Data Compression = Modeling + Coding

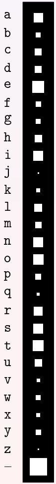
November 8, 2015

uniform codes

0	00	000	0000
			0001
		001	0010
			0011
	01	010	0100
			0101
		011	0110
			0111
1	10	100	1000
			1001
		101	1010
			1011
	11	110	1100
			1101
		111	1110
			1111

Symbol codes

i	a_i	p_i
1	a	0.0575
2	b	0.0128
3	c	0.0263
4	d	0.0285
5	e	0.0913
6	f	0.0173
7	g	0.0133
8	h	0.0313
9	i	0.0599
10	j	0.0006
11	k	0.0084
12	l	0.0335
13	m	0.0235
14	n	0.0596
15	o	0.0689
16	p	0.0192
17	q	0.0008
18	r	0.0508
19	s	0.0567
20	t	0.0706
21	u	0.0334
22	v	0.0069
23	w	0.0119
24	x	0.0073
25	y	0.0164
26	z	0.0007
27	–	0.1928



Some definitions

Definition

A *code (source code)* C for a random variable X is a mapping from $\{x_1, x_2, \ldots, x_n\}$ to $\mathcal{D}*$the set of finite-length strings of symbols from an alphabet of length D.

Note:
- $C(x)$ the **codeword** of x
- $l(x)$ the length of codeword $C(x)$

Definition

The expectation length $L(C)$ of a source code $C(X)$ for a random variable X with probability mass function $p(x)$is given by:

$$L(C) = \sum_{x_i} p(x_i) l(x_i) = \sum_i p_i l_i$$

source code examples

Example

Unary codes
ASCII (American Standard Code for Information Interchange) codes
- 96 printable keyboard characters
- log(96) ??

Morse code
- special stop symbol

Definition

A code is called a *prefix code* or *instantaneous code (prefix-free code)* if no codeword is a prefix of any other codeword.

Kraft-McMillan Inequality.

Theorem

For any ***uniquely decodable code*** *C*

$$\sum_{x \in C} 2^{-l(x)} \leq 1,$$

where $l(x)$ is the length of the codeword. Also, for any set of lengths L such that: $\sum_{l \in L} 2^{-l} \leq 1$, there is a prefix code C of the same size such that $l(x) = l_{i, i \in [1, \ldots, |L|]}$

Prefix code application

- **UTF-8 :** Universal Character Set Transformation Format—8-bit
  - defined in 1992 as an extention for ASCII
  - is a variable-width encoding
  - default character encoding in operating systems, programming languages, APIs, and software applications
  - labelled "Unicode"
  - the most common encoding for HTML files

Prefix code application

- **UTF-8 :** Universal Character Set Transformation Format—8-bit
  - defined in 1992 as an extention for ASCII
  - is a variable-width encoding
  - default character encoding in operating systems, programming languages, APIs, and software applications
  - labelled "Unicode"
  - the most common encoding for HTML files

Prefix code application

- **UTF-8 :** Universal Character Set Transformation Format—8-bit
    - defined in 1992 as an extention for ASCII
    - is a variable-width encoding
    - default character encoding in operating systems, programming languages, APIs, and software applications
    - labelled "Unicode"
    - the most common encoding for HTML files

Prefix code application

- **UTF-8 :** Universal Character Set Transformation Format—8-bit
  - defined in 1992 as an extention for ASCII
  - is a variable-width encoding
  - default character encoding in operating systems, programming languages, APIs, and software applications
  - labelled "Unicode"
  - the most common encoding for HTML files

Prefix code application

- **UTF-8 :** Universal Character Set Transformation Format—8-bit
  - defined in 1992 as an extention for ASCII
  - is a variable-width encoding
  - default character encoding in operating systems, programming languages, APIs, and software applications
  - labelled "Unicode"
  - the most common encoding for HTML files

Prefix code application

- **UTF-8 :** Universal Character Set Transformation Format—8-bit
  - defined in 1992 as an extention for ASCII
  - is a variable-width encoding
  - default character encoding in operating systems, programming languages, APIs, and software applications
  - labelled "Unicode"
  - the most common encoding for HTML files

UTF-8, 16, 32

Bits of code point	First code point	Last code point	Bytes in sequence	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	U+0000	U+007F	1	0xxxxxxx					
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx				
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx			
21	U+10000	U+1FFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	U+200000	U+3FFFFFF	5	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	U+4000000	U+7FFFFFFF	6	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Source Coding Theorem

Theorem

There exists a variable-length encoding C of an ensemble X such that the average length of an encoded symbol $L(C, X)$ satify:

$$H(X) \leq \quad L(C, X) \quad < H(X) + 1.$$

Huffman Codes

- Are **optimal** prefix codes
- Generated from a set of probabilities
- David Huffman developed the algorithm as a student on information theory at MIT in 1950
- The professor, Robert M. Fano proposed the problem of finding the most efficient binary code.
- The algorithm is the most used component of compression algorithms
- used as the back end of GZIP, JPEG

Huffman Codes

- Are **optimal** prefix codes
- Generated from a set of probabilities
- David Huffman developed the algorithm as a student on information theory at MIT in 1950
- The professor, Robert M. Fano proposed the problem of finding the most efficient binary code.
- The algorithm is the most used component of compression algorithms
- used as the back end of GZIP, JPEG

Huffman Codes

- Are **optimal** prefix codes
- Generated from a set of probabilities
- David Huffman developed the algorithm as a student on information theory at MIT in 1950
- The professor, Robert M. Fano proposed the problem of finding the most efficient binary code.
- The algorithm is the most used component of compression algorithms
- used as the back end of GZIP, JPEG

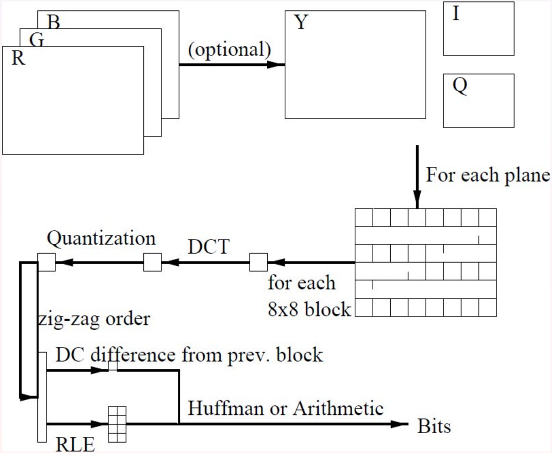
Huffman Codes

- Are **optimal** prefix codes
- Generated from a set of probabilities
- David Huffman developed the algorithm as a student on information theory at MIT in 1950
- The professor, Robert M. Fano proposed the problem of finding the most efficient binary code.
- The algorithm is the most used component of compression algorithms
- used as the back end of GZIP, JPEG

Huffman Codes

- Are **optimal** prefix codes
- Generated from a set of probabilities
- David Huffman developed the algorithm as a student on information theory at MIT in 1950
- The professor, Robert M. Fano proposed the problem of finding the most efficient binary code.
- The algorithm is the most used component of compression algorithms
- used as the back end of GZIP, JPEG

JPEG


B
G
R
(optional)
Y
I
Q
For each plane
Quantization
DCT
for each
8x8 block
zig-zag order
DC difference from prev. block
Huffman or Arithmetic
Bits
RLE

The Huffman algorithm

How it generates the prefix-code tree.

- Start with a forest of trees, one for each message. Each tree contains a single vertex with weight $w_i = p_i$
- Repeat until only a single tree remains
  - Select two trees with the lowest weight roots (w_1 and w_2)
  - Combine them into a single tree by adding a new root with weight $w_1 + w_2$, and making the two trees its children

The Huffman algorithm

How it generates the prefix-code tree.

- Start with a forest of trees, one for each message. Each tree contains a single vertex with weight $w_i = p_i$
- Repeat until only a single tree remains
  - Select two trees with the lowest weight roots (w_1 and w_2)
  - Combine them into a single tree by adding a new root with weight $w_1 + w_2$, and making the two trees its children.

The Huffman algorithm

How it generates the prefix-code tree.

- Start with a forest of trees, one for each message. Each tree contains a single vertex with weight $w_i = p_i$
- Repeat until only a single tree remains
  - Select two trees with the lowest weight roots (w_i and w_2)
  - Combine them into a single tree by adding a new root with weight $w_1 + w_2$, and making the two trees its children.

The Huffman algorithm

How it generates the prefix-code tree.

- Start with a forest of trees, one for each message. Each tree contains a single vertex with weight $w_i = p_i$
- Repeat until only a single tree remains
  - Select two trees with the lowest weight roots (w_i and w_2)
  - Combine them into a single tree by adding a new root with weight $w_1 + w_2$, and making the two trees its children.

The Huffman algorithm

How it generates the prefix-code tree.

- Start with a forest of trees, one for each message. Each tree contains a single vertex with weight $w_i = p_i$
- Repeat until only a single tree remains
  - Select two trees with the lowest weight roots (w_i and w_2)
  - Combine them into a single tree by adding a new root with weight $w_1 + w_2$, and making the two trees its children.

The Huffman implementation

Implementation with priority queue:

- Create a leaf node for each symbol and add it to the priority queue.
- While there is more than one node in the queue:
  - Pop the two nodes of highest priority (lowest probability) from the queue
  - Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
  - Add the new node to the queue.
- The remaining node is the root node and the tree is complete.

Complexity: priority queue insert $O(\log n) \rightarrow$ Huffman $O(n \, \log n)$

The Huffman implementation

Implementation with priority queue:

- Create a leaf node for each symbol and add it to the priority queue.
- While there is more than one node in the queue:
  - Pop the two nodes of highest priority (lowest probability) from the queue
  - Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
  - Add the new node to the queue.
- The remaining node is the root node and the tree is complete.

Complexity: priority queue insert $O(\log n) \rightarrow$ Huffman $O(n \log n)$

The Huffman implementation

Implementation with priority queue:

- Create a leaf node for each symbol and add it to the priority queue.
- While there is more than one node in the queue:
  - Pop the two nodes of highest priority (lowest probability) from the queue
  - Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
  - Add the new node to the queue.
- The remaining node is the root node and the tree is complete.

Complexity: priority queue insert $O(\log n) \rightarrow$ Huffman $O(n \; \log n)$

The Huffman implementation

Implementation with priority queue:

- Create a leaf node for each symbol and add it to the priority queue.
- While there is more than one node in the queue:
  - Pop the two nodes of highest priority (lowest probability) from the queue
  - Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
  - Add the new node to the queue.
- The remaining node is the root node and the tree is complete.

Complexity: priority queue insert $O(\log n) \rightarrow$ Huffman $O(n \ \log n)$

The Huffman implementation

Implementation with priority queue:

- Create a leaf node for each symbol and add it to the priority queue.
- While there is more than one node in the queue:
  - Pop the two nodes of highest priority (lowest probability) from the queue
  - Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
  - Add the new node to the queue.
- The remaining node is the root node and the tree is complete.

Complexity: priority queue insert $O(\log n) \to$ Huffman $O(n \ \log n)$

The Huffman implementation

Implementation with priority queue:

- Create a leaf node for each symbol and add it to the priority queue.
- While there is more than one node in the queue:
  - Pop the two nodes of highest priority (lowest probability) from the queue
  - Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
  - Add the new node to the queue.
- The remaining node is the root node and the tree is complete.

Complexity: priority queue insert $O(\log n) \rightarrow$ Huffman $O(n \log n)$

The Huffman in $O(n)$

Implementation with 2 queues:

- Create a leaf node for each symbol and add it to the first queue in increasing order
- While there is more than one node in the both queues:
  - Remove the two nodes of lowest weight from the both queues
  - Create a new internal node with these two nodes as children and with weight equal to the sum of the two nodes' probabilities
  - Add the new node at the end of the second queue
- The remaining node (should apear in the second) is the root node and the tree is complete.

The Huffman in $O(n)$

Implementation with 2 queues:

- Create a leaf node for each symbol and add it to the first queue in increasing order
- While there is more than one node in the both queues:
  - Remove the two nodes of lowest weight from the both queues
  - Create a new internal node with these two nodes as children and with weight equal to the sum of the two nodes' probabilities.
  - Add the new node at the end of the second queue.
- The remaining node (should apear in the second) is the root node and the tree is complete.

The Huffman in $O(n)$

Implementation with 2 queues:

- Create a leaf node for each symbol and add it to the first queue in increasing order
- While there is more than one node in the both queues:
  - Remove the two nodes of lowest weight from the both queues
  - Create a new internal node with these two nodes as children and with weight equal to the sum of the two nodes' probabilities.
  - Add the new node at the end of the second queue.
- The remaining node (should apear in the second) is the root node and the tree is complete.

The Huffman in $O(n)$

Implementation with 2 queues:

- Create a leaf node for each symbol and add it to the first queue in increasing order
- While there is more than one node in the both queues:
  - Remove the two nodes of lowest weight from the both queues
  - Create a new internal node with these two nodes as children and with weight equal to the sum of the two nodes' probabilities.
  - Add the new node at the end of the second queue.
- The remaining node (should apear in the second) is the root node and the tree is complete.

The Huffman in $O(n)$

Implementation with 2 queues:

- Create a leaf node for each symbol and add it to the first queue in increasing order
- While there is more than one node in the both queues:
  - Remove the two nodes of lowest weight from the both queues
  - Create a new internal node with these two nodes as children and with weight equal to the sum of the two nodes' probabilities.
  - Add the new node at the end of the second queue.
- The remaining node (should apear in the second) is the root node and the tree is complete.

The Huffman in $O(n)$

Implementation with 2 queues:

- Create a leaf node for each symbol and add it to the first queue in increasing order
- While there is more than one node in the both queues:
  - Remove the two nodes of lowest weight from the both queues
  - Create a new internal node with these two nodes as children and with weight equal to the sum of the two nodes' probabilities.
  - Add the new node at the end of the second queue.
- The remaining node (should apear in the second) is the root node and the tree is complete.

Adaptive Huffman coding

- Involves calculating the probabilities dynamically
- It is used rarely in practice because of the cost of updating the tree