

# Go deh!

Mainly Tech projects on Python and Electronic Design Automation.

Saturday, March 28, 2009

## Huffman Encoding in Python

Huffman encoding came up on [Rosetta Code](#).

Huffman encoding is a way to assign binary codes to symbols that reduces the overall number of bits used to encode a typical string of those symbols.

For example, if you use letters as symbols and have details of the frequency of occurrence of those letters in typical strings, then you could just encode each letter with a fixed number of bits, such as in ASCII codes. You can do better than this by encoding more frequently occurring letters such as e and a, with smaller bit strings; and less frequently occurring letters such as q and x with longer bit strings.

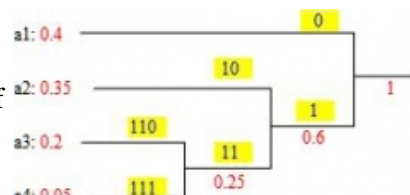
Any string of letters will be encoded as a string of bits that are no-longer of the same length per letter. To successfully decode such a string, the smaller codes assigned to letters such as 'e' cannot occur as a prefix in the larger codes such as that for 'x'.

If you were to assign a code 01 for 'e' and code 011 for 'x', then if the bits to decode started as 011... then you would not know if you should decode an 'e' or an 'x'.

The Huffman coding scheme takes each symbol and its weight (or frequency of occurrence), and generates proper encodings for each symbol taking account of the weights of each symbol, so that higher weighted symbols have less bits in their encoding. (See the [WP article](#) for more information).

A Huffman encoding can be computed by first creating a tree of nodes:

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



Traverse the constructed binary tree from root to leaves assigning and accumulating a '0' for one branch and a '1' for the other at each node. The accumulated zeroes and ones at each leaf constitute a Huffman encoding for those symbols and weights.

## In Python

I originally gave an an example that matched a definition that was later found to be insufficient, so substituted my own definition above.. My [first Python solution](#) on RC to the wrong definition, did have the advantage, (as I saw it), of not having to traverse a tree. My 'true' Huffman code creator assembles each symbol and its weight into the following structure initially (the **leaf structure**):

```
[ weight, [ symbol, []]]
```

The weight applies to every (in this case only one), of the [symbol, []] pairs after it in the same list.

The empty list is used to accumulate the Huffman code for the symbol as we manipulate the heap, without having to walk a constructed tree structure.

There are two types of input to the program that I am running examples with:

1. A string of space separated symbol, weight pairs, as used in small examples.
2. A sample of text for which letters and letter frequencies are extracted.

The if statement at line **23** allows me to switch between the two types of input whilst exploring the algorithm.

The tutor argument to the encode function shows what is happening in the loop around the heap pops

```

1
2 from heapq import heappush, heappop, heapify
3
4 def codecreate(symbol2weights, tutor= False):
5     ''' Huffman encode the given dict mapping symbols to weights '''
6     heap = [ [float(wt), [sym, []]] for sym, wt in symbol2weights.iteritems() ]
7     heapify(heap)
8     if tutor: print "ENCODING:", sorted(symbol2weights.iteritems())
9     while len(heap) >1:
10         lo = heappop(heap)
11         hi = heappop(heap)
12         if tutor: print "  COMBINING:", lo, '\n          AND:', hi
13         for i in lo[1:]: i[1].insert(0, '0')
14         for i in hi[1:]: i[1].insert(0, '1')
15         lohi = [ lo[0] + hi[0] ] + lo[1:] + hi[1:]
16         if tutor: print "  PRODUCING:", lohi, '\n'
17         heappush(heap, lohi)
18     codes = heappop(heap)[1:]
19     for i in codes: i[1] = ''.join(i[1])
20     return sorted(codes, key=lambda x: (len(x[-1]), x))
21
22 # Input types
23 if 1:
24     readin = "B 25   C 2.5 D 12.5 A 5 \n"
25     #readin = "a .1 b .15 c .3 d .16 e .29" # Wikipedia sample
26     #readin = "a1 .4 a2 .35 a3 .2 a4 .05" # Wikipedia sample
27     #readin = "A 50 B 25 C 12.5 D 12.5" # RC example
28
29     cleaned = readin.strip().split()
30     symbol2weights = dict((symbol, wt)
31                           for symbol, wt in zip(cleaned[0::2], cleaned[1::2]) )
32 else:
33     astring = "this is an example for huffman encoding"
34     symbol2weights = dict((ch, astring.count(ch)) for ch in set(astring)) # for astring
35
36 huff = codecreate(symbol2weights, True)
37 print "\nSYMBOL\tWEIGHT\tHUFFMAN CODE"
38 for h in huff:
39     print "%s\t%s\t%s" % (h[0], symbol2weights[h[0]], h[1])

```

A run, with the tutor enabled gives the following output:

```

ENCODING: [('A', '5'), ('B', '25'), ('C', '2.5'), ('D', '12.5')]
  COMBINING: [2.5, ['C', []]]
          AND: [5.0, ['A', []]]
  PRODUCING: [7.5, ['C', ['0']], ['A', ['1']]]

```

```

COMBINING: [7.5, ['C', ['0']], ['A', ['1']]]
AND: [12.5, ['D', []]]
PRODUCING: [20.0, ['C', ['0', '0']], ['A', ['0', '1']], ['D', ['1']]]

COMBINING: [20.0, ['C', ['0', '0']], ['A', ['0', '1']], ['D', ['1']]]
AND: [25.0, ['B', []]]
PRODUCING: [45.0, ['C', ['0', '0', '0']], ['A', ['0', '0', '1']], ['D', ['0', '1']], ['B', ['1']]]

```

| SYMBOL | WEIGHT | HUFFMAN CODE |
|--------|--------|--------------|
| B      | 25     | 1            |
| D      | 12.5   | 01           |
| A      | 5      | 001          |
| C      | 2.5    | 000          |

## Encode/Decode Round-tripping

I realised that I could use a method similar to how I accumulate the codes in the heap loop, to generate a single function that can recognise a single symbol from the beginning of the encoded symbols. By using the function in a loop, I could regenerate the symbol list.

In the codecreate function, the **leaf structure** is modified:

```
[weight, [ [symbol, []] ], repr(sym)]
```

There is an extra level of list around the [symbol, code accumulation list pair] as well as a new item: 'repr(sym)' it is the third item in the outer list and will always be the function to generate the item as accumulated so far.. This function starts off by returning just the symbol and an outer if/then/else expression is added as we go around the heap loop (see line **43**)

After the outer expression is accumulated, it is turned into a lambda expression, the string eval'd, and assigned to a global variable (see line **47**)

I use function probchoice (from earlier RC work), to create an arbitrary sequence of symbols to in the given weighting then encode and decode it as well as giving some stats on space saving.

```

1
2 from heapq import heappush, heappop, heapify
3 import random, bisect
4
5
6 # Helper routine for generating test sequences
7 def probchoice(items, probs):
8     '''
9     Splits the interval 0.0-1.0 in proportion to probs
10    then finds where each random.random() choice lies
11    (This routine, probchoice, was released under the
12    GNU Free Documentation License 1.2)
13    '''
14
15    probb_accumulator = 0
16    accumulator = []
17    for p in probs:
18        probb_accumulator += p
19        accumulator.append(probb_accumulator)
20
21    while True:
22        r = random.random()
23        yield items[bisect.bisect(accumulator, r)]
24
25
26 # placeholder
27 decode = lambda : None
28

```

```

29 def codecreate(symbol2weights, tutor= False):
30     ''' Huffman encode the given dict mapping symbols to weights '''
31     global decode
32
33     heap = [ [float(wt), [[sym, []]], repr(sym)] for sym, wt in symbol2weights.iteritems()]
34     heapify(heap)
35     if tutor: print "ENCODING:", sorted(symbol2weights.iteritems())
36     while len(heap) > 1:
37         lo = heappop(heap)
38         hi = heappop(heap)
39         if tutor: print "  COMBINING:", lo, '\n          AND:', hi
40         for i in lo[1]: i[1].insert(0, '0')
41         for i in hi[1]: i[1].insert(0, '1')
42         lohi = [ lo[0] + hi[0] ] + [lo[1] + hi[1]]
43         lohi.append('(%s if nextbit() else %s)' % (hi[2], lo[2]))
44         if tutor: print "  PRODUCING:", lohi, '\n'
45         heappush(heap, lohi)
46     wt, codes, decoder = heappop(heap)
47     decode = eval('lambda : ' + decoder, globals())
48     decode.__doc__ = decoder
49     for i in codes: i[1] = ''.join(i[1])
50     #for i in codes: i[:2] = i[:2]
51     return sorted(codes, key=lambda x: (len(x[-1]), x))
52
53 # Input types
54 if 1:
55     tutor = True
56     sequencecount = 50
57     readin = "B 25 C 2.5 D 12.5 A 5 \n"
58     #readin = "a .1 b .15 c .3 d .16 e .29" # Wikipedia sample
59     #readin = "a1 .4 a2 .35 a3 .2 a4 .05" # Wikipedia sample
60     #readin = "A 50 B 25 C 12.5 D 12.5" # RC example
61
62     cleaned = readin.strip().split()
63     symbol2weights = dict((symbol, wt)
64                           for symbol, wt in zip(cleaned[0::2], cleaned[1::2]) )
65 else:
66     tutor = False
67     sequencecount = 500
68     astring = "this is an example for huffman encoding"
69     symbol2weights = dict((ch, astring.count(ch)) for ch in set(astring)) # for astring
70
71 huff = codecreate(symbol2weights, tutor= tutor)
72 print "\nSYMBOL\tWEIGHT\tHUFFMAN CODE"
73 for h in huff:
74     print "%s\t%s\t%s" % (h[0], symbol2weights[h[0]], h[1])
75
76 ##
77 ## encode-decode check
78 ##
79 symbol2code = dict(huff)
80 symbols, weights = zip(*symbol2weights.iteritems())
81 # normalize weights
82 weights = [float(wt) for wt in weights]
83 tot = sum(weights)
84 weights = [wt/tot for wt in weights]
85 # Generate a sequence
86 nxt = probchoice(symbols, weights).next
87 symbolsequence = [nxt() for i in range(sequencecount)]
88 # encode it
89 bitsequence = ''.join(symbol2code[sym] for sym in symbolsequence)
90
91 sslen, slen, blen = len(symbolsequence), len(symbols), len(bitsequence)
92 countlen = len(bin(slen-1)[2:])
93 print '''
94
95
96 ROUND-TRIPPING
97 =====
98 I have generated a random sequence of %i symbols to the given weights.
99 If I use a binary count to encode each of the %i symbols I would need
100 %i * %i = %i bits to encode the sequence.
101 Using the Huffman code, I need only %i bits.
102 ''' % (slen, slen, sslen, countlen, sslen * countlen, blen )
103
104 ## decoding

```

```

105 nextbit = (bit=='1' for bit in bitsequence).next
106
107 decoded = []
108 try:
109     while 1:
110         decoded.append(decode())
111 except StopIteration:
112     pass
113
114 print "Comparing the decoded sequence with the original I get:", decoded == symbolsequence

```

This short run is in tutor mode, so you can track the accumulation of the decode function:

```

ENCODING: [('A', '5'), ('B', '25'), ('C', '2.5'), ('D', '12.5')]
COMBINING: [2.5, [['C', []], "'C'"]]
AND: [5.0, [['A', []], "'A'"]]
PRODUCING: [7.5, [['C', ['0']], ['A', ['1']]], "('A' if nextbit() else 'C')"]

COMBINING: [7.5, [['C', ['0']], ['A', ['1']]], "('A' if nextbit() else 'C')"]
AND: [12.5, [['D', []], "'D'"]]
PRODUCING: [20.0, [['C', ['0', '0']], ['A', ['0', '1']], ['D', ['1']]], "('D'
if nextbit() else ('A' if nextbit() else 'C'))"]

COMBINING: [20.0, [['C', ['0', '0']], ['A', ['0', '1']], ['D', ['1']]], "('D'
if nextbit() else ('A' if nextbit() else 'C'))"]
AND: [25.0, [['B', []], "'B'"]]
PRODUCING: [45.0, [['C', ['0', '0', '0']], ['A', ['0', '0', '1']], ['D',
['0', '1']], ['B', ['1']]], "('B' if nextbit() else ('D' if nextbit() else ('A'
if nextbit() else 'C'))")"]

```

| SYMBOL | WEIGHT | HUFFMAN CODE |
|--------|--------|--------------|
| B      | 25     | 1            |
| D      | 12.5   | 01           |
| A      | 5      | 001          |
| C      | 2.5    | 000          |

#### ROUND-TRIPPING

=====

I have generated a random sequence of 50 symbols to the given weights.  
 If I use a binary count to encode each of the 4 symbols I would need  
 $50 * 2 = 100$  bits to encode the sequence.  
 Using the Huffman code, I need only **90** bits.

Comparing the decoded sequence with the original I get: **True**

And if I change line **54** to be False, I get the following:

| SYMBOL | WEIGHT | HUFFMAN CODE |
|--------|--------|--------------|
| 6      | 101    |              |
| n      | 4      | 010          |
| a      | 3      | 1001         |
| e      | 3      | 1100         |
| f      | 3      | 1101         |
| h      | 2      | 0001         |
| i      | 3      | 1110         |
| m      | 2      | 0010         |
| o      | 2      | 0011         |
| s      | 2      | 0111         |
| g      | 1      | 00000        |
| l      | 1      | 00001        |

|   |   |        |
|---|---|--------|
| p | 1 | 01100  |
| r | 1 | 01101  |
| t | 1 | 10000  |
| u | 1 | 10001  |
| x | 1 | 11110  |
| c | 1 | 111110 |
| d | 1 | 111111 |

#### ROUND-TRIPPING

=====

I have generated a random sequence of 500 symbols to the given weights.  
 If I use a binary count to encode each of the 19 symbols I would need  
 $500 * 5 = 2500$  bits to encode the sequence.  
 Using the Huffman code, I need only 2012 bits.

Comparing the decoded sequence with the original I get: True

Note: The purpose of the program is to teach me more about Huffman coding and is not an exercise in speed!

I am the author of all the Python on this page. The diagram is from Wikipedia. Please refrain from passing-off my code as your own (that one is mainly for students).

Posted by Paddy3118 at 12:11 am 

Reactions:      interesting (0)      reactionary (0)  
                  waste of space (0)      disinterested (0)

#### 2 comments:

**Anonymous Sat Mar 28, 01:12:00 am**

**The bitarray module**

**<http://pypi.python.org/pypi/bitarray/>**

**efficiently represents an array of booleans as bits. It also provides example code for performing Huffman encoding and decoding.**

**Reply**




**Paddy3118 Sat Mar 28, 06:20:00 am**

**It takes doing my own program for me to truly appreciate what is being given in something like the bitarray module.**

**- Paddy.**

**Reply**

Enter your comment...

 Comment as: Ashok B (Goog ▼) 

Sign out

Publish

Preview

☐ Notify me

[Newer Post](#)[Home](#)[Older Post](#)[Subscribe to: Post Comments \(Atom\)](#)

---

## About Me

Paddy3118

[View my complete profile](#)

---

## Followers

Followers (16)

---

## Subscribe Now: google



---

## Go deh too!

---

[whos.amung.us](#)

---

## Blog Archive

- 2019 (8)
- 2018 (10)
- 2017 (10)
- 2016 (3)
- 2015 (8)
- 2014 (18)
- 2013 (14)
- 2012 (12)
- 2011 (7)
- 2010 (12)

- ▼ 2009 (42)
    - ▶ December (1)
    - ▶ November (4)
    - ▶ October (3)
    - ▶ September (3)
    - ▶ August (4)
    - ▶ July (1)
    - ▶ June (4)
    - ▶ May (6)
    - ▶ April (6)
    - ▼ March (3)
      - Huffman Encoding in Python
      - Batch Process Runner in bash shell - Forgotten Enh...
      - Psion Netbook
    - ▶ February (5)
    - ▶ January (2)
  - ▶ 2008 (28)
  - ▶ 2007 (23)
  - ▶ 2006 (7)
  - ▶ 2005 (1)
-