REPORT ON TEXT COMPRESSION

- **UID: 180128022**

We have implemented a Huffman Coding text compression algorithm in Python to investigate the pros and cons of character-based and word-based algorithms.

Code Implementation:

huff-compress.py – In this file, there are three classes, **cmd**, **nodeHuffman** and **codeHuffman**. The **cmd** class is used to enable character-based and word-based commandline operations on the file. The class **nodeHuffman** has the constructor method that includes probability of occurrence of words/characters and the final value of the string as arguments. The class **codeHuffman** has the following methods:

- constructor method for opening and reading a file (In our case, *'mobydick.txt')*
- **letter_count** for finding out individual characters through Regular Expressions and doing the subsequent operations on them to update frequencies, along with adding the pseudo-EOF marker
- **word_count** for finding out words through Regular Expressions and doing the same as above
- **tree_Huffman** for constructing the Huffman tree, mapping the parent-child relationship, assigning '1's and '0's depending on the left-child and right-child, traversing the tree in reverse and finally, writing the output to a *.pkl* file
- **encoder** for encoding the string with EOF
- **bit_func** for operations to represent the binary file properly and writing the text of compressed symbol model to a *.bin* file called

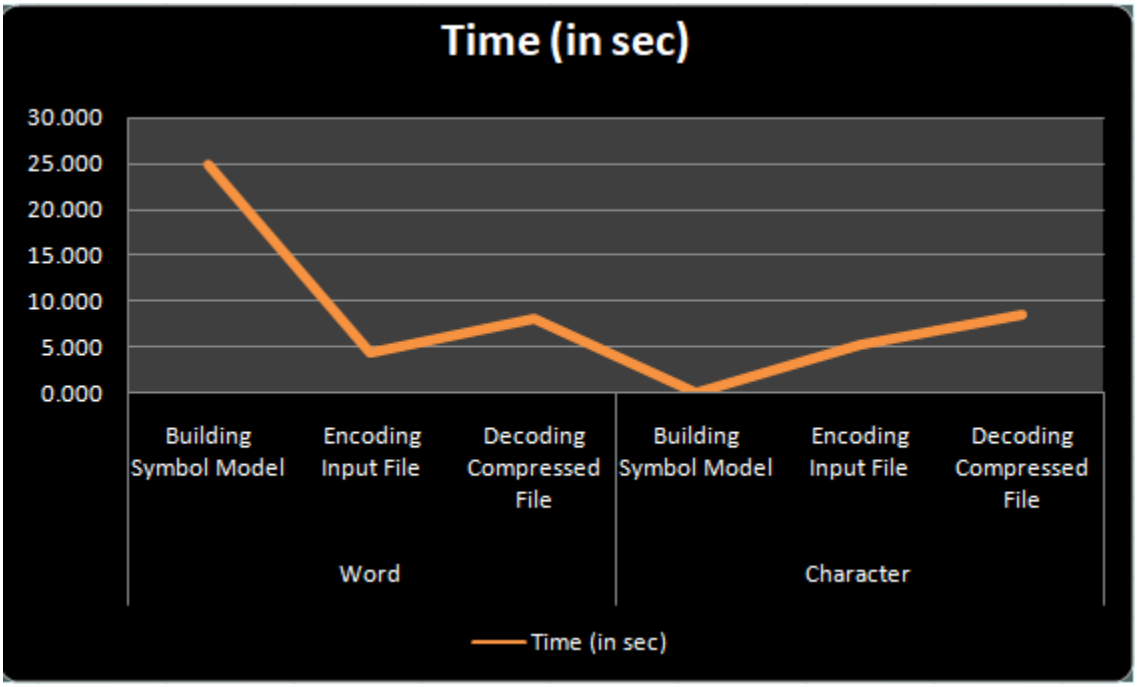
Apart from the above classes, the code block for calling class instances, operations for feeding any text file from the command line and timing our encoding operations, is also present.

huff-decompress.py – In this file, we have a class called **decompressorHuffman** with the following methods:

- constructor method with arguments to feed in the serialised version of the symbol model and its corresponding text
- **dictionary_reader** for loading the data from the compressed dictionary
- **array_reader** for reading the compressed code
- **binary_converter** for implementing the binary operations by iterating through our compressed code and returning its binary string equivalent, trimming '0b' from the string and returning the string after zero-padding
- **decoder** for getting back the original string and writing the decompressed original string in to a *.txt* file after removing 'EOF' from the text.

Apart from the above class, the code block for calling class instances, operations for feeding any text file from the command line and timing our decoding operations are also present.

Observation:



Quiz:

1. How does the compression performance of character-based versus word-based Huffman compare? Specifically:

(a)How big (in bytes) are the compressed text files produced in each case?
Ans. For **Word** Based – **833 KB**, For **Character** Based –**895 KB**
(b) How big (in bytes) are the symbol models produced in each case?
Ans. For **Word** Based – **773 KB**, For **Character** Based –**2 KB**
(a) How long does it take to
i. build the symbol model?
Ans. For **Word** Based - **24.953** sec, For **Character** Based – **0.002** sec
ii. encode the input file given the symbol model?
Ans. For **Word** Based - **4.475** sec, For **Character** Based – **5.343** sec
iii. decode the compressed file? (including unpickling the symbol model)
Ans. For **Word** Based – **8.055** sec, For **Character** Based – **8.489** sec

2.How could various aspects of the performance, as identified in response to the previous question, be improved?

Ans. First, the performance can be radically improved if tested on systems with better configuration, for instance, better processor and higher RAM. Moreover, while testing, Windows Operating System was used. On Linux based System, the time taken will be much shorter. In terms of compression size, Huffman coding can be widely improved by using Recursive Splitting.