Data Analysis, Homework 2.2

Elizabeth Gould

May 24, 2025

I originally thought that we needed to encrypt the machines and use the machines to encrypt the text. However when I encrypted the machines, I found that numerical encryption didn't help. Here I have taken the arithmetic encoder from the text encryption and modified it to encrypt the machines. The characters and counts are encrypted separately. The characters use a dynamic, fixed alphabet encryption, with the alphabet sent in advance (sent by counting the spacing between remaining ASCII character encoding with a dynamically decreasing fixed bit width), while the counts use a dynamic alphabet encryption, with the alphabet sent at the end (after it has been calculated) in ternary encryption.

Sizes: full data = 570464 bytes, with excess data removed – 309627 to 322814 bytes, after shortening the bit size – 170 kB, and after encoding – 90640 bytes or 89 kB.

1 Encoder

1.1 Loading of the Uncapitalized Text

The first section of the encoder is used to set up the character alphabet and calculate the number of instances for each character given a character context length of 0 to 3 characters.

The alphabet is loaded into the following lists:

- low – all lowercase letters
- num – all digits
- sim – all special symbols with a non-zero occurrence rate
- ch – all three of these lists together in ascending ASCII order, The length of this list is 61 characters.

The list of all 1 to 4 character strings with non-zero count are recorded as follows:

- c1_non0 – length 61, all of the indexes of the list ch – machines with $k = 0$ and non-zero count
- c2_non0 – length 2×1427, indexes of machines with $k = 1$, or 2 character strings, with non-zero count; tuple of two arrays, both with values from 0 to 60
- c3_non0 – length 3×12699, indexes of machines with $k = 2$, or 3 character strings, with non-zero count; tuple of three arrays, all with values from 0 to 60
- c3n_non0 – length 2×12699, indexes of machines with $k = 2$, or 3 character strings, with non-zero count; tuple of two arrays, the first with values from 0 to 1426 (an index of c2_non0, the context), and the second from 0 to 60 (an index of ch, the new character)
- c4_non0 – length 2×59088, indexes of machines with $k = 3$, or 4 character strings, with non-zero count; tuple of two arrays, the first with values from 0 to 12698 (an index of c3_non0 or c3n_non0, the context), and the second from 0 to 60 (an index of ch, the new character)

The counts for the character strings are given as follows:

- count1 – counts for each character in alphabet ch (shape (61))
- count2 – counts for every two character string from alphabet ch (shape (61,61))
- count3 – counts for every three character string from alphabet ch (shape (61,61,61))
- count3n – counts for every character from alphabet ch given a context / machine with $k = 2$ (shape (1427, 61))
- count4 – counts for every character from alphabet ch given a context / machine with $k = 3$ (shape (12699, 61))
- c4_arr – array of length 59088 – all the non-zero counts from count4 in the same order as c4_non0

I will only be encoding and decoding – ch, c2_non0[1], c3n_non0[1], c4_non0[1], and c4_arr, as everything else can be calculated from these five arrays.

1.2 Unencoded Count and Numerical Encoding Tests

This is my attempt to use numerical compression for the data, but I found that a fixed width encoding is shorter. This section contains data with my attempts to numerically encode spacings between data. All code has been commented out, and will not currently run. To note:

If I were to encode each non-zero machine and count: Each character is 1 byte. Our maximum count is 724362, which requires 20 bits to encode. We could do this with 3 bytes (but not 2 bytes), but 4 bytes is more typical. Total cost, with typical encoding:

$$(1+4)*61+(2+4)*1427+(3+4)*12699+(4+4)*59088 = \\ 570464 \text{ bytes} = 0.544 \text{ MB}.$$

I first remove all excess information: First, I only need the counts for the 4 character combinations, as all others can be calculated from these. Second, the machines are given in order. Therefore, instead of sending all of c1_non0, c2_non0, c3n_non0, c4_non0, I can send just the special symbols from the ch array to create the alphabet and just the second index (or spacings of the second index as an alternative) from the c2_non0, c3n_non0 and c4_non0 arrays, and increment the first index as I surpass the size of the alphabet. The size of the previous array gives the maximum first index, thus providing a stopping point. In the cases where I need to increment the count, but the first index of the next machine is greater than the last index of the previous machine, I will need to add an extra number beyond the greatest character index (61 in my case). This is then read as incrementing the index of the machine, while not yet adding a new pair of indexes.

The new calculation is:

$$1*61+1*1427+1*12699+(1+4)*59088 = 309627 \text{ bytes} = 302 \text{ kB}.$$

This is the minimum size based on the above formula. The maximum size, if we add an extra character to indicate the need to increment the first index for every such case, is:

$$2*61+2*1427+2*12699+(1+4)*59088 = 323814 \text{ bytes} = 316 \text{ kB}.$$

I next tried to numerically encode all of these numbers using the encoding methods we discussed in class, but none of the numerical encoding methods were better than using a short fixed-length number.

1.3 Final Numerical Encoding Variant

I will skip over the classes for now, and return to them in the next section. This is because here I only use one of the three classes.

This is my final encoding with fixed-length numerical encoding, removal of repeated information, and no other data compression. This encoding works for decoding the text, and has a size of 170 kB. The characters have a fixed length of 6 bits (calculated by the decoder based on alphabet size) and the counts have a fixed length of 16 bits (given to the decoder as number w below).

The data sent is encoded as follows:

- p1: {10, 22, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 5, 1, 3}
  bit length: {7, 6, 6, 6, 5, 5, ..., 5, 4, 4, 4, 4, 4, 2, 2}

- p2: "ABC" – The first three characters of the document (as indexes of ch from 0 - 60).
- p3: c2_non0[1][:] – Insert 61 where needed.
- x = 61
- p4: c3n_non0[1][:] – Insert 61 where needed.
- y = 61
- p5: c4_non0[1][:] – Insert 61 where needed.
- z = 61
- w = $(\text{bin}\,(\max\,(count4)) - 2)$ = number of bits in max count in ternary encoding
- c4_arr

Indicies encoded 0-60 (straight) 61 indicates need to increment the first index only if not obvious.

The old code to load this data onto a binary stream is commented out, while the new version just encodes the alphabet as explained above. It is based on spacing between special characters. Bit length is determined based on the number of remaining characters – it is always just enough to go to the last number of the array (0 means end of list for everything but the first character.) I have also included the function to insert 61 where needed into the lists of indexes.

1.4 Classes

I have three classes:

- byte_mng() – loads bits and ternary numbers onto a unified byte stream, which is then saved
- count_mng(alphabet_in=None, counts_in=None, new_mode=2) – manages the dynamic counts for characters, required for determining windows for arithmetic encoding. Set up to be able to do static counts, dynamic counts, and dynamic counts with a fixed alphabet, as well as defining the effective count of the new symbol.
- arith_code_mng(counts_mngr, bits_store) – requires an instance of both of the above classes; My window size is 5 bytes, as 4 bytes doesn't work.

1.5 Arithmetic Encoding of Machines

p2 to p5 have a fixed alphabet encryption of length 62 (0 to 61). The removal from arithmetic encoding is identical to the case without any encryption. 5 bytes are "removed" from the byte stream while decrypting, but can be returned after the end of the file is reached. This means that I can just load characters into an encoder. End of text is noted like in the unencrypted case, by measuring known byte lengths. I can add 0s to the end, but it shouldn't even be necessary, as excess bits on the end of any type don't change the decoding, and I know they are for the next encoding.

- p2: "ABC" – The first three characters of the document (index 0 - 60).
- p3_arr = c2_non0[1][:], add 61 for unclear boundaries
- p4_arr = c3n_non0[1][:], add 61 for unclear boundaries
- p5_arr = c4_non0[1][:], add 61 for unclear boundaries

The code for adding 61 for unclear boundaries is in function np2D_to_list1D(arr, mng_arr, m_ind = 0), which puts cx_non0 arrays into the correct form. It is in the section Final Numerical Encoding Variant.

I end this section by adding a 1 and 39 zeros onto the byte stream (partial_finalize()). I actually needed to add more zeros equal to the number of bits from widening of the window on the array. I instead fixed this problem in the decoder, but there exist rare cases when this will not work. Since I can back up my reading of the byte stream, there should be a variant where I don't need to add so many 0s on the end, but I didn't want to develop one, and an extra 10 bytes is relatively small in context.

The next section is for the counts, which are encoded with a dynamical alphabet. This means that I no longer need to indicate the maximum count. Each potential count is a new character. I then finalize my arithmetic encoder (add 1 followed by 39 0s) and add the alphabet to the end in ternary encoding. Note that:

- I know the length of the count array already (59088).
- Adding the alphabet at the end of the stream doesn't cause any problems – I can just dynamically build the alphabet as indexes of the alphabet array, which will fully reconstruct my alphabet. If I add the alphabet at the end, I don't need to in advance send the size of the alphabet. And I will need padding at the end of the arithmetic encoder anyway if I add it to the encoding of the text, so this is not an issue.

The size is now 90640 bytes (89 kB on disk)

2 Decoder

2.1 Byte Management Classes and Functions

Loads libraries, defines my byte stream reader, and loads my file into the byte stream reader. The last step is to create the arrays of lowercase characters and digits, with the symbol array ready for reading.

2.2 Arithmetic Decoder Classes and Objects

Two classes, one for managing counts for the arithmetic decoder, and one for the decoder itself which requires a count class and a bit stream class. This is all reversing the previous code, and therefore in parallel.

2.3 Finding the Character Array

As I load my alphabet onto the stream, I need to retrieve it. This builds ch, my character list. For decoding the text, which characters are here will be important, but right now we only need the length.

2.4 Finding the Machines

Create an arithmetic decoder with a static alphabet and dynamic counts. Read p2,p3,p4,p5. Note that we already know how many characters we will be reading off of the array. Check that we retrieve what we loaded on, and stop at the correct places.

We then have a function end_stream() to back the byte stream up to the correct place, if we have bits. In my case, I should have added more zeros on, but it is unimportant here. This can also be used if I were to have done something requiring fewer bits of padding.

2.5 Finding the Counts

This is the second arithmetic decoder, with a fully dynamic alphabet. The decoded symbols are actually the indexes in the count array, not the counts themselves. I already know how many counts I need to remove from the array, based on the size of p5.

Again, I call end_stream() to bring the byte stream reader to the correct place after decoding the text. This time it is important, as bits = 2. I also check the accuracy of my results.

2.6 Getting the Counts

Now I need to retrieve my count alphabet. It is ternary encoded, with a known length (1476). I can just tell my byte stream to retrieve 1476 ternary numbers. That should bring me to the end of the file, but no further. If the text were here, I would then call to_end_of_byte() to start the next decoder at the next byte.

The numbers and order of them were found to be correct.

2.7 Forming the Count Array

This is simple. I have a count4 array of indexes of my count alphabet. I just form a new count4 array where every element is the ith element in the count alphabet, with i being the old value. This gives me the same count4 array as I used for decoding the text in the previous time.

3 Machines

The machines are then found with this function (from the arithmetic decoder of the full decoder program):

```
def find_nonzero_chars(self, context):
    ind1 = 0
    ind2 = 0
    char_arr = []

    i1 = p3_arr[0].index(context[0])
    ii = p3_arr[1].index(context[1], i1)
    if p3_arr[0][ii] != context[0]:
        print('find_index issue, p3, ', p3_arr[0][ii], ", ", jj)

    c3 for jj
    j1 = p4_arr[0].index(ii)
    jj = p4_arr[1].index(context[2], j1)
    if p4_arr[0][jj] != ii:
        print('find_index issue, p4, ', p4_arr[0], ", ", ii)

    c4 for kk
    ind1 = p5_arr[0].index(jj)
    if jj < len(p4_arr[0])-1:
        ind2 = p5_arr[0].index(jj+1, ind1) - 1
    else:
        ind2 = len(p5_arr[0]) - 1

    char_arr = p5_arr[1][ind1:ind2+1]

    return ind1, ind2, char_arr
```

Context here are the three indexes (from the ch array) of the three proceeding characters char_arr are the indexes (from the ch array) of the potential characters with this context, and count4[ind1:ind2+1] gives the counts of those characters.

I can also get the machines for shorter context, but after the first three characters, they are unneeded for a fixed alphabet encoding, so I never calculated them. count3 can be formed by the sum of all of count4s with a given context, count2 from the sum of all count3s with a given context, and count1 from the sum of all count2s with a given context.