Text Processing – Text Compression

By Vijeta Agrawal

Aim:

The main focus of this report is the code implementation of the Text Compression/Decompression that we have created based on Huffman Coding Algorithm, its performance analysis under different models (word/symbols) and results drawn from it.

Code Implementation:

huff-compress.py: To begin with **NodeMaker** class, we are creating data structure for nodes **(root, left_leaf, right_leaf)**, then we create **CodeForHuffman** class to code Huffman compression. First of all we take input text file as **{filename}.txt** and read it, then we create two functions **CountWord** and **CountSymChar** to create symbol models for words and char using appropriate regular expressions respectively. In both functions, we create two dictionaries to store regex resulted elements and elements with their frequency of occurrence in the text file and return them.

In **Sorter** function, first we are sorting the items in the **ascending order/pre-order** to get the **least frequent occurring element at top** and most frequent to the end. Then we are calculating the **probability** for each element and swapping the positions of keys and values, now we return the sorted element dictionary **(sorted_WordDictionary)** of elements with label and probability of frequencies.

After this we move forward to make the **Huffman tree and Binary array with Make_Huffman_Tree and TreeTraverse** functions, we initiate a **huff_tree** list and append all the elements of **sorted_WordDictionary** as nodes using **NodeMaker**. Then we make the tree following the rules by creating a new node **(Element3)** and adding to it the frequency probabilities as well as the string values of the lowest nodes **(Element1, Element2)** and storing it to the new node **(Element3)**. After this we remove those nodes **(Element1, Element2)** from the list and append the new node **(Element3)** to the **huff_tree** list. Then **we sort the list again** with respect to frequency probabilities and set the new node **(Element3)** as the **root/parent node** for the previously lowest elements **(Element1, Element2)**. We repeat the above process till we complete the whole tree. In **TreeTraverse** function, we traverse the tree and create the dictionary **(Final_dict)** with **binary code** for the elements by adding '1' for left and '0' for right traverse. Then we save the symbol model to **{filename}-symbol-model.pkl** by using **pickle** module of python. We return the final dictionary **(Final_dict)** for final encoding process.

In **Encoder** function, we create the binary string for the entire text by using each element in **listofwords** and adding the binary code of each element to the binary string **(Final_string)**. We create the proper binary code of the file by removing the '0B' padding by using **CreatingBitArray** function and save it to **{filename}.bin file**.

In **CommandLine** class we construct the options of **arguments (word/symbol models)** to use while running the **huff-compress.py** in cmd.

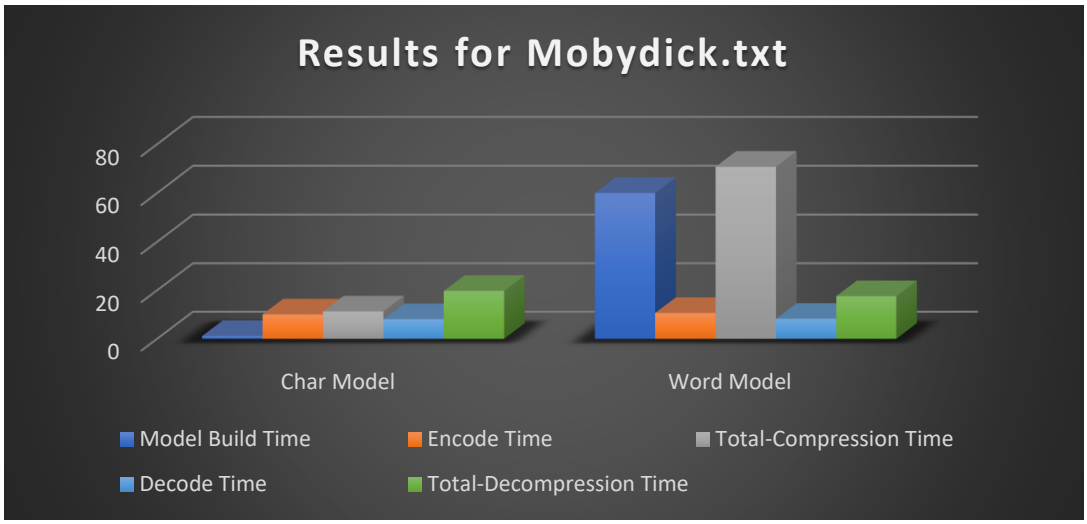
In **__main__**, we create an instance of **CodeForHuffman class (InstanceForHuffman)** and call each function. We also use **time** module to calculate the time taken to build the symbol model and Encoding.

huff-decompress.py: We take the **{filename}-symbol-model.pkl** and **{filename}.bin file** as input and initialize them. In **Convert** function, we add the '0B' padding to the code that we removed after encoding in compression and get the padded file **(Binary)** to decompress it. Now in **CodeReader_ReaderDict**, we are reading the compressed files.

Then in the **Decoder** function, we check the **binary code** against the elements **(Binary)** and as soon as it matches, we add it to **StrDecoded**. When we get the complete file, we save the output to **{filename}-decompressed.txt**.

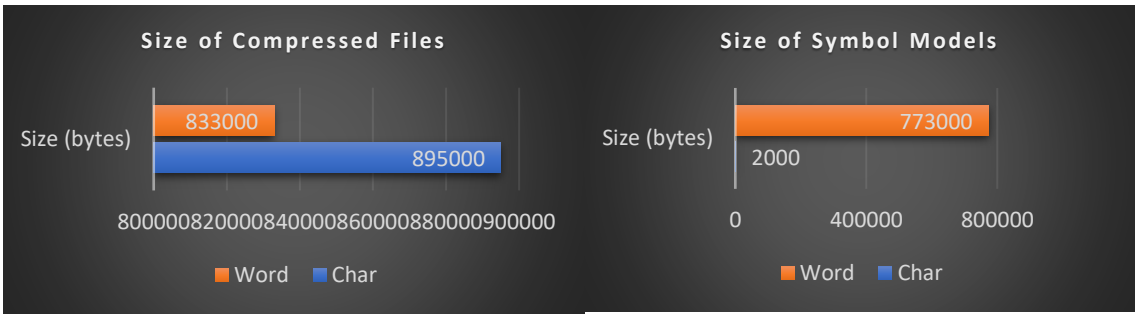
Model Performance:

To analyse the performance of the huff-compress.py/huff-decompress.py on Char/Word model, we are analyzing the time taken on different steps by each model. We are using Test-harness.py on Mobydick.txt for testing and results are shown below:



Observations based on Performance Analysis:

- The compression performance of character-based Huffman is better than word based Huffman, which can be easily observed through the above tables.
- **Size of Files:**



- **Time taken:**

Time Taken	Char Model	Word Model
Model Build Time	1.211579731	60.09393471
Encode Time	10.15393954	10.72823031
Total-Compression Time	11.36604397	70.82274154
Decode Time	8.158789085	8.328326642
Total-Decompression Time	19.84869239	17.67346218

- The various aspects of the performance shown above can be improved by optimizing the code further like using heapq module for making the min-tree.
- We can try to reduce the size and time taken for building the Symbol model for word-based Huffman.
- We can try and compress the char based Huffman model into smaller size.