Adaptive Huffman coding

If we want to code a sequence from an unknown source using Huffman coding, we need to know the probabilities of the different symbols.

Most straightforward is to make two passes over the sequence. First we calculate statistics of the different symbols and then we use the estimated probabilities to code the source.

Instead we would like to do everything in one pass. In addition we would like to have a method that automatically adapts if the statistics of the source changes.

Adaptive Huffman coding, cont.

Simple method:

1. Start with a maximally "flat" code tree.
2. Code N symbols from the source and at the same time gather statistics, ie count how many times each symbol appears. Build a new Huffman tree with the new estimated probabilities.
3. Repeat from 2.

No side information about the tree structure need to be transmitted, since the decoder has access to the same data as the coder.

The smaller N is chosen to be, the faster the coder adapts to a change in source statistics. On the other hand we have to construct a new Huffman tree more often which takes time.

Adaptive Huffman coding, cont.

Smarter method: Adjust the code tree after each coded symbol. We need to keep track of some extra information in each node of the tree.

A binary tree with L leaves has $2L - 1$ nodes.

Give each node a number between 1 and $2L - 1$.

Each node has a weight. For a leaf (outer node) the weight is the number of times the corresponding symbol has appeared (cf. probability). For an inner node the weight is the sum of the weights of its children.

If node j has the weight w_j we need

$$w_1 \leq w_2 \leq \ldots \leq w_{2L-1}$$

Nodes with number $2j - 1$ and $2j$ should have the same parent and the parent should have a higher number than its children.

Trees with these properties are huffman trees.

Adaptive Huffman coding, cont.

Start with a maximally flat code tree (corresponding to a fixlength code if $L = 2^k$). The weight in each leaf is set to 1, and the weight in each inner node is set to the sum of its children's weights. Enumerate the nodes so that the requirements are met.

For each symbol to be coded:

1. Send the codeword corresponding to the symbol.
2. Go to the symbol's corresponding node.
3. Consider all nodes with the same weight as the current node. If the current node is not the node with highest number we switch places (ie move weight, pointers to children and possible symbol) between the current node and the node with highest number.
4. Increase the weight of the current node by 1.
5. If we are in the root node we are done, otherwise move to the parent of the current node and repeat from 3.

Adaptive Huffman coding, cont.

Since the update of the tree is done after coding a symbol, the decoder can do the same update of the code after decoding a symbol. No side information about the tree needs to be transmitted.

One variant is to not start with a full tree. Instead we introduce an extra symbol (NYT, Not Yet Transmitted) and start with a "tree" that only contains that symbol, with weight 0 and number $2L - 1$.

When you code a symbol that hasn't been seen before it is coded with the codeword for NYT, followed by a fixlength codeword for the new symbol. The old NYT node is then split into two branches, one for the NYT symbol and one for the new symbol. The new NYT node keeps the weight 0, the new symbol node gets the weight 1. If the new symbol is the last symbol not yet coded in the alphabet we don't need to split, we can just replace NYT with the new symbol.

Modified algorithm

1. If the symbol hasn't been coded before, transmit the codeword for NYT followed by a fixlength codeword for the new symbol, otherwise transmit the codeword corresponding to the symbol.
2. If we coded a NYT split the NYT node into two new leaves, one for NYT with weight 0 and one for the new symbol with weight 1. The node numbers for the new nodes should be the two largest unused numbers. If it was the last not yet coded symbol we don't have to split, just replace NYT with the new symbol.
3. Go to the symbol's corresponding node (the old NYT node if we split).
4. Consider all nodes with the same weight as the current node, except its parent. If the current node is not the node with highest number we switch places (ie move weight, pointers to children and possible symbol) between the current node and the node with highest number.
5. Increase the weight of the current node by 1.
6. If we are in the root node we are done, otherwise move to the parent of the current node and repeat from 4.

Forgetting factor

If we want the coding to depend more on more recent symbols than on older symbol we can use a forgetting factor.

When the weight of the root node is larger than N we divide the weight in all nodes with K.

If we want to keep the weights as integers we have to divide the weights of all leaf nodes by K (round up) and then add up the weights from the children to the parents, all the way to the root node.

Depending on how we choose N and K we can adjust the speed of adaptation. Large K and small N give fast adaptation and vice versa.

Run-length coding

Sometimes we have sources that produce long partial sequences consisting of the same symbol. It can then be practical to view the sequence as consisting of *runs* instead of symbols. A run is a tuple describing what symbol that is in the run and how long the run is.

For example, the sequence

$$aaaabbbbbbbccbbbbaaaa$$

can be described as

$$(a, 4)(b, 7)(c, 2)(b, 4)(a, 4)$$

Basically we have switched to another alphabet than the original one.

The gain is that it might be easier to find a good code for the new alphabet, and that it's easier to take advantage of the memory of the source.

Note that if the original alphabet is binary, we only have to send the symbol for the first run.

Fax coding

Fax coding is a typical example when run-length coding is used.

There are two digital fax standard: Group 3 (T.4) and group 4 (T.6).

A fax machine scans a page one line at a time (1728 pixels per line for A4 papers). Each pixel is either white or black. Typically we get runs of alternating white and black.

A line can either be coded separately or with the help of the previous line.

Fax coding, cont.

When coding a line separately, run-length coding is used. Since the number of possible run-lengths is big it is impractical to have a Huffman code over all run-lengths. Instead, a run-length r is described as

$$r = 64 \cdot m + t \ , \quad t = 0, \ldots, 63 \text{ and } m = 1, \ldots, 27$$

An extra symbol to code the end of line (EOL) is also introduced, ie to be used when the rest of the line has the same colour.

The first run of each line is assumed to be white.

The alphabet with different m, t and EOL is coded using static tree codes, one each for white and black runs.

This type of coding is called MH (*modified huffman*).

Fax coding, cont.

Two consecutive lines are probably very similar. This can be exploited in the coding.
A few definitions:

- a_0 The last pixel on a line that is know by both sender and receiver, ie current position on the line. When coding starts, this is an imaginary white pixel to the left of the first pixel of the line.
- a_1 The first pixel to the right of a_0 with the opposite colour. Known only by the sender.
- a_2 The first pixel to the right of a_1 with the opposite colour. Known only by the sender.
- b_1 The first pixel to the right of a_0 on the previous line with the opposite colour. Known by both sender and receiver.
- b_2 The first pixel to the right of b_1 that has the opposite colour. Known by both sender and receiver.

Fax coding, cont.

At coding you get three cases.

1. If both b_1 and b_2 are between a_0 and a_1 the codeword 0001 is transmitted. All pixels up to the location under b_2 have the same colour. This point will become our new a_0. New b_1 and b_2 are found.
2. a_1 is located before b_2 and the distance between b_1 and a_1 is no more than 3. The distance $a_1 - b_1$, $\{-3, -2, -1, 0, 1, 2, 3\}$ is coded using the codewords $\{0000010, 000010, 010, 1, 011, 000011, 00000011\}$. a_1 becomes the new a_0.
3. I all other cases 001 is transmitted and the run-lengths from a_0 to a_1 and from a_1 to a_2 are coded using the MH code.

Fax coding, cont.

In group 3 both methods are used. With regular intervalls a line is coded using pure one-dimensionel MH coding, so that any transmission errors will not propagate over the whole image. This coding method is called MR (*modified READ*).

In group 4 only the two-dimensional method is used. This is called MMR (*modified MR*).

Monotonously decreasing distributions

When coding waveform data, such as sound or images, we often have distributions where the alphabet consists of integers $\mathcal{A} = \{0, 1, 2, 3, \ldots\}$ (or $\mathcal{A} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$) and where the probabilities are monotonously decreasing with increasing (absolute) values.

Instead of counting statistics and constructing tree codes we can then often use codes where the codewords can easily be found directly from the symbols and where small values have short codewords and large values have long codewords.

The unary code (the Umbra code)

The codeword for a non-negative integer n consists of n ones followed by a zero.

Symbol	codeword
0	0
1	10
2	110
3	1110
4	11110
$\vdots$	$\vdots$

The unary code achieves the entropy bound for the dyadic distribution $p(i) = 2^{-(i+1)}$

In some applications long sequences of ones are not desirable. Then you can use the reverse definition, where the codeword is n zeros followed by a one.

Golomb codes

$\mathcal{A} = \{0, 1, 2, \ldots\}$

Choose the parameter m. In practice, m is usually chosen to be an integer power of two, but it can be any positive integer. Golomb codes where m is an integer power of two are sometimes referred to as Rice codes.

Represent the integer n with $q = \lfloor \frac{n}{m} \rfloor$ and $r = n - qm$.

Code q with a unary code.

If m is an integer power of two, code r binary with $\log m$ bits.

If m is not an integer power of two:

$0 \leq r < 2^{\lceil \log m \rceil} - m$	Code r binary with $\lfloor \log m \rfloor$ bits
$2^{\lceil \log m \rceil} - m \leq r \leq m - 1$	Code $r + 2^{\lceil \log m \rceil} - m$ binary with $\lceil \log m \rceil$ bits

(This type of code is called a *truncated binary code*.)

Examples of Golomb codes

Symbol	$m = 1$	$m = 2$	$m = 3$	$m = 4$
0	0	0 0	0 0	0 00
1	10	0 1	0 10	0 01
2	110	10 0	0 11	0 10
3	1110	10 1	10 0	0 11
4	11110	110 0	10 10	10 00
5	111110	110 1	10 11	10 01
6	1111110	1110 0	110 0	10 10
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Golomb codes are optimal for distributions of the type

$$p(i) = q^i \cdot (1 - q) \quad ; \quad 0 < q < 1$$

if we choose $m = \lceil -\frac{1}{\log q} \rceil$
Golomb codes are for instance used in the image coding standard JPEG-LS and in the video coding standard H.264.

Exp-Golomb codes

$\mathcal{A} = \{0, 1, 2, \ldots\}$

Choose the parameter $m = 2^k$, k non-negative integer.

Calculate $s = \lfloor \log_2(n + m) \rfloor$.

Code $s - k$ with a unary code.

Code $n - 2^s + m$ binary with s bits.

Examples of Exp-Golomb codes

Symbol	$k = 0$	$k = 1$	$k = 2$
0	0	0 0	0 00
1	10 0	0 1	0 01
2	10 1	10 00	0 10
3	110 00	10 01	0 11
4	110 01	10 10	10 000
5	110 10	10 11	10 001
6	110 11	110 000	10 010
7	1110 000	110 001	10 011
8	1110 001	110 010	10 100
9	1110 010	110 011	10 101
10	1110 011	110 100	10 110
⋮	⋮	⋮	⋮

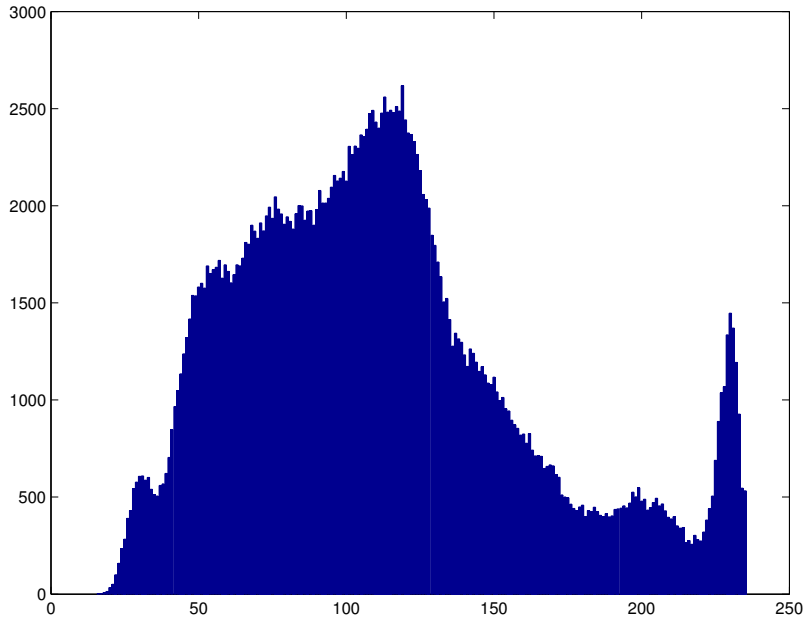
Exp-Golomb codes are for instance used in i H.264.

Test image Goldhill

512×512 pixels, 8 bits/pixel

Simple Huffman coding

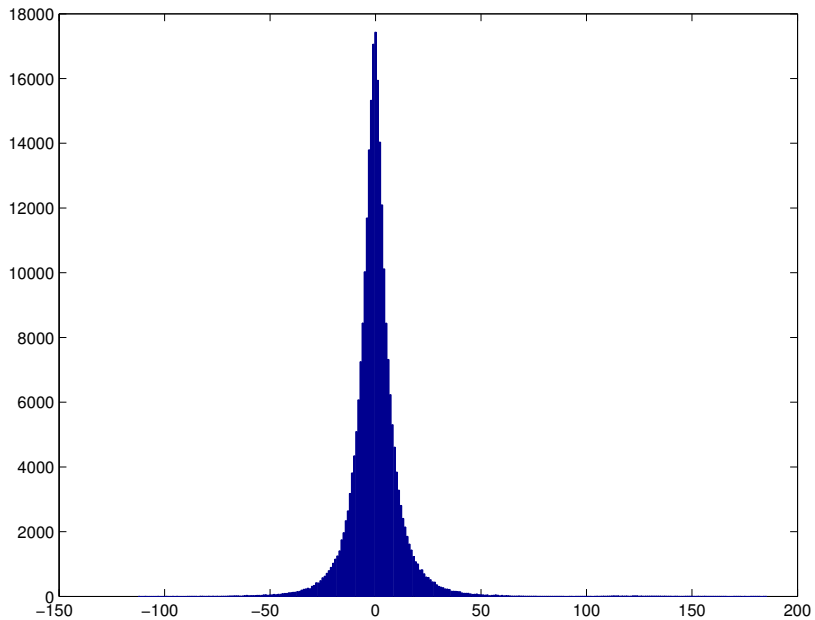
Histogram for Goldhill:



Huffman coding gives an average data rate of 7.50 bits/pixel
The longest codeword is 16 bits, the shortest codeword is 7 bits.
We haven't used any of the dependence between pixels.

Huffman coding of differences

Instead of coding the pixels directly, we code the difference in pixel value between a pixel and the pixel above it. Imaginary pixels outside of the image are assumed to be medium gray, ie have the value 128. The smallest difference is -112, The largest difference is 107.
Histogram of differences:



Huffman coding of the differences gives an average data rate of 5.34 bits/pixel.
The longest codeword is 18 bits, the shortest codeword is 4 bits.

Golomb coding I

We must first modify the values so that we only have non-negative values. This can for instance be done using the mapping

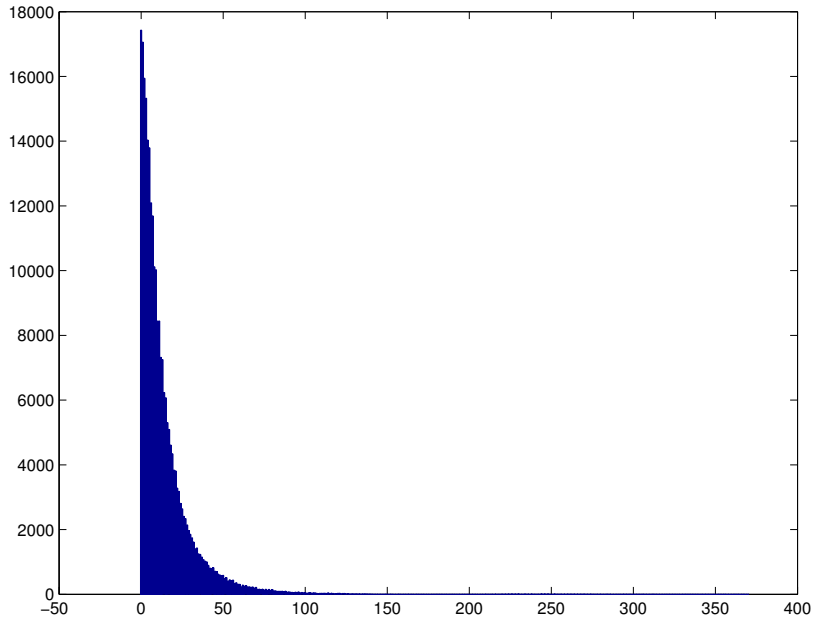
$$F(x) = \begin{cases} 2x & ; \ x \geq 0 \\ -2x - 1 & ; \ x < 0 \end{cases}$$

ie the negative numbers are mapped to odd positive numbers and the positive numbers are mapped to even positive numbers.

$$F^{-1}(x) = \begin{cases} \frac{x}{2} & ; \ x \text{ even} \\ -\frac{x+1}{2} & ; \ x \text{ odd} \end{cases}$$

Golomb coding I, cont.

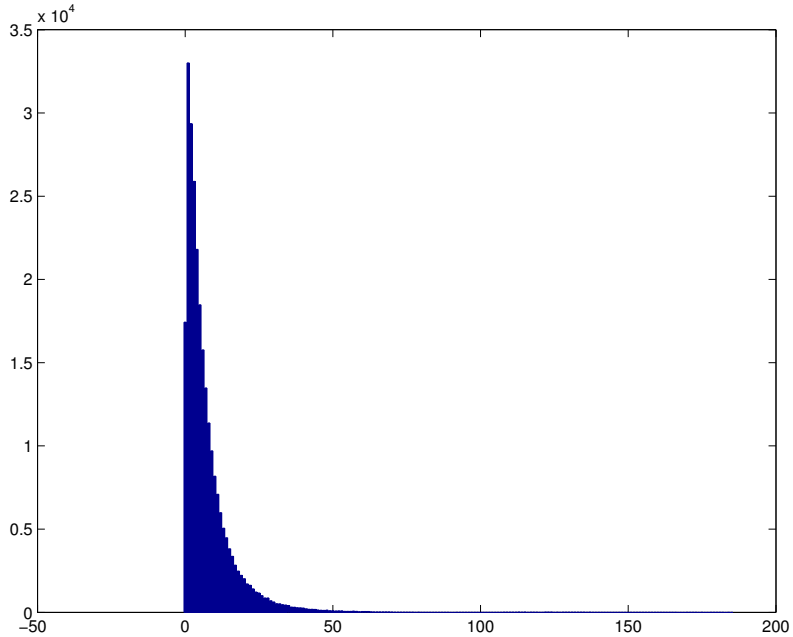
Histogram for modified differences



The best Golomb code is the one with parameter $m = 10$, which gives an average data rate of 5.37 bits/pixel.

Golomb coding II

Alternatively we can code the absolute value of the differences with a Golomb code and then send an extra sign bit for each non-zero value.
Histogram for the absolute value of differences



The best Golomb code is the one with parameter $m = 5$, which gives an average data rate of 5.40 bits/pixel.

Lossless JPEG

JPEG is normally an image coding method that gives distortion, but there is also a lossless mode in the standard.

The pixels are coded row-wise from the top down.

The pixel I_{ij} on position (i, j) is predicted from neighbouring pixels. There are 7 predictors to choose from:

1. $\hat{I}_{ij} = I_{i-1,j}$
2. $\hat{I}_{ij} = I_{i,j-1}$
3. $\hat{I}_{ij} = I_{i-1,j-1}$
4. $\hat{I}_{ij} = I_{i,j-1} + I_{i-1,j} - I_{i-1,j-1}$
5. $\hat{I}_{ij} = I_{i,j-1} + \lfloor (I_{i-1,j} - I_{i-1,j-1})/2 \rfloor$
6. $\hat{I}_{ij} = I_{i-1,j} + \lfloor (I_{i,j-1} - I_{i-1,j-1})/2 \rfloor$
7. $\hat{I}_{ij} = \lfloor (I_{i,j-1} + I_{i-1,j})/2 \rfloor$

Lossless JPEG, cont.

The difference $d_{ij} = I_{ij} - \hat{I}_{ij}$ is coded either by an adaptive arithmetic coder, or using a Huffman code.
Huffman coding is not performed directly on the differences. Instead cathegories

$$k_{ij} = \lceil \log(|d_{ij}| + 1) \rceil$$

are formed. Statistics for the cathegories are calculated and a Huffman tree is constructed.
The codeword for a difference d_{ij} consists of the Huffman codeword for k_{ij} plus k_{ij} extra bits used to exactly specify d_{ij}.

k_{ij}	d_{ij}	extra bits
0	0	–
1	$-1, 1$	$0, 1$
2	$-3, -2, 2, 3$	$00, 01, 10, 11$
3	$-7, \ldots, -4, 4, \ldots, 7$	$000, \ldots, 011, 100, \ldots, 111$
$\vdots$	$\vdots$	$\vdots$

Lossless JPEG, cont.

Coding Goldhill using lossless JPEG:

Predictor 1	5.39 bits/pixel
Predictor 2	5.42 bits/pixel
Predictor 3	5.80 bits/pixel
Predictor 4	5.27 bits/pixel
Predictor 5	5.16 bits/pixel
Predictor 6	5.15 bits/pixel
Predictor 7	5.13 bits/pixel

For different images different predictors will work best. The standard supports coding different parts of an image with different predictors.

JPEG-LS

Standard for lossles and near lossless coding of images. Near lossless means that we allow the pixel values of the decoded image to be a little different from the original pixels.

The pixels are coded row-wise from the top down.

When pixel (i, j) is to be coded you first look at the surrounding pixels in position $(i, j-1)$, $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$. A *context* is formed by first calculating the gradients

$$\begin{aligned} D_1 &= I_{i-1,j+1} - I_{i-1,j} \\ D_2 &= I_{i-1,j} - I_{i-1,j-1} \\ D_3 &= I_{i-1,j-1} - I_{i,j-1} \end{aligned}$$

JPEG-LS, cont.

The gradients D_k are quantized to three integers Q_k such that $-4 \leq Q_k \leq 4$. The quantizer bounds can be chosen by the coder. Each Q_k takes 9 possible values, which means that we have 729 possible combinations. A pair of combinations with inverted signs counts as the same context which finally gives us 365 different contexts.

A prediction of I_{ij} is done according to:
If $I_{i-1,j-1} \geq \max(I_{i,j-1}, I_{i-1,j}) \Rightarrow \hat{I}_{ij} = \max(I_{i,j-1}, I_{i-1,j})$
if $I_{i-1,j-1} \leq \min(I_{i,j-1}, I_{i-1,j}) \Rightarrow \hat{I}_{ij} = \min(I_{i,j-1}, I_{i-1,j})$
Otherwise: $\hat{I}_{ij} = I_{i,j-1} + I_{i-1,j} - I_{i-1,j-1}$

For each context q we keep track if the prediction has a systematic error, if that is the case the prediction is adjusted a little in the correct direction.

JPEG-LS, cont.

The difference between the real pixel value and the predicted value $d_{ij} = I_{ij} - \hat{I}_{ij}$ is coded using a Golomb code with parameter $m = 2^{k_q}$. For each context q we keep track of the best Golomb code, and each k_q is constantly adjusted during the coding process.

The coder also detects if we get long runs of the same value on a row. In that case the coder switches to coding run-lengths instead.

If we code Goldhill using JPEG-LS we get an average data rate of 4.71 bits/pixel.