Methods of Minimize Pre-processing Cost

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. So that, 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 is a popular problem and is becoming a necessary task.

The general definition of this problem is called *Serialization* which basically a process that transforms *(Seriallize)* an Object to a format that can be stored on disk or database [7], and then it can be transform back *(Deserialize)* to the original Object using *Deserialization* methods. The core mechanism of these methods is to flatten the multi-dimensional Objects into a one- dimensional stream of characters or bits, so then they can decode these data stream in order to reconstruct the original Object.

There are many method described in *Object Persistence Techniques* [4] is used 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 Java Data Structure so the problem is narrowed down to "How to store a Java Object to a file" and it can be solved using these feasible methods:

1. 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.
2. 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.
3. 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 *<startLine>.data.csv*, another index CSV named *<startLine>.index.csv* will store the Trie of words as well as the index tables of occurrences.

One other reason that makes me prefer this method is that I can implements the *serialization* and *deserialization* methods by myself so that I can fully analyze the time complexity of my algorithms. It helps me to understand more about Object Persistent process.

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 of **c** >> **w**. The detailed time complexity analysis is described in the following pseudo-code fragments.

To verify the efficiency of implementing load and store Text Pre-processing Result in to CSV files, I conducted some experiments which are shown in the *Experiment Results Section* to derived the actual running time of each try including i) read content directly from document, ii) road content and then store the result to files and iii) load the result from files. The running time of using stored pre-processing results can be reduced by around 35% (from 5.56s to 3.59s), however, the time for reading and loading at the first time is increased by nearly 50% (from 5.56s to 8.22s).

In conclusion, storing the Text Pre-processing results for reusing can significantly reduce the cost of Pre-processing tasks. However, we have to trade off some other resources that are the increase of the disk space needed to store the files and the increase in running time of the first run.

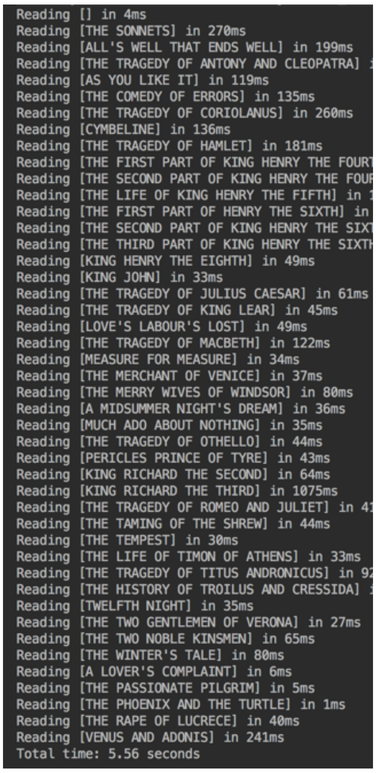
Time Complexity Analysis

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.startLine()+".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.startLine()+".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

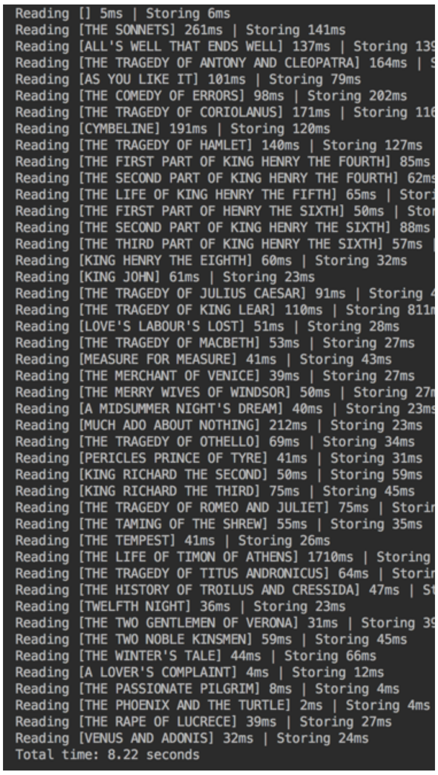
Accumulated: **s + s(n + w(2c)) ≈ swc.** Since **c** can be considered as constant, then the time complexity can be considered as **O(sw)** which is actually the number of words in the document.

Experiment Results



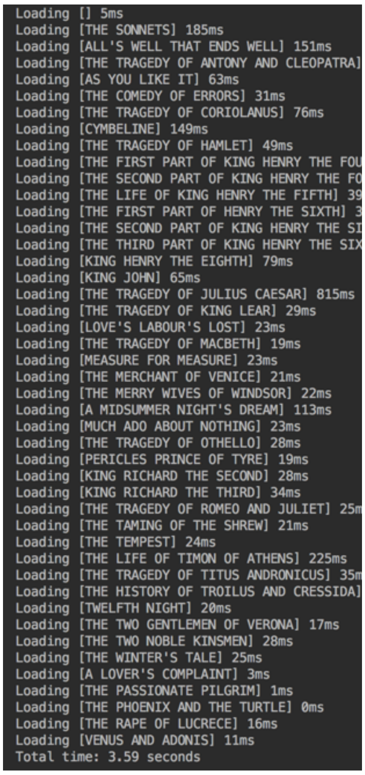
```
Reading [] in 4ms
Reading [THE SONNETS] in 270ms
Reading [ALL'S WELL THAT ENDS WELL] in 199ms
Reading [THE TRAGEDY OF ANTONY AND CLEOPATRA]
Reading [AS YOU LIKE IT] in 119ms
Reading [THE COMEDY OF ERRORS] in 135ms
Reading [THE TRAGEDY OF CORIOLANUS] in 260ms
Reading [CYMBELINE] in 136ms
Reading [THE TRAGEDY OF HAMLET] in 181ms
Reading [THE FIRST PART OF KING HENRY THE FOUR
Reading [THE SECOND PART OF KING HENRY THE FOU
Reading [THE LIFE OF KING HENRY THE FIFTH] in
Reading [THE FIRST PART OF HENRY THE SIXTH] in
Reading [THE SECOND PART OF KING HENRY THE SIX
Reading [THE THIRD PART OF KING HENRY THE SIXT
Reading [KING HENRY THE EIGHTH] in 49ms
Reading [KING JOHN] in 33ms
Reading [THE TRAGEDY OF JULIUS CAESAR] in 61ms
Reading [THE TRAGEDY OF KING LEAR] in 45ms
Reading [LOVE'S LABOUR'S LOST] in 49ms
Reading [THE TRAGEDY OF MACBETH] in 122ms
Reading [MEASURE FOR MEASURE] in 34ms
Reading [THE MERCHANT OF VENICE] in 37ms
Reading [THE MERRY WIVES OF WINDSOR] in 80ms
Reading [A MIDSUMMER NIGHT'S DREAM] in 36ms
Reading [MUCH ADO ABOUT NOTHING] in 35ms
Reading [THE TRAGEDY OF OTHELLO] in 44ms
Reading [PERICLES PRINCE OF TYRE] in 43ms
Reading [KING RICHARD THE SECOND] in 64ms
Reading [KING RICHARD THE THIRD] in 1075ms
Reading [THE TRAGEDY OF ROMEO AND JULIET] in 4
Reading [THE TAMING OF THE SHREW] in 44ms
Reading [THE TEMPEST] in 30ms
Reading [THE LIFE OF TIMON OF ATHENS] in 33ms
Reading [THE TRAGEDY OF TITUS ANDRONICUS] in 9
Reading [THE HISTORY OF TROILUS AND CRESSIDA]
Reading [TWELFTH NIGHT] in 35ms
Reading [THE TWO GENTLEMEN OF VERONA] in 27ms
Reading [THE TWO NOBLE KINSMEN] in 65ms
Reading [THE WINTER'S TALE] in 80ms
Reading [A LOVER'S COMPLAINT] in 6ms
Reading [THE PASSIONATE PILGRIM] in 5ms
Reading [THE PHOENIX AND THE TURTLE] in 1ms
Reading [THE RAPE OF LUCRECE] in 40ms
Reading [VENUS AND ADONIS] in 241ms
Total time: 5.56 seconds
```

Fig. 1: Read from Document



```
Reading [] 5ms | Storing 6ms
Reading [THE SONNETS] 261ms | Storing 141ms
Reading [ALL'S WELL THAT ENDS WELL] 137ms | Storing 13
Reading [THE TRAGEDY OF ANTONY AND CLEOPATRA] 164ms |
Reading [AS YOU LIKE IT] 101ms | Storing 79ms
Reading [THE COMEDY OF ERRORS] 98ms | Storing 202ms
Reading [THE TRAGEDY OF CORIOLANUS] 171ms | Storing 11
Reading [CYMBELINE] 191ms | Storing 120ms
Reading [THE TRAGEDY OF HAMLET] 140ms | Storing 127ms
Reading [THE FIRST PART OF KING HENRY THE FOURTH] 85ms
Reading [THE SECOND PART OF KING HENRY THE FOURTH] 62m
Reading [THE LIFE OF KING HENRY THE FIFTH] 65ms | Stor
Reading [THE FIRST PART OF HENRY THE SIXTH] 50ms | Sto
Reading [THE SECOND PART OF KING HENRY THE SIXTH] 88ms
Reading [THE THIRD PART OF KING HENRY THE SIXTH] 57ms
Reading [KING HENRY THE EIGHTH] 60ms | Storing 32ms
Reading [KING JOHN] 61ms | Storing 23ms
Reading [THE TRAGEDY OF JULIUS CAESAR] 91ms | Storing
Reading [THE TRAGEDY OF KING LEAR] 110ms | Storing 811
Reading [LOVE'S LABOUR'S LOST] 51ms | Storing 28ms
Reading [THE TRAGEDY OF MACBETH] 53ms | Storing 27ms
Reading [MEASURE FOR MEASURE] 41ms | Storing 43ms
Reading [THE MERCHANT OF VENICE] 39ms | Storing 27ms
Reading [THE MERRY WIVES OF WINDSOR] 50ms | Storing 27
Reading [A MIDSUMMER NIGHT'S DREAM] 40ms | Storing 23m
Reading [MUCH ADO ABOUT NOTHING] 212ms | Storing 23ms
Reading [THE TRAGEDY OF OTHELLO] 69ms | Storing 34ms
Reading [PERICLES PRINCE OF TYRE] 41ms | Storing 31ms
Reading [KING RICHARD THE SECOND] 50ms | Storing 59ms
Reading [KING RICHARD THE THIRD] 75ms | Storing 45ms
Reading [THE TRAGEDY OF ROMEO AND JULIET] 75ms | Stori
Reading [THE TAMING OF THE SHREW] 55ms | Storing 35ms
Reading [THE TEMPEST] 41ms | Storing 26ms
Reading [THE LIFE OF TIMON OF ATHENS] 1710ms | Storing
Reading [THE TRAGEDY OF TITUS ANDRONICUS] 64ms | Stori
Reading [THE HISTORY OF TROILUS AND CRESSIDA] 47ms | S
Reading [TWELFTH NIGHT] 36ms | Storing 23ms
Reading [THE TWO GENTLEMEN OF VERONA] 31ms | Storing 3
Reading [THE TWO NOBLE KINSMEN] 59ms | Storing 45ms
Reading [THE WINTER'S TALE] 44ms | Storing 66ms
Reading [A LOVER'S COMPLAINT] 4ms | Storing 12ms
Reading [THE PASSIONATE PILGRIM] 8ms | Storing 4ms
Reading [THE PHOENIX AND THE TURTLE] 2ms | Storing 4ms
Reading [THE RAPE OF LUCRECE] 39ms | Storing 27ms
Reading [VENUS AND ADONIS] 32ms | Storing 24ms
Total time: 8.22 seconds
```

Fig. 2 Read and Store the Results



```
Loading [] 5ms
Loading [THE SONNETS] 185ms
Loading [ALL'S WELL THAT ENDS WELL] 151ms
Loading [THE TRAGEDY OF ANTONY AND CLEOPATRA]
Loading [AS YOU LIKE IT] 63ms
Loading [THE COMEDY OF ERRORS] 31ms
Loading [THE TRAGEDY OF CORIOLANUS] 76ms
Loading [CYMBELINE] 149ms
Loading [THE TRAGEDY OF HAMLET] 49ms
Loading [THE FIRST PART OF KING HENRY THE FOU
Loading [THE SECOND PART OF KING HENRY THE FO
Loading [THE LIFE OF KING HENRY THE FIFTH] 39
Loading [THE FIRST PART OF HENRY THE SIXTH]
Loading [THE SECOND PART OF KING HENRY THE SI
Loading [THE THIRD PART OF KING HENRY THE SIX
Loading [KING HENRY THE EIGHTH] 79ms
Loading [KING JOHN] 65ms
Loading [THE TRAGEDY OF JULIUS CAESAR] 815ms
Loading [THE TRAGEDY OF KING LEAR] 29ms
Loading [LOVE'S LABOUR'S LOST] 23ms
Loading [THE TRAGEDY OF MACBETH] 19ms
Loading [MEASURE FOR MEASURE] 23ms
Loading [THE MERCHANT OF VENICE] 21ms
Loading [THE MERRY WIVES OF WINDSOR] 22ms
Loading [A MIDSUMMER NIGHT'S DREAM] 113ms
Loading [MUCH ADO ABOUT NOTHING] 23ms
Loading [THE TRAGEDY OF OTHELLO] 28ms
Loading [PERICLES PRINCE OF TYRE] 19ms
Loading [KING RICHARD THE SECOND] 28ms
Loading [KING RICHARD THE THIRD] 34ms
Loading [THE TRAGEDY OF ROMEO AND JULIET] 25
Loading [THE TAMING OF THE SHREW] 21ms
Loading [THE TEMPEST] 24ms
Loading [THE LIFE OF TIMON OF ATHENS] 225ms
Loading [THE TRAGEDY OF TITUS ANDRONICUS] 35
Loading [THE HISTORY OF TROILUS AND CRESSIDA]
Loading [TWELFTH NIGHT] 20ms
Loading [THE TWO GENTLEMEN OF VERONA] 17ms
Loading [THE TWO NOBLE KINSMEN] 28ms
Loading [THE WINTER'S TALE] 25ms
Loading [A LOVER'S COMPLAINT] 3ms
Loading [THE PASSIONATE PILGRIM] 1ms
Loading [THE PHOENIX AND THE TURTLE] 0ms
Loading [THE RAPE OF LUCRECE] 16ms
Loading [VENUS AND ADONIS] 11ms
Total time: 3.59 seconds
```

Fig. 3 Load Results from File

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.