Department of Electrical Engineering
Indian Institute of Technology Kharagpur
Machine Learning for Signal Processing Laboratory
(EE69210)
Spring, 2022-23

Experiment 2: Lossless Compression

Grading Rubric

	Tick the best applicable per row			Points
	Below Expectation	Lacking in Some	Meets all Expectation	
Completeness of the report				
Organization of the report (5 pts)				
Quality of figures (5 pts)				
Building the min-heap (20 pts)				
Generating the code book and compressed bit stream (25 pts)				
Ability to decode the Huffman tree from its in-order traversal representation (20 pts)				
Ability to recover the dataset (25 pts)				
			TOTAL (100 pts)	

1 Introduction

Compression of a signal involves multiple kinds of approaches. Since a sample of signal my be represented in a dimension much higher than its intrinsic dimension, so one kind of approaches involves operations to reduce its dimension. Principal component analysis (PCA) or Singular value decomposition (SVD) are few of such approaches to reduce the intrinsic dimensionality of a signal. Alternatively, operations like Discrete Fourier transform (DFT), Discrete cosine transform (DCT) also play a role of compacting the energy of the signal, by reducing the effective number of samples in the transformed domain which may be needed to recover the original domain signal. Such kind of approaches are generally referred to as energy compacting transforms and also offer lossy compression of the signal. The other family of operations involve reducing the number of bits required to represent a sample. Since the number of samples or its dimension is not effected, these kind of transforms do not typically lead of any loss of energy of the signal. Such family of operations are known as lossless compression.

In this experiment we would implement the mechanism of lossless compression of a vector representing a set of samples using principles of Entropy of values of samples.

2 Huffman coding

The method of Huffman coding is adopted to minimize the number of bits required to represent a sample. The minimum limit being the Entropy of values which make up the samples.

Let us consider a vector $\mathbf{X}$ constituted of the following samples $\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_i, \cdots, \mathbf{x}_{N-1}]$, where $\mathbf{x}_i \in \mathbb{R}^{D\times 1}$. As an example we can consider the string $\mathbf{X}$ ={'h','e','l','l','o'}. If every alphabet is represented as 8-bit ASCII, then we would have $\mathbf{x}_i \in \mathbb{B}^{8\times 1}$, where $\mathbb{B} \in \{0, 1\}$ represents the set of bits. Thus we have $\mathbf{X} \in \mathbb{B}^{8\times 5}$. So the average number of bits per sample is 8. However, it can be seen that since the number of unique symbols in this case is only 4, so one may say that only 2 bits per symbol is enough. However, this number of bits per symbol would vary with the constituents of $\mathbf{X}$. Huffman coding is one such way to evolve a reduced code representation for each sample, and interestingly, through this approach the number of bits per sample is not fixed, and this variable number of bits per sample enables us to reduce the average number of bits per sample.

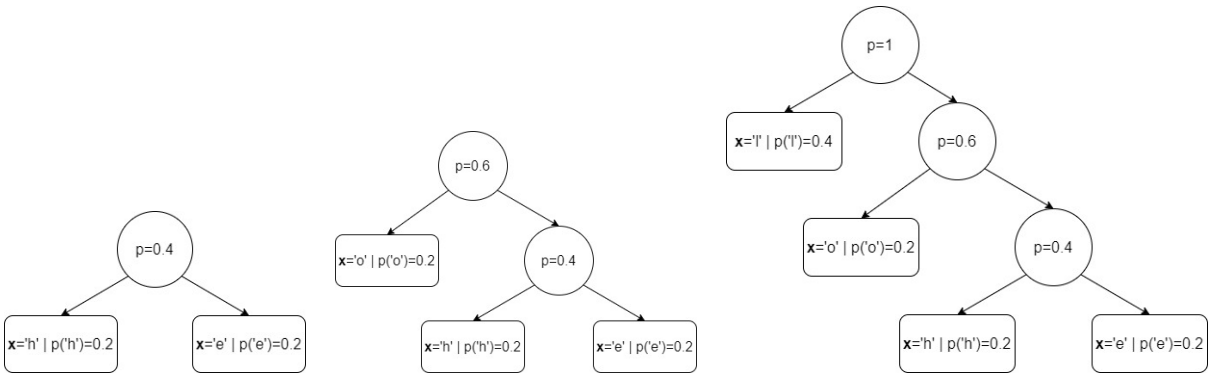
2.1 Constructing the Huffman tree

The process starts with identifying the number of unique symbols in $\mathbf{X} \in \mathbb{R}^{D\times N}$ say represented as $\mathbf{U} = [\mathbf{u}_0, \mathbf{u}_1, \cdots, \mathbf{u}_j, \cdots, \mathbf{u}_{k-1}] \in \mathbb{R}^{D\times k}$. In this case we have the following $k = 4$ unique symbols $\mathbf{U}$ ={'h','e','l','o'}$\in \mathbb{B}^{8\times 4}$. Let us now find the probability of occurrence of each of the unique symbols in $\mathbf{X}$ as $p(\mathbf{u}_i)$. Here we have $p(\text{'h'}) = 0.2, p(\text{'e'}) = 0.2, p(\text{'l'}) = 0.4, p(\text{'o'}) = 0.2$.

Now, let us build a min-heap such that a left child n_a of the parent node n_c has $p(n_a) \leq p(n_b)$ where n_b is the right child. Also, $p(n_c) = p(n_a) + p(n_b)$, and any leaf represents one of the unique symbols in $\mathbf{U}$, such that if n_a is a leaf node with the symbol $\mathbf{u}_i$, then $p(n_a) = p(\mathbf{u}_i)$. Here we can start with creating the min-heap in steps as shown in Fig. 1. This binary tree representing the heap is referred to as the *Huffman tree*.

2.2 Code book generation

The Huffman tree can be used to generate the variable length bit code to be assigned to each symbol accordingly. Starting at a node n_c, traversal to its left child n_a is assigned a bit code of 0 while traversal to its right child n_b is assigned the bit code of 1. When we encounter a child to be a leaf node with a symbol, then the bit code assigned to that symbol corresponds to the accumulation of all such bits starting at root node of the min-heap. The tree would be



(a) Step 1 (b) Step 2 (c) Step 3

Figure 1: Steps wise evolution of the min-heap.

represented similar to as in Fig. 2. Now, using this tree we can find the bit code for the symbols as in Table 1.

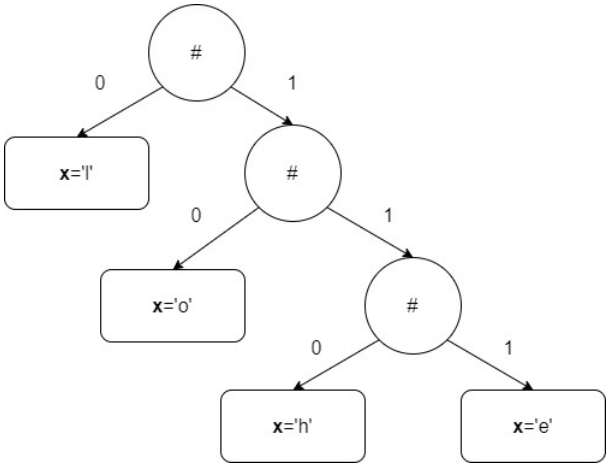


Figure 2: Huffman tree created for the dataset in example. # denotes a non-leaf node, and the symbol present at a leaf node is appropriately mentioned viz. $\mathbf{x}$ ='l', etc.

$\mathbf{u}_i$	Bit code
'h'	110
'e'	111
'l'	0
'o'	10

Table 1: Bit code for each symbol $\mathbf{u}_i$ generated using the Huffman tree.

2.3 Coded bit stream

Using the bit code for the symbols in Table 1 we can now represent the dataset $\mathbf{X}$ using a concatenated string of bits, by replacing each sample by the corresponding bit code of its unique symbol. This would provide us with a new dataset $Y \in \mathbb{B}^l$ where l is the total number of bits in Y. Thus we obtain for this example $Y = [1101110010]$.

2.4 Decoding the bit stream

In order to recover the original form of the dataset stored as $\hat{\mathbf{X}}$ from the received bit stream Y, we would again have to employ the Huffman tree. We start the process accordingly by reading one bit at a time and traversing through the *Huffman tree* till we reach a leaf node. As with this example we would see the following steps being adopted.

1. $Root \rightarrow Right \rightarrow Right \rightarrow Left$ so as to reach $\mathbf{x}$ ='h'.
   Store 'h' as the first sample in the recovered dataset $\hat{\mathbf{X}}$.
2. Reset tree pointer to Root.
3. $Root \rightarrow Right \rightarrow Right \rightarrow Right$ so as to reach $\mathbf{x}$ ='e'.
   Store 'e' as the second sample in the recovered dataset $\hat{\mathbf{X}}$.
4. Reset tree pointer to Root.
5. $Root \rightarrow Left$ so as to reach $\mathbf{x}$ ='l'.
   Store 'l' as the third sample in the recovered dataset $\hat{\mathbf{X}}$.
6. Reset tree pointer to Root.
7. $Root \rightarrow Left$ so as to reach $\mathbf{x}$ ='l'.
   Store 'l' as the fourth sample in the recovered dataset $\hat{\mathbf{X}}$.
8. Reset tree pointer to Root.
9. $Root \rightarrow Right \rightarrow Left$ so as to reach $\mathbf{x}$ ='o'.
   Store 'o' as the fifth sample in the recovered dataset $\hat{\mathbf{X}}$.

2.5 Building the Encoder-Decoder

Given the task of lossless compression of the dataset $\mathbf{X}$ using Huffman coding, the following steps would have to undertaken.

2.5.1 Encoder

The encoder would perform the following tasks

1. Generate the set of unique symbols $\mathbf{U}$ from the dataset $\mathbf{X}$.
2. Compute the probability of occurance of each unique symbol as $p(\mathbf{u}_i)$.
3. Construct the min-heap.
4. Generate the code book for each unique symbol $\mathbf{u}_i$.
5. Generate the coded bit stream Y.
6. Encode the *Huffman tree* with in-order traversal, also known as the *Code book*.

2.5.2 Compressed file

The compressed file should consist of the *Huffman tree* and the coded bit stream Y.

2.5.3 Decoder

The decoder would perform the following set of steps to recover the dataset

1. Reconstruct the *Huffman tree* using the *Code book*.
2. Using traversal on the *Huffman tree* recover $\hat{\mathbf{X}}$ from Y.

3 Experiments

3.1 Dataset 1

3.1.1 Generation of the dataset

Generate a lower case alphabet and numeric string of a given length N consisting of random characters without blank spaces or line breaks or special characters. You can use the `string` and `random` library in Python to achieve it. This would denote the dataset $\mathbf{X}$. Store the generated string as a .txt file.

3.1.2 Encoder - Decoder - Compressed file

Write a function for the Huffman encoder which would accept a `.txt` file as an input, and it would produce a compressed file `.huf`.

The `.huf` file would consist of the first Byte stating the number of Bytes which are used to storing the *Huffman tree* in its in-order traversal representation. The second Byte onward would be the *Huffman tree*. On completion of this, the *compressed bit code* would be stored, written in bit format.

Write a function for the Huffman decoder which would accept a `.huf` file as an input, and it would produce the decompressed file .txt.

3.1.3 Assignments to solve and report

Write down the codes for the following tasks in a .ipynb file, including all visualizations and submit the executed file. Submit also a separate pdf of your report of this experiment prepared using Latex and the documentclass Article. This report should describe your observations and reasoning while executing these experiments.

1. Generate $t = 10$ different `.txt` files for each of $N = \{50, 100, 500, 1000, 5000\}$.
2. Use encoder to encode each of the file and store each as a separate `.huf`.
3. Compute the compression factor for each file as `sizeof`($<$ `fileName` $>$ `.txt`)/`sizeof`($<$ `fileName` $>$ `.huf`) where `sizeof`$(\cdot)$ operator returns the size of the file in Bytes. Report the variation of compression factors for each N using a *notch-box* plot. Here *x-axis* should represent N as specified above and *y-axis* should represent the compression factor.
4. Use the decoder to decode each of the `.huf` files.
5. Measure the MSE between the original `.txt` and the decoded `.huf` file.
6. Measure the time to encode and decode each file, and report the times separate as a *notch-box* plot for encoder and for decoder. Here *x-axis* should represent N as specified above and *y-axis* should represent the computation time in sec. Encoder and decoder times to be shown as stacked / grouped items for each N.

3.2 Dataset 2

3.2.1 Preparation of the dataset

Here we would strive to solve Huffman on grayscale images of human faces of size 64×64, from the Olivetti faces dataset[1] with $N = 400$ samples. Since each image $\mathcal{I} \in \mathbb{Z}^{64\times64\times1}$ and has 8-bit grayscale representation, so we can alternatively write it as $\mathbf{X} \in \mathbb{B}^{8\times4,096}$ where $\mathbb{B} = \{0, 1\}$.

3.2.2 Assignments to solve and report

Write down the codes for the following tasks in a .ipynb file, including all visualizations and submit the executed file. Submit also a separate pdf of your report of this experiment prepared using Latex and the documentclass *article*. This report should describe your observations and reasoning while executing these experiments.

1. Over each image in the dataset store it as a .bmp file and then perform the following.
2. Use encoder to encode each of the image in `.bmp` format and store each as a separate `.huf`.
3. Compute the compression factor for each file as `sizeof`($<$ `fileName` $>$ `.bmp`)/`sizeof`($<$ `fileName` $>$ `.huf`) where `sizeof`$(\cdot)$ operator returns the size of the file in Bytes. Report the variation of compression factors over all images using a *notch-box* plot. Here *y-axis* should represent the compression factor.
4. Use the decoder to decode each of the `.huf` files to obtain `.bmp` file.
5. Measure the MSE between the original `.bmp` and the decoded `.huf` file.
6. Measure the time to encode and decode each file, and report the times separately as a *notch-box* plot for encoder and for decoder. Here *y-axis* should represent the computation time in sec. Encoder and decoder times to be shown as stacked / grouped items.
7. Plot the compression factor on *x-axis* and encoder time on *y-axis* over all images as a scatter plot. Repeat the same with decoder time on *y-axis*, and comment on your observation.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_olivetti_faces.html