Huffman Code Text Compression and Decompression

Hao Xu

I. Introduction

Huffman coding is a simple but lossless algorithm for data compression. In this report, a model for encoding Huffman code has been developed. After describing the implementation and presenting the result, compression performances based on both character-based and word-based Huffman has been compared and evaluated.

II. Model

A. Encoding

Formally, in an encoding task we have a set of symbols called alphabet $A = \{a_1, a_2, ..., a_n\}$ and their corresponding weight $W = \{w_1, w_2, ..., w_n\} s.t.\ \sum_i w_i = 1$. A **code** C is a binary-string valued function over A, usually its image $C(A)$ is call the set of code words. What we want to look for is a code C that minimizes the quantity:

$$L(C) = \sum_{i=1}^{n} w_i \times \text{length}(C(a_i))$$

From information theory we can prove that for all possible code function, L is lower bounded by the entropy of W:

$$L(C) \geq H(W) = -\sum_i w_i \log w_i \quad \forall C$$

B. Huffman code

The Huffman code is one of the optimal prefix-free codes. A code C is **prefix-free** if no code word is a prefix of any other code word in $C(A)$. In this report I consider two specific implementation of Huffman code for text compression:

- *Char based*: Treat any character as symbol
- *Word based*: Treat word and only non-letter character as symbol

For both cases the weight for each symbol is its frequency in the text.

III. Implementation

Packages used: argparse, re, array, bisect, pickle, os, collections

A. Build Huffman tree & code dictionary

- For *char* mode, *re.split* function was used to split text into a list of words and special characters
- Use *collections.Counter* function to count the distinct elements in that list (or in the text directly in *word* mode), which outputs a dictionary with entries: 'symbol: number of occurrence'
- Declare a *Node* class, construct a *node* object for each entry in the dictionary and them into a list
- Sort the list in ascending order of node's weight (number of occurrence of corresponding symbol)
- Pop the first two nodes in the list and construct a new one from them. New nodes weight is the sum of the two's weight. Repeat the step until there is only one node left, this node is the root of Huffman tree
- Do the tree traversal, build the code dictionary from leaves with each dictionary entry: 'symbol on this leave: binary-string representing the path from root to this leaf'

B. Encoding (Compression)

- For each symbol (a character in *char* mode or a word/non-letter character in *word*) mode) in the text, get it code word by referring to the code dictionary.
- Concatenate all code words into a single binary string. This string is the encoded text.

C. Decoding (Decompression)

- Define a pointer of node *p* which initially points at the root of the Huffman code tree.
- Scan the encoded text bit by bit, point *p* to its left child if the bit is '0', otherwise point *p* to its right child. Whenever *p* is pointing at a leaf, append the symbol represented by this leaf to a list.
- Concatenate all symbols in the list, this string is the decoded text and should be same as the original one.

IV. Result

The Project Gutenberg version of Melville's novel 'moby-dick.txt' is supplied as a data file for development and testing. The size of this file is 1220.15KB.

	'char'	'word'
Compressed file size Mode	675KB	383KB
Symbol model size	6KB	1433KB
Model building time	0.0056s	0.394s
Encoding time	1.072s	0.892s
Decoding time	3.105s	1.914s

V. Discussion

In the 'char' mode, the model size is pretty small (6KB) and building model is quick (0.005s), but the compressed file is not that small (675KB), and decoding time is relatively long (3s).

In the 'word' mode, model size is large (1.4MB) and it takes a relatively long time to build the model (0.4s), but the compressed file is smaller (383KB), also the decoding time is shorter.

The encoding time is not significantly different between two modes.

Regarding the time of compression and decompression, changing to another language, like C++, will probably reduce the executing time of all parts. Also, due to the clumsiness of Python class , the model size will be smaller in C++.

Regarding the compression rate, we can only do better by changing to other algorithms. Taking the correlation between adjacent word into consideration might further increase the compression rate. For example, n-gram models.

REFERENCES

[1] Huffman D A. A method for the construction of minimum-redundancy codes[J]. Proceedings of the IRE, 1952, 40(9): 1098-1101.

[2] Cover T M, Thomas J A. Elements of information theory[M]. John Wiley & Sons, 2012.