Министр науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский университет ИТМО»

Факультет информационных технологий и программирования

Лабораторная работа № 2

Эффективное кодирование

Выполнил студент группы № M3101
Михеев Артем Романович
Подпись:

Проверил:
Страдина Марина Владимировна

Санкт-Петербург
2020

Оглавление

Исходное изображение

- размер 128x128
- цвет 8-битный (256 градаций) серый

Полученное из него сообщение (63-я строка):

- Изначальное = [61, 99, 125, 128, 99, 65, 67, 47, 38, 44, 54, 41, 36, 39, 40, 46, 47, 45, 63, 83, 90, 80, 85, 88, 89, 95, 102, 110, 88, 95, 99, 75, 51, 53, 46, 47, 37, 39, 41, 37, 37, 34, 34, 39, 35, 39, 46, 63, 69, 63, 57, 48, 28, 19, 19, 21, 44, 117, 134, 137, 151, 156, 160, 162, 150, 131, 131, 138, 133, 133, 151, 165, 169, 172, 173, 177, 176, 174, 174, 178, 177, 169, 169, 162, 156, 148, 151, 155, 156, 154, 156, 157, 156, 162, 164, 160, 162, 162, 150, 83, 26, 17, 12, 13, 11, 14, 15, 16, 18, 19, 21, 21, 23, 21, 18, 18, 18, 15, 19, 22, 23, 32, 36, 35, 30, 24, 20, 18]
- После квантования = [60, 100, 120, 120, 100, 60, 60, 40, 40, 40, 60, 40, 40, 40, 40, 40, 40, 40, 60, 80, 80, 80, 80, 80, 80, 100, 100, 120, 80, 100, 100, 80, 60, 60, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 60, 60, 60, 60, 40, 20, 20, 20, 20, 40, 120, 140, 140, 160, 160, 160, 160, 160, 140, 140, 140, 140, 140, 160, 160, 160, 180, 180, 180, 180, 180, 180, 180, 180, 160, 160, 160, 160, 140, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 80, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 40, 40, 40, 40, 20, 20, 20]

Выделение числовой последовательности, как и выполнение всей работы, проводилось при помощи программы на языке Python3.8 с использованием библиотек numpy, PIL и bitarray.

Исходный код программы, с помощью которой проводилось выделение сообщения:

```
#!/usr/bin/env python3
import numpy
from PIL import Image
from binascii import hexlify

# открываем исходное изображение, переводим его в вид numpy-массива
image = Image.open('cat.png')
image_data = numpy.array(image)

# проверяем, чтобы размер обязательно был 128 на 128
assert image.size == (128, 128), "Image should be of size 128x128 with
only one gray 8-bit channel"

# выделяем центральную строку,
# у нее будет индекс 63 т.к. нумерация с нуля
center_row = image_data[63]
# выводим её в виде списка чисел, а также в виде байтов,
# представленных 16-ричными значениями
print(f'Central row values: {list(center_row)}')
print(f'Central row bytes: {hexlify(bytes(center_row))}\n')
```

```
# проводим квантование, для этого с помощью библиотеки numpy
# делим каждое число массива на 20, потом округляем до ближайшего
# и этот результат домножаем на 20, после чего приводим
# к типу unsigned int размерностью 8 бит
quantized_row = numpy.asarray(numpy.round(center_row / 20) * 20,
dtype=numpy.uint8)
# выводим список значений и байты, как раньше
print(f'Quantized row values: {list(quantized_row)}')
print(f'Quantized row bytes: {hexlify(bytes(quantized_row))}')

# записываем полученное сообщение в виде байт для использования
# в других частях лабороторной работы
open('message.bin', 'wb').write(bytes(quantized_row))
```

Анализ сообщения

Упорядоченные символы алфавита с частотой встречаемости в сообщении

Символ	Частота в сообщении
20	0.21875
40	0.2265625
60	0.0859375
80	0.0703125
100	0.046875
120	0.03125
140	0.0625
160	0.1953125
180	0.0625

Количество символов в алфавите = 9
Энтропия всех символов алфавита = 2.8619012559140726

Из кол-ва символов можно получить расчетную длину равномерного двоичного кода:

- Минимальная длина кодового слова должна быть ceil($\log_2$(размер_алфавита)), где ceil - округление вверх (по формуле Хартли). То есть для нас это 4 бит.

Равномерное кодирование

Закодированное равномерным кодом сообщение
В битовом представлении:
00100100010101010100001000100001000100010010000100010001000100010001000
10010001100110011001100110011010001000101001101000100001100100010000100
00100010001000100010001000100010001000100010001001000100010001000100001000000
00000000000001010101100110011101110111011101110110011001100110011001110111
10111100010001000100010001000100010000111011101110111011001110111011101110111
01110111011101110111011101110111011100110000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001000100010001000000
00000000
В виде байт в шестнадцатеричном представлении:
24554221112111111123333334453443221111111111111222210000156677777766666777
88888888777767777777777777300000000000000000000001111000

Длина кодового слова = итоговая_длина / размер_сообщения = 512 бит / 128 = 4 бита, как мы и заранее посчитали

Количество переданной информации будет равнятся размеру сообщения, умноженное на количество информации, передаваемой одним символом. Соответственно получаем 128 * 4 бита = 512 бит

Исходный код программы, с помощью которой проводился анализ изначального сообщения и проведение вычислений, связанных с равномерным двоичным кодированием:

```
#!/usr/bin/env python3
import numpy
import struct
import bitarray
from binascii import hexlify

# открываем ранее записанное нами сообщение, уже после квантования
# и приводим его к виду массива байтов
data = numpy.array(list(open('message.bin', 'rb').read()),
dtype=numpy.uint8)

# с помощью библиотеки numpy получаем отсортированные уникальные
# элементы сообщения, а также кол-во повторений каждого элемента
# соответственно это алфавит и кол-во раз, которое встречается
# каждый символ алфавита в сообщении
alphabet, counts = numpy.unique(data, return_counts=True)
alphabet_size = alphabet.size
# просто выводим информацию об алфавите
print(f'Number of unique elements in alphabet (the size of the
alphabet): {alphabet_size}')
print(f'The alphabet itself: {list(alphabet)}\n')
```

```
# считаем вероятности символов, поделив каждое число в массиве
# с кол-вом повторений каждого символа на размер сообщения
ordered_probabilities = counts / data.size
# приводим вероятности к виду множества пар символ - вероятность
probabilities = dict(zip(alphabet, ordered_probabilities))
print(f'The probabilities of each letter: {probabilities}\n')

# считаем энтропию алфавита с помощью полученных вероятностей
# по формуле Шеннона, с помощью операций над массивами в библиотеке
# numpy
entropy = -numpy.sum(ordered_probabilities *
numpy.log2(ordered_probabilities))
print(f'Entropy of the alphabet = {entropy}\n')

# высчитываем минимальную длину равномерного кода по формуле
# ceil(log2(размер_алфавита))
binary_code_length = int(numpy.ceil(numpy.log2(alphabet_size)))

# создаем массив самих кодов, а также множество пар символ - код
# и выводим эту информацию
ordered_codes = [i for i in range(alphabet_size)]
binary_codes = dict(zip(alphabet, ordered_codes))
print(f'Minimal length of binary code = {binary_code_length}')
print(f'Equal-length binary codes for the alphabet: {binary_codes}\n')

# сохраняем файл, описывающий все коды, полученные при равномерном
# кодировании такого сообщения
# формат:
# 4 байта - размер алфавита
# разме_алфавита раз 1 байт - символ, 1 байт - код символа
binary_code_file = open('binary_alphabet.alph', 'wb')
binary_code_file.write(struct.pack('I', binary_code_length))
for letter, code in binary_codes.items():
  binary_code_file.write(struct.pack('BB', letter, code))
binary_code_file.close()

# считаем предполагаемую длину сообщения, закодированного нашим
# равномерным кодом
print(f'Printed binary code alphabet to binary_alphabet.alph')
print(f'Supposed length of message encoded with binary code =
{data.size * binary_code_length}\n')

# кодируем наше сообщение с помощью равномерного кода
# для каждого символа в сообщении берем его код, и приписываем к уже
# полученной последовательности кодов
encoded = bitarray.bitarray()
for value in data:
```

```
encoded.extend(bin(binary_codes[value])[2:].rjust(binary_code_length,
'0'))

# выводим информацию о длине сообщения, закодированного равномерным
# кодом, о длине каждого кода, а также само сообщение и кол-во
# информации в сообщении (которое равно длине полученного сообщения)
print(f'Length of message encoded with equal-length binary code =
{len(encoded)}')
print(f'Length of every letter: {len(encoded) // len(data)}')
print(f'Message encoded with equal-length binary code:
{encoded.to01()}')
print(f'Message encoded with equal-length binary code as hex:
{hexlify(encoded.tobytes())}')
print(f'Information transmitted with message (length * bits) =
{len(data) * binary_code_length} bits')
```

Кодирование Шеннона-Фано

Символ	Код
20	10
40	11
60	010
160	011
140	0001
180	0010
80	0011
120	00000
100	00001

Закодированное сообщение
В битовом представлении:
01000001000000000000001010010111111010111111111111111010001100110011001100
11001100001000010000000110000100001001101001011111111111111111111111111010
01001001011110101010110000000010001011011011011011000100010001000100010011
01101100100010001000100010001000100010011011011011000101101101101101101
01101101101101101101100111010101010101010101010101010101010101010101111111
11101010
В виде байт в шестнадцатеричном представлении:
41000297ebfff46666661080c2134bffffff492eab008b6db111116d9111111136d8b6db6db6
db3aaaaaaaaaabfea0

Средняя длина кодового слова (с учётом частот символов) = 2.90625, что находится в пределах 1-го бита от идеального (энтропии). Считалась, как мат. ожидание, т.е. сумма (частота_символа * длина_кодового_слова).

Количество информации, размер нашего сообщения, закодированного таким способом равно 372 бита, что уже заметно меньше 512, полученных при равномерном кодировании.

Степень сжатия, сравнивая с равномерным кодированием, равна 372 / 512 = 0.7265625 = 72.65625%

Степень сжатия, сравнивая с размером изначального сообщения, без квантования, равна 372 / 1024 = 0.36328125 = 36.328125%.

Относительная избыточность кодирования Шеннона-Фано в нашем случае равна 1 - 2.8619012559140726 / 2.90625 = 0.015259782911286

Исходный код программы, с помощью которой строились коды Шеннона-Фано и проводились аналитические вычисления связанные с этим кодированием:

```
#!/usr/bin/env python3
import numpy
import struct
import bitarray
from binascii import hexlify

# функция, которая строит коды шеннона-фано
# по сути бинарное дерево, т.к. рекурсивно строит для левого и правого
# принимает список пар символ-вероятность и префикс для всех
# символов в списке
def shannon_fano_tree(alph_probs, code = bitarray.bitarray()):
  if len(alph_probs) == 1:
    return dict([(alph_probs[0][1], code.to01())])
  left_part, right_part = 0, 0
  # когда у нас больше двух элементов, то построим массив
  # префиксных сумм вероятностей, и найдем, где стоит поделить,
  # чтобы сумма вероятностей с левой части была максимально близка
  # к сумме вероятностей с правой части
  if len(alph_probs) > 2:
    probabilities = numpy.array([i[0] for i in alph_probs])
    total_prob = numpy.sum(probabilities)
    prefix_sums = numpy.cumsum(probabilities)
    # бинарный поиск по отсортированному
    # массиву префиксных сумм
    split_point = numpy.searchsorted(prefix_sums, total_prob / 2)
    # если у нас элемент справа от середины на самом деле
    # ближе к ней, то сместим точку разделения
    if abs(prefix_sums[split_point] - total_prob / 2) <
abs(prefix_sums[split_point - 1] - total_prob / 2):
```

```
      split_point += 1
    # разделим список на 2 части
    left_part, right_part = alph_probs[:split_point],
alph_probs[split_point:]
  # иначе у нас только 2 элемента, присвоим левому
  # код заканчивающийся на 0, правому - на 1
  # и вернем как мн-во пар символ-код, которое
  # потом соединим с другими результатами
  else:
    return dict([(alph_probs[0][1], code + '0'), (alph_probs[1][1],
code + '1')])
  return {**shannon_fano_tree(left_part, code + '0'),
          **shannon_fano_tree(right_part, code + '1')}

# как в прошлый раз, прочитаем само сообщение
# и сделаем базовую статистику для него, посчитаем
# вероятности
data = numpy.array(list(open('message.bin', 'rb').read()),
dtype=numpy.uint8)

alphabet, counts = numpy.unique(data, return_counts=True)
alphabet_size = alphabet.size
print(f'Number of unique elements in alphabet (the size of the
alphabet): {alphabet_size}')
print(f'The alphabet itself: {list(alphabet)}\n')

ordered_probabilities = counts / data.size
probabilities = list(zip(ordered_probabilities, alphabet))
print(f'The probabilities of each letter: {probabilities}\n')

# теперь на созданном алфавите и вероятностях каждого символа
# построим коды фано
tree = shannon_fano_tree(sorted(probabilities))
print(f'Shannon-fano codes for this message: {tree}')

# закодируем наше сообщение полученными кодами
encoded = bitarray.bitarray()
for value in data:
  encoded.extend(tree[value])

# и выведем информацию о полученных результатах - длина
# закодированного сообщения, средняя длина кодов, кол-во инф.
print(f'Length of message encoded with Shannon-Fano code =
{len(encoded)}')
# считаем среднюю длину как мат. ожидание длин всех кодов
code_lengths = [len(tree[c]) for c in alphabet]
average_length = numpy.array(code_lengths)*ordered_probabilities
print(f'Average length of every code: {numpy.sum(average_length)}')
print(f'Message encoded with Shannon-Fano code: {encoded.to01()}')
print(f'Message encoded with Shannon-Fano code as hex:
{hexlify(encoded.tobytes())}')
print(f'Information transmitted with message = {len(encoded)} bits')
```

Кодирование Хаффмана

Символ	Код
20	01
40	10
60	000
160	111
140	0010
180	0011
80	1100
120	11010
100	11011

Закодированное сообщение
В битовом представлении:
000110111101011010110110000001010100001010101010101000011001100110011001
100110011011110111101011001101111011110000000010101010101010101010101010100
000000000001001010101101101000100010111111111111111001000100010001000101011
111111001100110011001100110011001100111111111111110010111111111111111111111
111111111111111111111000101010101010101010101010101010101010101010110101010
010101
В виде байт в шестнадцатеричном представлении:
1bd6b60542aaa1999999bdeb37bc02aaaaaa000956d117fff22222ff99999999fff97ffffffffc
55555555556a950

Средняя длина кодового слова (с учётом частот символов) = 2.90625, что, как и в случае с кодированием Шеннона-Фано, находится в пределах 1-го бита от идеального (энтропии). Считалась, как мат. ожидание, т.е. сумма (частота_символа * длина_кодового_слова).

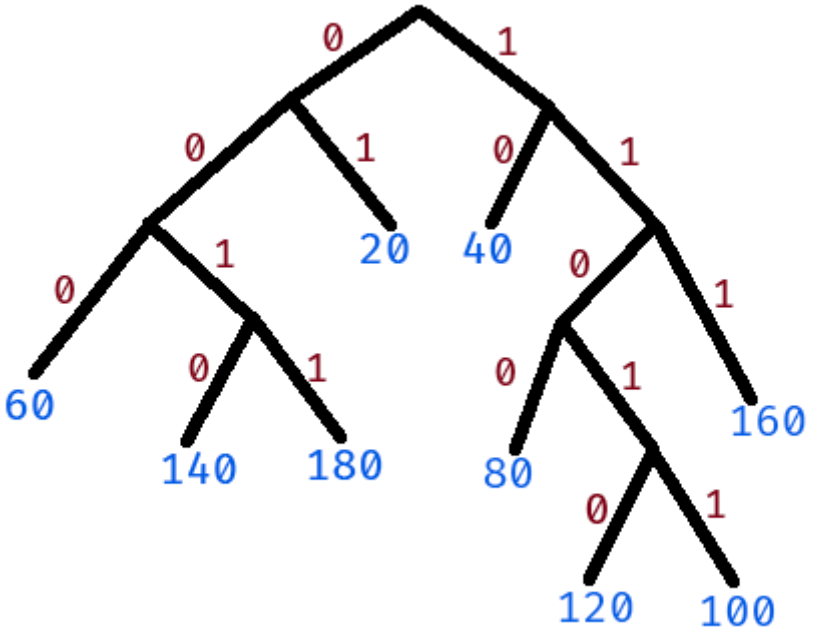
Количество информации, размер нашего сообщения, закодированного таким способом равно 372 бита, что в данном случае получилось равным результату с кодированием Шеннона-Фано. В общем случае кодирование Хаффмана должно быть сильно лучше, но так как у нас алфавит состоит только из 9-ти символов, разница между этими способами кодирования стала незаметной.

Степень сжатия, сравнивая с равномерным кодированием, равна 372 / 512 = 0.7265625 = 72.65625%.

Степень сжатия, сравнивая с размером изначального сообщения, без квантования, равна 372 / 1024 = 0.36328125 = 36.328125%.

Относительная избыточность кодирования Хаффмана в нашем случае также равна 1 - 2.8619012559140726 / 2.90625 = 0.015259782911286

Кодовое дерево Хаффмана:



Исходный код программы, с помощью которой строились коды Хаффмана и проводились аналитические вычисления связанные с этим кодированием:

```
#!/usr/bin/env python3
import numpy
import struct
import bitarray
import heapq
from binascii import hexlify

# функция, которая на вход принимает мн-во пар символ-вероятность
# и строит по нему дерево Хаффмана, а точнее коды для символов
def huffman_tree(probabilities):
  # создаем из символов и вероятностей мин. кучу, сортированную
  # по сумме вероятности всех символов, находящихся в поддереве
  # данной вершины
  heap = [[prob, [char, bitarray.bitarray()]] for char, prob in
probabilities.items()]
  heapq.heapify(heap)
  # строим дерево пока у нас не останется одна единственная вершина
  # которая будет корнем всего дерева
  while len(heap) > 1:
    # берем две вершины с самыми минимальными суммами
    # вероятностей символов в поддереве
    low = heapq.heappop(heap)
    # для каждого символа в левом поддереве добавляем к его
```

```
    # коду бит 0
    for code in low[1:]:
      code[1] = code[1] + '0'
    # а для правого бит 1
    high = heapq.heappop(heap)
    for code in high[1:]:
      code[1] = code[1] + '1'
    # и вставляем в кучу новую вершину, у которой вероятность
    # равна сумме этих двух, а символы это все символы из
    # левого и из правого поддерева
    heapq.heappush(heap, [low[0]+high[0]] + low[1:] + high[1:])
  result = dict()
  # просто переводим кучу в вид мн-ва символ - код
  for pair in heap[0][1:]:
    pair[1].reverse()
    result[pair[0]] = pair[1]
  return result

data = numpy.array(list(open('message.bin', 'rb').read()),
dtype=numpy.uint8)

alphabet, counts = numpy.unique(data, return_counts=True)
alphabet_size = alphabet.size
print(f'Number of unique elements in alphabet (the size of the
alphabet): {alphabet_size}')
print(f'The alphabet itself: {list(alphabet)}\n')

ordered_probabilities = counts / data.size
probabilities = dict(zip(alphabet, ordered_probabilities))
print(f'The probabilities of each letter: {probabilities}\n')

# здесь строим дерево кодов Хаффмана, а потом с помощью него кодируем
# как и с помощью кодирования Шеннона-Фано
tree = huffman_tree(probabilities)
print(f'Huffman codes for this message: {tree}')

encoded = bitarray.bitarray()
for value in data:
  encoded.extend(tree[value])

print(f'Length of message encoded with Huffman code = {len(encoded)}')
code_lengths = [len(tree[c]) for c in alphabet]
average_length = numpy.array(code_lengths)*ordered_probabilities
print(f'Average length of every code: {numpy.sum(average_length)}')
print(f'Message encoded with Huffman code: {encoded.to01()}')
print(f'Message encoded with Huffman code as hex:
{hexlify(encoded.tobytes())}')
print(f'Information transmitted with message = {len(encoded)} bits')
```

Выводы

Из полученных в этой лабораторной работе знаний и результатов, можно сделать вывод, что использование даже более тривиального кодирования Шеннона-Фано может привести к неплохой степени сжатия, а в случае с очень маленьким алфавитом даже к равной степени сжатия, полученной при использовании кодирования Хаффмана. На практике, когда алфавиты гораздо больше, то кодирование Хаффмана приводит к более высокой степени сжатия, очень близкой к теор. минимуму (энтропии). В нашем же случае полученные степени сжатия были такими:

- Кодирование Шеннона-Фано : 72.65625% по сравнению с равномерным кодом и 36.328125% по сравнению с изначальным сообщением.
- Кодирование Хаффмана : 72.65625% по сравнению с равномерным кодом и 36.328125% по сравнению с изначальным сообщением.