Entropy Codings

Coding theory, Symbol codes, Stream codes

October 2018

Content

- Uniquely Decodable
- Prefix codes
- Huffman coding
- Arithmetic coding
- LZW coding

Coding theory - Data compression

- How many bits are needed to describe the outcome of an symbol (outcome of a random experiment)?
- Information theory v.s. Coding theory
  - Information theory deals with how best we can compress data, not dealing with how to design coding/decoding algorithms to compress/decompress data
  - Coding theory deals with designing efficient coding scheme to compress/code data

Coding Approach - binary symbol code

- Compression: compressor function that maps $x \to \dot{x} = c(x)$ which is a bit string
- Decompression: decompressor function that maps $\dot{x} \to x = d(\dot{x})$
- x is called "source symbol", $c(x)$ is called codeword of x

How to evaluate efficiency of a symbol coding scheme?

Given a coding scheme C of source X that maps $x_i \rightarrow c(x_i)$. Call l_i length of codeword for symbol x_i
Average code length:

$$L(C, X) = \sum_i p_i l_i \tag{1}$$

- $L(C_1, X) < L(C_2, X)$ means C_1 is better than C_2 in terms of data compression ... if both of them are decodable
- If l_i are equal for all symbols $\rightarrow$ "fix-length coding". Otherwise, ... "variable-length coding"

Decodability

- Given a coding scheme C if we can decode every codeword c_i back to source symbol x_i uniquely, we call C is unquely decodable
- If a source X has 8 symbols totally, can we code every symbols it with only 2 bits?
  - Answer: No, we cann't (pigean hole principle)
- There are two fundamental questions
  - How to test decodability?
  - Is there any fundamental constraint on code lengths for decodability?

Decodability

Symbol	code 1	code 2	code 3	code 4
x_1	0	0	0	0
x_2	0	1	10	01
x_3	1	00	110	011
x_4	10	11	111	0111

- "Code 1"is not uniquely decodable. Codeword $"0"$ can be decoded into 2 different symbols
- "Code 2"is not uniquely decodable. Codeword $"00"$ can be decoded into x_3 or $x_1 x_1$
- "Code 3"and "code 4"are uniquely decodable. Code 3 is prefix code

How to test decodability?

- Two codeword 01 and 01101. 01 is prefix and 101 is dangling suffix
- Prefix code is code that all codeword are not prefix of any other codeword.
- $c' =$ FindPrefix(c): c is prefix of c'
- $sfix =$ Sufx(c', c): $sfix$ is dangling suffix of c' and c

How to test decodability

```
input: list of codewords list
Result: return "decodable" or "undecodable"
while dangling suffix found do
    foreach c in list do
        c' ← FindPrefix(c) ;
        sfix ← Sufx(c',c) ;
        if sfix ∈ list then
            return "undecodable" ;
        else
            add sfix to list ;
        end
    end
end
return "decodable"
```

Prefix codes

- Prefix code is uniquely decodable.
- Prefix code is equivalent to a tree (binary tree), where codewords are leaf nodes
- Prefix code is easily to decode (instanteneous code)

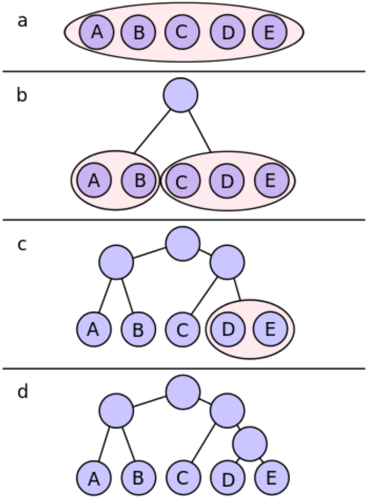
Construct Prefix Code (topdown) - Shanon-Fano code

Given: A set of symbols with their probabilities
Output: a tree presenting a codebook
Algorithm:

1. Sort symbols in order of decreasing probabilities
2. Create a root node that corresponds to the whole sorted set
3. Bisecting the codeword set into left-set and right-set such that both set are ballanced in terms of total probability
4. Assign "0" to left-set and "1" to right-set and create child node accordingly
5. Repeat the procedure for each left-set and right-set until we cannot process further. The resulting tree is the coding tree.

Shanon-Fano code - Example

symbols	count	code
A	15	00
B	7	01
C	6	10
D	6	110
E	5	111



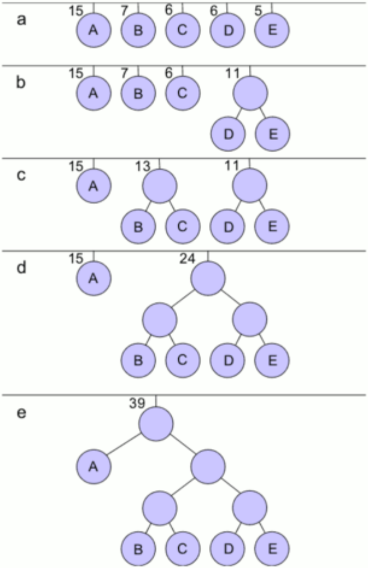
Construct Prefix Code (bottom up) - Huffman code

Given: A set of symbols with their probabilities
Output: a tree presenting a codebook
Algorithm:

1. Sort symbols in order of decreasing probabilities in a queue
2. Create leaf nodes, each corresponds to a symbol
3. Remove two least probable nodes from the queue and create a parent node of the two nodes. The probability of the new node is sum of probs of it child nodes. Put the new node into the queue.
4. Repeat from step 1 until there is a single node in the queue

Huffman code - Example

symbols	count	code
A	15	0
B	7	100
C	6	101
D	6	110
E	5	111



Shannon Code vs. Huffman Code - Optimality

			Shannon		Huffman	
Symbols	count	$I(x)$	code	length	code	length
A	15	1.38	00	2	0	1
B	7	2.48	01	2	100	3
C	6	2.70	10	2	101	3
D	6	2.70	110	3	110	3
E	5	2.96	111	3	111	3
Average/Entropy		2.19		2.28		2.23

Kraft inequality

If l_i are length of codewords c_i, then we have (Kraft-McMillan inequality)

$$\sum_i \frac{1}{2^{l_i}} \leq 1 \tag{2}$$

When equality happens, $C = \{c_i, \forall i\}$ is called complete code

Optimum code length

- If C^* is optimal compressor of X, $L(C, X)$ is $minimized$!
- How to find optimum code length?:

$$L(C, X) \geq H(X) \tag{3}$$

Intepretation:

- Average code length v.s. Entropy: An information source X has entropy H(X) and is coded with a compressor C that has average code length $L(C, X)$
  - Entropy $H(X)$: average information content of all symbols of the source
  - Average Code Length $L(C, X)$: average amount of data (bits) that is used to represent $H(X)$

Shanon source coding theorem

Theorem:

- N i.i.d. random variables, each with entropy H(X) can be compressed into more than $NH(X)$ bits with negligible risk of information loss, as $N \to \infty$

Theorem - symbol code version

- There exists a prefix code with expected code length $L(C, X)$ for source X that satisfies

$$H(X) \leq L(C, X) < H(X) + 1 \tag{4}$$

Optimality of symbol codes

- Huffman code is optimal, Shannon code is suboptimal
  - We cannot find other symbol code that is better than Huffman code
- Disadvantages of Huffman code
  - Poor performance when symbol frequencies are incorrect
  - Poor performance when symbol frequencies are changing
  - The gap from optimality $L(C, X) - H(X) \leq 1bit$ is negligible when $H(X)$ is large. But if $H(X)$ is small, it is noticeable.
- We can improve further if we use other coding approach other than symbol code (E.g.: Stream code)

Stream code - arithmetic code

- Code one symbol at a time cannot improve further to Shannon limit -> Encode a block of symbols at a time

Original source symbols:

$$x_1 \quad x_2 \quad x_3$$

Combined source symbols:

$$\begin{array}{ccc} x_1x_1 & x_1x_2 & x_1x_3 \\ x_2x_1 & x_2x_2 & x_2x_3 \\ x_3x_1 & x_3x_2 & x_3x_3 \end{array}$$

Stream code - Arithmetic code

- Arithmetic code defines a way to efficiently encode a sequence of symbols into binary string
- Each sequence of symbols $\Longleftrightarrow$ a number in unit interval $[0, 1)$ (tag value)

Given:

- Source $X = a_i | \forall i$,
- Probability (pdf) of each symbol: $Pr(X = a_i) = P(a_i) = p_i$
- Cdf of each symbol: $F_X(i) = \sum_{k=1}^{i} P(X = k) = \sum_{k=1}^{i} p_k$

Assign each symbols onto unit interval $[0, 1)$:

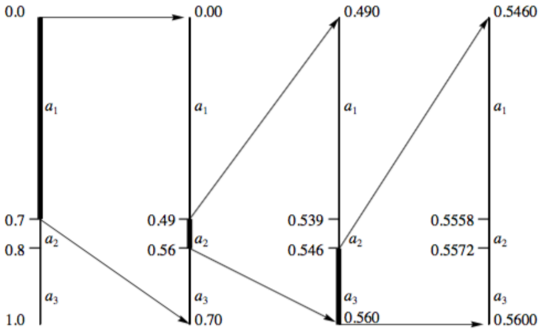
- Symbol i is mapped to $[F_X(i-1), F_X(i))$ (*)
- Denote $I(i) = [F_X(i-1), F_X(i))$

Arithmetic coding

Coding Algorithm: *find tag value for a sequence*

1. Start with interval $I = [0, 1)$
2. Map symbols on I according to (*)
3. Read a symbol x from source.
4. Let $I = I(x)$. Repeat to step 1

Arithmetic coding - example



Arithmetic coding - Interval contains tag value

$$interval = [l^{(k)}, u^{(k)})$$

$$
\begin{array}{ll}
sequence & interval \\
NULL : & \left\{ \begin{array}{lcl} l^{(0)} & = & 0 \\ u^{(0)} & = & 1 \end{array} \right. \\
a_i : & \left\{ \begin{array}{lcl} l^{(1)} & = & F_X(i-1) \\ u^{(1)} & = & F_X(i)) \end{array} \right. \\
a_i a_j : & \left\{ \begin{array}{lcl} l^{(2)} & = & F_X(i-1) + F_X(j-1)(F_X(i) - F_X(i-1)) \\ & = & l^{(0)} + F_X(j-1)(u^{(0)} - l^{(0)}) \\ u^{(2)} & = & F_X(i-1) + F_X(j)(F_X(i) - F_X(i-1)) \\ & = & l^{(0)} + F_X(j)(u^{(0)} - l^{(0)}) \end{array} \right.
\end{array}
$$

- Recursively identify intervals as receiving symbols
- The more symbols received, the narrower intervals become
- The mid-point of the final interval can be used as tag value for a sequence.