

Hand in problem 2 in Information Theory

Fredrick Nilsson

February 2, 2024

Report

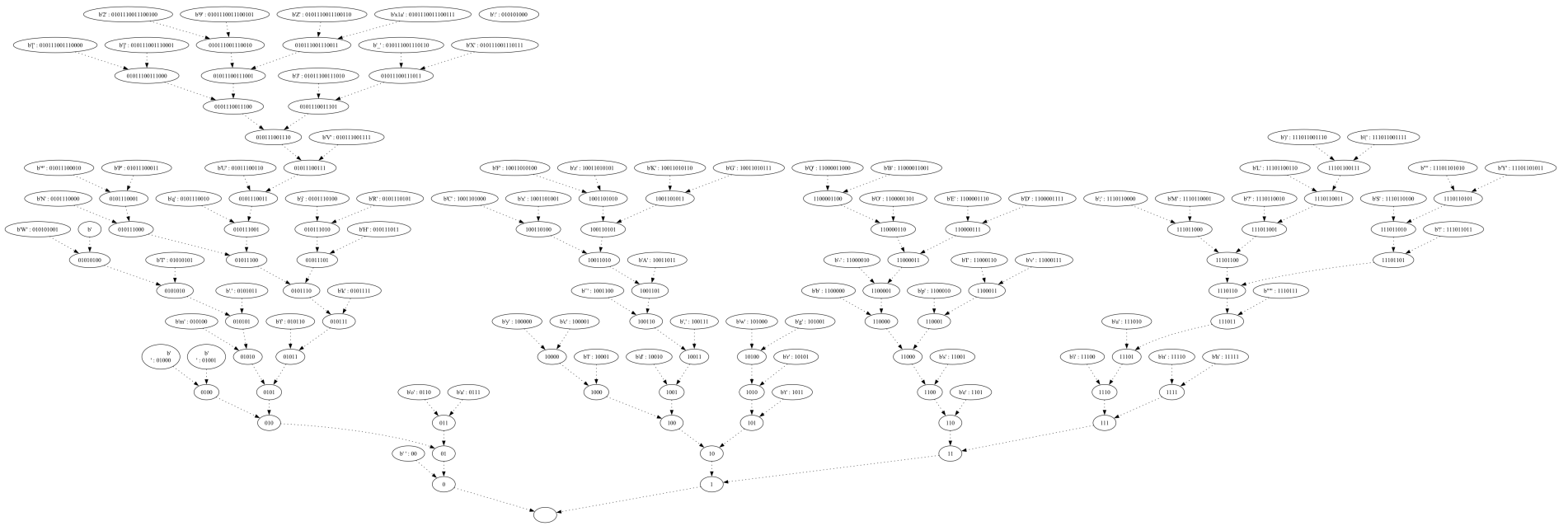


Figure 1: The huffman tree

Source code

```
from collections import Counter
from bitarray import bitarray
import numpy as np
from entropy import InfoTheory

INPUT_PATH = 'Alice29.txt'
COMPRESSED_PATH = 'compressed'
DECOMPRESSED_PATH = 'decompressed.txt'
CODE_TABLE_PATH = 'code_table.csv'
OCCURANCE_PATH = 'occurrence.csv'

class Node(object):
    def __init__(self, zero=None, one=None):
        self.zero = zero
        self.one = one

    def children(self):
        return (self.zero, self.one)

    def __str__(self):
        return '%s_%s' % (self.zero, self.one)

def read_data(path):
    # Output:
    # data: file as array of raw bytes
    if isinstance(path, str):
        data = []
        with open(path, 'rb') as f:
            for byte in f.read():
                data.append(byte.to_bytes(1, byteorder='big'))
        return data
    else:
        raise TypeError('Input must be a string')

def print_to_file(data, path):
    # Input:
    # data: bytes to print
    # path: string with the path to the file
    # print(type(data))
    # print(type(path))
    if isinstance(data, bytes) and isinstance(path, str):
        with open(path, 'wb') as f:
            f.write(data)
```

```

else:
    raise TypeError('Input must be bytes and a string')

def huffman_tree(occurrences):
    # Input:
    # occurrences: dictionary with the number of occurrences of each
    #               ↪ symbol
    # Output:
    # root: the root of the Huffman tree
    if isinstance(occurrences, dict):
        probabilities = sorted([(key, value)
                                for (key, value) in
                                ↪ occurrences.items()], key=lambda x:
                                ↪ x[1])
        while len(probabilities) > 1:
            left = probabilities.pop(0)
            right = probabilities.pop(0)
            node = Node(left[0], right[0])
            probabilities.append((node, left[1]+right[1]))
            probabilities = sorted(probabilities, key=lambda x: x[1])
        return probabilities.pop(0)[0]
    else:
        raise TypeError('Input must be a dictionary')

def codebook(node, code=''):
    # Input:
    # root: the root of the Huffman tree
    # code: the code for the current node
    # Output:
    # codebook: a dictionary with the Huffman code for each symbol
    if isinstance(node, Node):
        book = {}
        (zero, one) = node.children()
        book.update(codebook(zero, code+'0'))
        book.update(codebook(one, code+'1'))
        return book
    elif isinstance(node, bytes):
        return {node: code}
    elif node is None:
        return {}
    else:
        print(type(node))
        raise TypeError('Input must be a Node')

```

```

def compress(data, codebook):
    # Input:
    #   data: list of bytes with the data to compress
    #   codebook: a dictionary with the Huffman code for each byte
    # Output:
    #   compressed_data: bytes with the compressed data
    if isinstance(data, list) and isinstance(codebook, dict):
        s = ''
        for symbol in data:
            s += (codebook[symbol])

        # If the compressed data is not a multiple of 8, it will add
        #   ↪ dummy bits to make it a multiple of 8
        # Select dummy bits so that they match no code in the codebook
        # If it doesn't find any unused bit combination in  $2^l$ 
        #   ↪ attempts, it will simply fill it with zeroes which might
        #   ↪ add additional characters on decompression
        if len(s) % 8 != 0:
            l = (8 - len(s) % 8)
            inv_codebook = {value: key for key, value in
                            ↪ codebook.items()}
            for _ in range(pow(2, l)):
                dummy_bits = ''.join(str(element) for element in [
                                    np.random.choice([0, 1]) for i in
                                    ↪ range(l)])
                if any([inv_codebook.get(dummy_bits[0:i]) is not None
                        ↪ for i in range(0, len(dummy_bits))]):
                    continue
                else:
                    s += dummy_bits
                    break
            compressed_data = bytearray(s).tobytes()
        return compressed_data
    else:
        raise TypeError('Input must be a list and a dictionary')

def decompress(compressed_data, codebook):
    # Input:
    #   compressed_data: list of bytes with the data to decompress
    #   codebook: a dictionary with the Huffman code for each byte
    # Output:
    #   data: bytes with the decompressed data
    if isinstance(compressed_data, list) and isinstance(codebook,
        ↪ dict):
        data = bytes()
        codebook = {value: key for key, value in codebook.items()}

```

```

compressed_data = b''.join(compressed_data)
s = bytearray()
s.frombytes(compressed_data)
s = s.to01()
i = 0
while i < len(s):
    j = i+1
    while s[i:j] not in codebook.keys() and j < len(s):
        j += 1
    if j < len(s)-1:
        data += (codebook[s[i:j]])
    elif j == len(s) and s[i:j] in codebook.keys():
        data += (codebook[s[i:j]])
    i = j
return data
else:
    raise TypeError('Input must be a list and a dictionary')

def print_codebook(codebook, occurrences):
    # Input:
    # codebook: a dictionary with the Huffman code for each byte
    # Output:
    # code_table.csv: a csv file with the codebook
    if isinstance(codebook, dict):
        probabilities = sorted([(key, value)
                                for (key, value) in occurrences.items(), key=lambda x:
                                ↪ x[1], reverse=True])
        with open(CODE_TABLE_PATH, 'w') as f:
            f.write("symbol,probability,code\n")
            for [key, value] in probabilities:
                f.write("%s,%s,%s\n" % (key.hex(), str(value),
                ↪ codebook[key]))
    else:
        raise TypeError('Input must be a dictionary')

def print_occurrence(occurrences):
    # Input:
    # codebook: a dictionary with the Huffman code for each byte
    # Output:
    # code_table.csv: a csv file with the codebook
    if isinstance(codebook, dict):
        probabilities = sorted([(key, value)
                                for (key, value) in occurrences.items(), key=lambda x:
                                ↪ x[1], reverse=True])
        with open(OCCURANCE_PATH, 'w') as f:
            f.write("symbol,probability\n")
            for [key, value] in probabilities:

```

```

        f.write("%s,%s\n" % (key.hex(), str(value)))
    else:
        raise TypeError('Input must be a dictionary')

if __name__ == '__main__':

    # COMPRESS
    data = read_data(INPUT_PATH)
    occurrences = Counter(data)
    root = huffman_tree(occurrences)
    codebook = codebook(root)
    compressed_data = compress(data, codebook)

    entropy = InfoTheory.Entropy(InfoTheory, np.array([value/len(data)
        ↪ for _, value in occurrences.items()]))
    entropy_compressed =
        ↪ InfoTheory.Entropy(InfoTheory, np.array([value/len(data) for _,
        ↪ value in Counter(compressed_data).items()]))
    print("Entropy of original text: " + entropy)
    print("Entropy of compressed text: " + entropy_compressed)

    average_length = sum([len(codebook[key])*value for key, value in
        ↪ occurrences.items()])/len(data)
    print("Average character length in compressed text: " +
        ↪ average_length)

    print_to_file(compressed_data, COMPRESSED_PATH)
    print_codebook(codebook, occurrences)

    # DECOMPRESS
    compressed = read_data(COMPRESSED_PATH)
    comp_count = Counter(compressed)
    print_occurance(comp_count)
    decompressed = decompress(compressed, codebook)
    print_to_file(decompressed, DECOMPRESSED_PATH)

```


Codebook

| Ascii | Count | Code | | | |
|-------|-------|------------|----|-----|------------------|
| 20 | 28900 | 00 | 4f | 176 | 1100001101 |
| 65 | 13381 | 1101 | 43 | 144 | 1001101000 |
| 74 | 10212 | 1011 | 78 | 144 | 1001101001 |
| 61 | 8149 | 0111 | 52 | 140 | 0101110101 |
| 6f | 7965 | 0110 | 6a | 138 | 0101110100 |
| 68 | 7088 | 11111 | 71 | 125 | 0101110010 |
| 6e | 6893 | 11110 | 4e | 120 | 0101110000 |
| 69 | 6778 | 11100 | 59 | 114 | 11101101011 |
| 73 | 6277 | 11001 | 22 | 113 | 11101101010 |
| 72 | 5293 | 10101 | 4c | 98 | 11101100110 |
| 64 | 4739 | 10010 | 42 | 91 | 11000011001 |
| 6c | 4615 | 10001 | 51 | 84 | 11000011000 |
| 0d | 3608 | 01000 | 4b | 82 | 10011010110 |
| 0a | 3608 | 01001 | 47 | 82 | 10011010111 |
| 75 | 3402 | 111010 | 7a | 77 | 10011010101 |
| 67 | 2446 | 101001 | 46 | 74 | 10011010100 |
| 77 | 2437 | 101000 | 55 | 66 | 01011100110 |
| 2c | 2418 | 100111 | 50 | 64 | 01011100011 |
| 63 | 2253 | 100001 | 2a | 60 | 01011100010 |
| 79 | 2150 | 100000 | 28 | 56 | 111011001111 |
| 66 | 1926 | 010110 | 29 | 55 | 111011001110 |
| 6d | 1907 | 010100 | 56 | 42 | 010111001111 |
| 27 | 1761 | 1110111 | 4a | 8 | 01011100111010 |
| 70 | 1458 | 1100010 | 5f | 4 | 010111001110110 |
| 62 | 1383 | 1100000 | 58 | 4 | 010111001110111 |
| 60 | 1108 | 1001100 | 5b | 2 | 010111001110000 |
| 6b | 1076 | 0101111 | 5d | 2 | 010111001110001 |
| 2e | 977 | 0101011 | 32 | 1 | 0101110011100100 |
| 76 | 803 | 11000111 | 39 | 1 | 0101110011100101 |
| 49 | 733 | 11000110 | 5a | 1 | 0101110011100110 |
| 2d | 669 | 11000010 | 1a | 1 | 0101110011100111 |
| 41 | 638 | 10011011 | | | |
| 54 | 472 | 01010101 | | | |
| 21 | 449 | 111011011 | | | |
| 48 | 284 | 010111011 | | | |
| 57 | 237 | 010101001 | | | |
| 3a | 233 | 010101000 | | | |
| 53 | 218 | 1110110100 | | | |
| 3f | 202 | 1110110010 | | | |
| 4d | 200 | 1110110001 | | | |
| 3b | 194 | 1110110000 | | | |
| 44 | 192 | 1100001111 | | | |
| 45 | 188 | 1100001110 | | | |