section.Trie <- **Trie**	1
Section.Lines <- **Lines**	1
SectionMap.Add(**Entry**.Title(), **Section**)	1 (expected)
Accumulated: **s + s(n(m + m(3c))) ≈ snmc.** Since **c** can be considered as constant, then the time complexity can be considered as **O(snm)** which is actually the number of character in the document	

Load from Stored Result Pseudo Code	#Operations
Indexes <- GetIndexes()	#sections (s)
Sections <- Empty Map of Sections	1
For each **Index** in **Indexes**:	#sections (s)
Section <- New Section	1
Trie <- New Trie	1
Lines <- Empty Map of Lines	1
DFile <- New File(**Index**.Title()+".data.csv")	1
While !**DFile**.EOF:	#lines in sect. (n)
LineNo, Text <- **DFile**.ReadLine()	2
Lines.Add(**LineNo, Text**)	1 (expected)
IFile <- New File(**Index**.Title()+".index.csv")	1
Word <- Null;	1
Table <- Null;	1
While !**IFile**.EOF:	#words in sect. (w)
NextWord, Pos <- **IFile**.ReadLine()	2
If **NextWord** != **Word**:	1
Trie.Insert(**Word**)	#chars in word (c)
Table <- New IndexTable	1
Trie.Set(**Word, Table**)	#chars in word (c)
If **Table** != Null:	1
Table.Add(**Pos**)	1 (expected)
Section.Trie <- **Trie**	1
Section.Lines <- **Lines**	1
SectionMap.Add(**Entry**.Title(), **Section**)	1
Accumulated: **s + s(n + w(2c)) ≈ swc.** Since **c** can be considered as constant, then the time complexity can be considered as **O(sw)** which is actually the number of words in the document	

References

[1] E. Haddi, X. Liu and Y. Shi, "The Role of Text Pre-processing in Sentiment Analysis", *Procedia Computer Science*, vol. 17, pp. 26-32, 2013.

[2] Y. Shinyama and S. Sekine, "Named entity discovery using comparable news articles," in *Proceedings of the 20th international conference on Computational Linguistics*, 2004, p. 848: Association for Computational Linguistics.

[3] M. Mayo, "A General Approach to Preprocessing Text Data", *Kdnuggets.com*, 2018. [Online]. Available: https://www.kdnuggets.com/2017/12/general-approach-preprocessing-text-data.html. [Accessed: 12- Oct- 2018].

[4] C. JMTauro, R. Kumar Sahai and S. Rani A., "Object Persistence Techniques - A Study of Approaches, Benefits, Limits and Challenges", *International Journal of Computer Applications*, vol. 85, no. 5, pp. 19-27, 2014.

[5] "The Java™ Tutorials - Object Streams", *docs.oracle.com*, 2018. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/io/objectstreams.html. [Accessed: 12- Oct- 2018].

[6] S. man and A. Utama Siahaan, "Huffman Text Compression Technique", *International Journal of Computer Science and Engineering*, vol. 3, no. 8, pp. 103-108, 2016.