3F7 Laboratory
Data compression: build your own CamZIP...

Dr Jossy Sayir
`js851@cam.ac.uk`

Michaelmas 2017

1 Introduction

In this lab, you will be led through the implementation and use of various data compression algorithms that you've learned about in the lectures, in particular

- Shannon-Fano coding;
- Huffman's algorithm;
- arithmetic coding.

This lab assumes that you have understood the basics of these data compression algorithms. If they haven't fully been covered in the lectures yet, you can still attempt the lab but the theoretical basis for some of the data compression methods you will study will only become clear once the methods are covered in the lecture.

The lab is currently designed in Matlab/Octave and all the example code provided has been tested both in Matlab and in Octave and uses commands compatible with both. Matlab is a commercial software that you can access free of charge[1] while at CUED either in the DPO, or for download and install on your personal computer. Octave is a GNU free software alternative to Matlab that uses the same syntax. There is extensive information about Matlab and Octave including links for downloading and installing on

`https://www.vle.cam.ac.uk/course/view.php?id=62572`

Before you start this lab, you should download the relevant files to your home directory. These are available on the moodle site as a package `3f7lab.tgz`. The following instructions assume that you are a novice to Linux and are working in the DPO. If one or both of these assumptions is incorrect, you may download the files however convenient and start Matlab or Octave[2]

1. in the DPO, log into a Linux workstation (NOT Windows!)

[1] It becomes very expensive once you are no longer a student, with prices in the £1000s for full commercial licenses.

[2] Just make sure that you point Matlab or Octave to the directory where the files are located either by `cd`-ing your way there or by adding the directory to the Matlab/Octave path.

2. open a web browser, log into moodle (`http://vle.cam.ac.uk`), surf to the course page and click on the lab file package `3f7lab.tgz`. Depending on your browser, you will either be prompted to select a location for the file (pick your home directory) or the file will automatically be downloaded to a "Download" folder

3. open a terminal window[3], find the file (e.g. by using the commands `cd` to change directory to where the file is located and `ls` to list the content of each directory). If necessary, move it to your home directory (e.g. by using the command `mv 3f7lab.tgz ~`).

4. uncompress the file using the command `gunzip 3f7lab.tgz` It is ironic that you need to use data decompression software in order to learn about data compression. Who knows? By the end of this lab, you may have invented a data compression algorithm that beats `gzip` / `gunzip`...

5. The resulting `.tar` file (short for "tape archive") is an archive that contains all the files you need. You can unpack the archive using the command `tar xvf 3f7lab.tar`

6. There should now be a new directory `3f7lab` in your home directory containing all the relevant files for this lab. Move to this directory (`cd 3f7lab`) and start Matlab or Octave by typing in `matlab` or `octave` at the command prompt. You should also start your favourite editor or use Matlab/Octave's editor.

In this lab, you will be guided through the use of a few Matlab/Octave functions we have written for you, and in some cases required to complete incomplete functions writing your own code. You are very welcome to implement the algorithms in this lab in the programming language of your choice if you prefer that to Matlab/Octave and we will support you in any language, but we currently do not offer pre-written code in those.

2 Trees and tables

The compression algorithms studied in 3F7 work by constructing a tree based on the probability distribution of the random variable you wish to encode. In order to implement these methods on hardware other than pen and paper, we need to learn how to represent a tree in computer memory. A tree according to graph theory is simply a graph with no cycles, or in other words a graph in which any two vertices are connected by exactly one path. There are various constructs and data structures in higher level programming languages for dealing with graphs and trees.

Matlab/Octave has a few functions for handling trees. We will only use one of these, which will be useful for visualising our prefix-free variable length code trees: `treeplot`. This function assumes that the tree is described by a simple vector, where each entry corresponds to one node in the tree, and its value is the index of its parent. An entry of zero is the "root" node that has no parent. For example `[0 1 2 2 1]` is a tree with 5 nodes, where nodes 2 and 5 are the children of node 1, nodes 3 and 4 are the children of node 2, and node 1 is the root. Nodes 3, 4 and 5 are the *leaves* of this tree. Type

```
treeplot([0 1 2 2 1])
```

[3]You may also do this step with a file and folder browser of your choice

in Matlab/Octave to visualise the tree. The result is slightly different in Matlab and Octave: Matlab re-orders the leaves in the tree so that the shortest paths are on the left, whereas Octave maintains the leaf order as specified.

Visualising trees is the mainstay of phylogeneticists (people studying how an elephant is related to a mongoose) and genealogists (people studying how your brother-in-law is related to your great-uncle.) There are many websites and software packages specialising in representing complex trees. A standard tree description syntax has been developed by phylogenetics researchers and is known as the "Newick format" (named after a restaurant where it was first proposed). We have provided a function to convert between Matlab/Octave's format and the Newick format. Type

```
tree2newick([0 1 2 2 1])
```

then cut and paste the resulting string into an online tree visualiser in your browser. We suggest[4] "`http://phylo.io`". `tree2newick` takes an optional argument `labels` for when the labels of the leaves and/or nodes do not correspond to the natural ordering $1, 2, 3, \ldots$. We currently only support numerical labels. Try the command

```
tree2newick([0 1 2 2 1], [0 11 22])
```

and visualise again with an online tree visualiser.

An alternative approach to storing a variable-length code is simply as a table of codewords. Matrices and arrays in Matlab/Octave and in most programming languages have a fixed number of columns and are thus ill suited for storing variable length objects. The workaround for this is simply to keep a list of lengths along with a matrix, where each row in the matrix contains a codeword up to its length and the remaining symbols in the row are irrelevant. We have provided functions `[c,cl]=tree2code(t)` and `t = code2tree(c,cl)` to convert between tree and table notation. Type

```
[c,cl] = tree2code([0 1 2 2 1])
```

to see an example of the table notation. The function returns a table `c` of codewords and a vector `cl` of codeword lengths. The first two codewords have length 2 and hence are the full first two rows of the code matrix `c`, 00 and 01. The third codeword has length 1 and hence the codeword is simply the first symbol of the third row, 1. The trailing 0 is irrelevant since it is beyond the length of the corresponding codeword. Now type `code2tree(c,cl)` and persuade yourself that the original tree is recovered.

The code table representation of a code is more useful for example when encoding data, whereas the tree representation may be more convenient when used for decoding or implementing a tree construction algorithm such as Huffman's algorithm. We have so far brushed over one minor detail, that the code and tree representations are not fully equivalent. Try for example the commmands:

```
tree2code(code2tree([0;1],[1;1]))
tree2code(code2tree([1;0],[1;1]))
```

[4] In the current version of `https://phylo.io` as of 25 October 2017 there is a minor bug. The tree is only drawn if the string is terminated by a newline. If you press "Render" after pasting the string as is, it won't draw anything. You need to place the cursor at the end of the string a tap return to add a newline, and then place render.

Both commands convert a code with two codewords of length 1 to a tree and back to a code table. The first code has codeword 0 in the first position, whereas the second code has codeword 1 in the first position. However, both operations result in the code that has codeword 0 in its first position. This is because the tree representation does not store the branch assignment order: by default, `tree2code` will label the children of a node with 0 or 1 in the order in which the children are listed in the node list. While the codes obtained this way are essentially equivalent in the sense that they have the same number of codewords of the same lengths, this discrepancy would cause problems for example when using a tree for decoding if symbols have different values than in the code used for encoding. To remedy this, we introduce a third and final representation for a prefix- free code which we call the *extended tree* or `xtree`. In this format, the tree is represented as a matrix where the first row is simply the tree as represented in the `tree` format above, and further rows indicate the indices of a node's children. For example a third column

$$\begin{bmatrix} 4 \\ 1 \\ 2 \end{bmatrix}$$

in the matrix indicates that Node 3 has Node 4 as a parent, and Nodes 1 and 2 as children, where the symbol '0' corresponds to Node 1 and the symbol '1' corresponds to Node 2. A zero child pointer indicates that the node has no child corresponding to that symbol, for example if the fifth column is

$$\begin{bmatrix} 7 \\ 0 \\ 3 \end{bmatrix},$$

then Node 5 in the tree has Node 7 as a parent and Node 3 as a child corresponding to symbol '1' and no child for symbol '0'. We have provided the functions `xt = code2xtree(c,cl)`, `[c,cl] = xtree2code(xt)` and `xt = tree2xtree(t)`. In the latter, children are assigned as per default. There is no need for a function to convert an `xtree` to a `tree` as this can be done simply by retaining the first row of an extended tree matrix `xt`. Repeat the example above converting from a code to a tree and back with `code2xtree` and `xtree2code` instead of `code2tree` and `tree2code` and persuade yourself that it solves the problem.

You are encouraged to play with more elaborate examples of codes and trees. What happens if your code has unused leaves in the tree? What happens if you supply a code that is not prefix-free? Can these functions handle non-binary codes or do they only work with binary codes? Try a few code tables and trees, convert between them, and visualise them with `treeplot` or with `tree2newick`.

You are only expected to learn to use the tools described in this section and do not necessarily need to study their implementations. Should you be curious about them, you are of course very welcome to examine the functions and ask questions or suggest improvements on the online forum.

Summary of tools covered in this section:

- `treeplot` - a tree plotting function native in Matlab and Octave
- `tree2newick` - a tool we provided to convert to the Newick standard tree description syntax

- `http://phylo.io` - a Newick syntax tree visualisation website
- `tree2code` - a tool we provided to convert from the tree representation to a code table
- `code2tree` - a tool we provided to convert from the code table representation to a tree
- `code2xtree` - a tool we provided to convert from the code table representation to an extended tree
- `xtree2code` - a tool we provided to convert from the extended tree representation to a code table
- `tree2xtree` - a tool we provided to convert from tree to extended tree representation

3 Shannon-Fano Coding

Your task in this section is to implement a MATLAB function that constructs a Shannon-Fano code. There are two ways you can approach the implementation of `sf(p)`:

1. as described in the course material, you could compute the lengths $\lceil -\log_2 p_k \rceil$ and construct the tree that has leaves at these depths.
2. Claude E. Shannon in his 1948 paper [**?**] introducing information theory describes a constructive method for the Shannon-Fano code on page 17[5]

Unless you are a confident programmer, we *strongly recommend* that you opt for the second method. If you opt for the first approach, there is quite a bit of programming involved and you will be asked to provide your program with appropriate comments to explain every step of the implementation. If you opt for the second approach on the other hand, we will provide the essential program steps and you will need to answer a few questions probing your understanding of Shannon's description on page 17 of his paper [**?**].

We have provided an incomplete function `shannon_fano(p)`. The function checks for the validity of its input, temporarily removes any symbols with zero probability from its input, orders probabilities in decreasing order, then calls an internal function `[c,cl] = sf(p)`. Your task is to provide this function `sf(p)`. Note that if your function constructs a tree rather than a table, you can use `xtree2code` to convert the tree to the required code table output. Within your function `sf()`, you can assume that all probabilities are non-zero and ordered in decreasing order. The calling shell function `shannon_fano` will then re-introduce the zero probability symbols, assigning them zero length codewords, and re-order symbols in their original order. Note that zero probability symbols should by rights be assigned *infinite* length codewords. Assigning them no codeword or, equivalently, a zero length codeword, achieves the same result because they are never expected to occur.

We now give an outline of Shannon's approach. Please read the passage in Shannon's paper [**?**] *before* reading this outline. First, you need to compute the *cumulative* probability

[5]The paper is available on the 3F7 Moodle site.

function corresponding to the random variable, where source symbols are ordered in order of decreasing probabilities. You may use the function `cumsum(p)` to do so, but note that `cumsum()` sums its input vector from 1 to k to calculate its k-th component so that the first entry is `p(1)` and the last entry is 1 when `p` is a probability vector, whereas Shannon requires that you sum it from 1 to $k-1$, so that the first entry is 0. Hence, you need to suitably alter the output of `cumsum()` to fit your purpose. Once you obtain the cumulative probabilities, Shannon's method consists in writing each cumulative probability as a binary number, and using that number without its leading 0 as the codeword, suitably truncated to the length $\lceil \log_2 p_k \rceil$. Shannon shows in his paper [**?**] that the resulting code is prefix-free. You may find the notion of a fractional binary number confusing, so the following example should hopefully clarify the notion:

$$0.43 = 4 \times 10^{-1} + 3 \times 10^{-2} \equiv 0.01101110\ldots = 2^{-2} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-7} + \ldots$$

If unsure about how to compute a binary representation from a number x, you can use the following command[6]to obtain a codeword of maximum length `n` from a value `x`:

```
c = double(dec2bin(round(x*2^n),n)) - '0'
```

You then need to truncate the codeword to the required length.

Once your Shannon-Fano code is working, try to generate codebooks for various probability distributions. Try for example a random probability distribution:

```
p = rand(1,10) ; p = p/sum(p);
[c,cl] = shannon_fano(p);
t = code2tree(c,cl);
treeplot(t);
```

If everything works, you should now see a tree with 10 leaves.

Try probability distributions that include values with probability zero. Examine the corresponding lines in the code matrix. Is the associated codeword length zero as intended? In the next section, you will have the opportunity to apply your Shannon-Fano code to actual data.

4 Compressing and de-compressing a file

We have provided a file `hamlet.txt` in the directory that contains the full text of Hamlet by Shakespeare. You can load the file using the commands

```
fid = fopen('hamlet.txt', 'r')
hamlet = fread(fid)';
fclose(fid);
```

[6]This operation works by multiplying your fractional number `x` by 2^m, then rounding the resulting integer, representing the result in binary using `m` digits, then converting to double noting that Matlab/Octave will view the binary string as characters and hence convert 0 to its ASCII value 48, which is why the trailing `- '0'` is necessary to convert back to 0s and 1s.

Make sure the first command doesn't return `-1` before proceeding to the following commands. If it does return `-1`, check that you are in the directory where `hamlet.txt` is located[7]. Include the apostrophe after `fread()` to make `hamlet` a row vector as it is more convenient for the next steps. Check that the file you accessed contains the text of Hamlet as expected using the command `char(hamlet(1:1000))`. This should display the first page or so of Hamlet. If you see a long vertical string of text instead of what looks like a page, you likely forgot the apostrophe: enter `hamlet=hamlet';` and that should solve it for you.

The vector `hamlet` consists of ASCII codes of the characters in the file `hamlet.txt`. Find an ASCII code table on the internet and have it ready for the next sections.

You can now obtain the frequency counts of the ASCII symbols in Hamlet using the commands

```
p = hist(hamlet, 0:255);
p = p/sum(p);
```

which instructs Matlab/Octave to return the number of symbols $0, 1, 2, \ldots, 255$ occurring in the file then normalise the resulting vector so it sums to 1.0.

Note that the file only contains ASCII codes for the characters

```
!&'(),-.:;?ABCDEFGHIJKLMNOPQRSTUVWYZ[]abcdefghijklmnopqrstuvwxyz|
```

and carriage return as you can check for yourself by entering `find(p)-1` (for the ASCII codes) and `char(find(p)-1)` (for the characters). The "`- 1`" is necessary because `find(p)` finds the non-zero positions in the vector `p`, where vector indices in Matlab/Octave are numbered starting from 1, but the ASCII code is numbered starting from 0.

You can now apply the Shannon-Fano algorithm to the probability vector you computed from the frequencies in Hamlet. Examine the code table `c` and `cl` returned by `shannon_fano(p)`. Note that this is now not formally a prefix-free code as you can persuade yourself by typing `code2tree(c,cl)`. This is because there are zero-length codewords that are technically prefixes of other codewords. This is irrelevant for the code at hand because the symbols with zero length codewords will never occur so these will never be used. You can restrict your attention to symbols with non-zero probability using the commands

```
nonzero = find(p);
t = code2tree(c(nonzero,:),cl(nonzero));
```

then view the resulting tree using the command `tree2newick(t,nonzero-1)` and an online tree visualiser. The second argument of `tree2newick` is the ASCII labeling of the characters with non-zero probability. This tree has too many leaves for you to examine usefully (how many leaves does it have?). To reduce the number of leaves and examine the tree graphically, you could for example convert Hamlet to upper-case only using the commands

```
lowercase = find(hamlet>='a' & hamlet<= 'z');
hamlet_uppercase = hamlet;
hamlet_uppercase(lowercase) = hamlet_uppercase(lowercase) - 'a' + 'A';
```

[7]Check using commands `pwd` to display the name and location of the current directory, `cd ..` to move up one directory, `cd directory_name` to change to sub-directory `directory_name`, `ls` or `dir` to list the contents of the current directory.

Check again by viewing the first 1000 symbols of the new vector that its contents are as expected. Repeating the steps above, you should now have a tree with 42 leaves. Examine the tree carefully.

Now, while we trust that you found it entertaining to examine code trees and play around with them, surely you are getting impatient and wondering when we will finally get to compress data rather than just examine code trees? We have provided a function `vl_encode(x,c,cl)` that takes an input data vector `x`, a codetable and code length vector `c` and `cl`, and outputs a binary vector of compressed data. There is an optional fourth argument `alphabet` for when there are codewords for symbols that don't occur in the data, as would be the case for example if you compress `hamlet.txt` using a code table that contains codewords (some of zero length) for all possible byte values from 0 to 255. Try the commands[8]

```
hamlet_bin = vl_encode(hamlet_uppercase,c,cl,0:255);
length(hamlet_bin)
```

If you followed all instructions, `hamlet_bin` should be a binary vector of length 940420. You may at first wonder why it should be considered a success to have compressed a file/vector of length `length(hamlet_uppercase) = 207039` to a vector/file of length 940420? Remember that the result you obtained is a binary vector. We have provided a function `bits2bytes(x)` that converts a binary vector `x` to a vector of bytes (groups of 8 bits) and a reverse operation `bytes2bits(x)` that converts back to the original binary vector. This is not as trivial as it sounds, because the number of bits may not be divisible by 8. If we simply stuff zeros at the end of the input vector to make it divisible by 8, we may be adding codewords to the end of a compressed stream, resulting in extra symbols being decoded when the compressed file is uncompressed. To avoid this, `bits2byte(x)` adds a 3-bit prefix to the beginning of the binary input string `x`, specifying how many bits must be appended to the new string of length `length(x)+3` to make it divisible by 8. This will be a number between 0 and 7 and hence can be encoded in 3 bits. Try the following commands

```
bytes = bits2bytes([0,1])
char(dec2bin(bytes , 8))-'0'
bytes2bits(bytes)
```

The first command converts the binary string `[0,1]` to bytes using the format described above. The second command allows you to visualise the contents of the variable `bytes`. The length of the input vector to `bits2bytes` is 2. The added 3 bit prefix brings the length up to 5. This means that 3 bits need to be appended to make up 8 bits or one byte. Hence, the value of the 3 bit prefix is the number 3 expressed in 3-bit binary format, `011`, followed by the input binary string `[0,1]`, followed by the 3 appended zeros. The last command recovers the original binary string of length 2 using the formatting described.

Now try the following

```
hamlet_cz1 = bits2bytes(hamlet_bin);
length(hamlet_cz1)
```

[8] We noticed that `vl_encode` is slow on some systems, e.g. Octave in the DPO. We provided an alternative `vl_encode_fast` that is not as easy to understand if you look at the code, but works faster on all systems. The new function only works with alphabet $\{0, 1, \ldots, 255\}$ (i.e. bytes) so it can't take an `alphabet` argument like `vl_encode`.

You should obtain a length of 117553, corresponding to a compression rate

$$R_{\text{cz1}} = \frac{117553}{207039} = 0.568.$$

Compression performance for data compression algorithms is more often measured in how many compressed bits are produced on average per input byte, giving the figure of

$$R_{\text{cz1}} = \frac{8 \times 117553}{207039} = 4.54 \text{ bits per byte.}$$

We have provided a function `H(p)` that computes the entropy of a probability distribution in bits. What is the lower bound to the compression rate achievable for the probability distribution the code was designed for? How far are we from that lower bound? How large should the compressed file be if we were able to achieve the lower bound?

Before we can celebrate the implementation our first compression algorithm, we need a quick reality check: how do we decompress our file? The history of information theory is littered with bogus claims of stellar compression rates that later unravelled as "one way" compression algorithms where you can compress a file but never recover the file from its compressed version.

We have provided a function `vl_decode(x,c,cl)` that decodes a binary compressed data string. Note that, internally, the function uses the extended tree representation of the code, after removing all symbols of non-zero probability and suitably indexing them. The function takes an optional fourth argument `alphabet` to be used if the source alphabet isn't simply the codeword indices minus one. Try the commands

```
hamlet_uppercase_decoded = vl_decode(hamlet_bin, c, cl);
char(hamlet_uppercase_decoded(1:1000))
```

If all went well, you should now again see the first page or so of Hamlet in uppercase. Note that the commands above assume that your last use of `c` and `cl` was after applying your Shannon Fano function to `hamlet_uppercase`.

Now that you've tested the encoding and decoding path succesfully, you may want to put it all together in one command. We've done this for you in a function `camzip1(filename)` that reads a file whose name, say '`myfile`', is provided as an input argument, writes the compressed output to a file `myfile.cz1` and saves the code table to a file `myfile.cz1c`. We also provided a function `camunzip1(filename)` that implements the reverse path. The resulting decompressed file uses the extension `.uz1` so as not to overwrite the original file. Try compressing a range of files with the Shannon-Fano algorithm and see how well it does. Download a few interesting files from the internet. You could for example download files from the Canterbury Corpus (`http://corpus.canterbury.ac.nz/`), one of the standard file repositories for benchmarking compression algorithms. You may neglect the storage space required to store the code table (those doing an FTR will learn more about how this can be avoided.) What can you say about the compression ratios achieved? Can you explain why some do better than others? You may want to start recording the performance obseved in a table at this point as you will be asked to test several algorithms throughout the remaining sectinos of this lab and you will eventually want to compare their performance.

Summary of tools covered in this section:

- `fopen, fread, fwrite, fclose` - Matlab/Octave functions for reading and writing files
- `bits2bytes, bytes2bits` - tools we provided for converting a stream of bits to bytes and vice versa
- `vl_encode, vl_decode` - tools we provided for encoding and decoding data using a variable length code/tree
- `H(p)` - a tool we provided to compute the entropy of a probability distribution in bits
- `camzip1, camunzip1` - skeleton functions putting it all together to compress/uncompress a file directly to another file

5 Huffman's Algorithm

To implement Huffman's algorithm, we need two data structures:

- a tree array `t` that we initialise to be an all zero array of leaves corresponding to the source alphabet. Note that a standard tree is sufficient for this. There is no need for an extended tree. In its initial state, this is an illegal tree where every leaf is a root and a leaf. The algorithm should end on a legal tree with only one root.
- a matrix `p` with two columns where the first column contains leaf/node probabilities and the second column contains the indices of the nodes they correspond to. This matrix is initialised with the source probabilities in its first column and the list of leaf indices 1 to n in its second column (where n is the alphabet size.)

Huffman's algorithm proceeds by selecting the two nodes in `p` with the smallest probabilities, creating a new parent node in `t` and connecting those two nodes to the parent, then replacing the two nodes in `p` with the new parent node and assigning as its probability the sum of the probabilities of the deleted nodes. The process is repeated until there is only one node left in `p` and its probability is one. By this point, all nodes in `t` should have been connected and the only node remaining with a zero entry is the root.

We have provided a skeleton function `t = huffman(p)`. Please complete the missing commands. The commands that need to be added are described in the comments. Once you have done this, test your algorithm and the resulting trees with a few simple probability vectors and visualise the tree to verify that the algorithm is running as intended.

We have again provided functions `camzip2` and `camunzip2` that call the Huffman function and work just like `camzip1` and `camunzip1` in the previous section. Compress the same files you compressed in the previous section using Huffman's algorithm and compare the results you obtain to the results obtained with the Shannon-Fano algorithm. Is the improvement significant?

6 Arithmetic Coding

In order to implement an arithmetic encoder effectively, we begin by highlighting the limitations of the encoder as described in the lecture notes. The procedure introduced in the lecture notes can be described as follows:

1. For a source string $(x_1, x_2, \ldots, x_n)$, determine an interval of length equal $P(x_1, \ldots, x_n)$ such that the intervals corresponding to all possible source strings are disjoint.

2. The interval can be computed recursively by subdividing the interval for a partial string $(x_1, \ldots, x_k)$ into sub-intervals of length proportional to the probability distribution of X_{k+1}, and picking the sub-interval corresponding to the value x_{k+1} in the string to obtain the interval for $(x_1, \ldots, x_k, x_{k+1})$.

3. The procedure above is repeated n times starting from the interval $[0.0, 1.0)$ to yield the interval for the sequence $(x_1, \ldots, x_n)$.

4. The codeword is chosen as the binary representation of the lower end-point of the largest dyadic interval $[\frac{j}{2^\ell}, \frac{j+1}{2^\ell})$ that fits within the interval computed for $(x_1, \ldots, x_n)$, where j, ℓ are integers.

Let us try this procedure in Matlab/Octave for a specific string. Load again the data vector `hamlet` from `hamlet.txt` and compute `p` the probability distribution inferred from its letter frequency counts as we did in the previous sections. We will initialise the interval end-points with `lo = 0.0` and `hi = 1.0` and compute the interval recursively for the first `n` symbols of `hamlet` for various values of `n`. Enter the following commands in one line[9] in Matlab/Octave:

```
n=5;lo=0.0;hi=1.0;for k=1:n;range=hi-lo; ...
   hi=lo+range*sum(p(1:(hamlet(k)+1))); ...
   lo=lo+range*sum(p(1:hamlet(k)));end
```

Note how the new interval is defined as the scaled interval between the sum of the probabilities up to the value of the symbol at position k and the sum of the probabilities including the next value. There is a subtle play with indexing here as Matlab/Octave indexing starts at 1 whereas ASCII indexing starts at 0, so if the ASCII symbol `0/NULL` is transmitted, the interval before scaling would be `[sum(p(1:0)), sum(p(1:1)))` which computes as `[0,p(1))` as intended. Now examine the value of `lo` and `hi` after the operation above and repeat the commands varying the value of `n` defined at the start of the line. What happens when you let `n` grow? What would happen if you tried to encode the whole file, setting `n = 207039`? What would step 4 look like? You can visualise an approximation of the output of Step 4 after `n` iterations (where we denote ℓ as `L` in Matlab/Octave) using the command

```
L = 128; dec2bin(lo*2^L,L)
```

where you can vary the value of `L` to your liking. What should `L` ideally be if we ran the operation above on the whole file setting `n = 207039`? Do you think that the file could be decoded from the value of the binary string obtained? Explain why?

[9] Note that entering "..." in Matlab/Octave allows you to write a "one-line" command in more than one line. If you have space to write the whole command on one line you do not need to copy the "...".

The lesson you have learned from the steps above is that the arithmetic encoding procedure as described in the lectures requires an infinite precision calculator. More precisely, it requires a calculator whose precision grows with `n` to yield sufficient significant digits in the expression of `lo` and `hi` so that their binary expansion is meaningful to the length of a binary string required for the encoding to be reversible. Matlab/Octave is *not* an infinite precision calculator. While arbitrary precision calculators are a topic of interest to computer scientists (see for example `https://www.gnu.org/software/bc/`), they are not in fact necessary for implementing an arithmetic code. There are a few tricks that can be used to remedy the problems that we observed in the steps above without resorting to an arbitrary precision calculator. They are:

1. You do not need to wait to end the interval computation step to start encoding bits. If the interval `[lo,hi)` is such that `hi > lo > 1/2`, then it is already clear that any dyadic interval within `[lo,hi)` will have a 1 in its first position. Similarly, if `lo < hi < 1/2`, there will be a 0 in the first position. In both cases, the interval can be rescaled. If `lo < hi < 1/2`, the interval end-points can simply be multiplied by 2. If `hi > lo > 1/2`, the interval end-points can be multiplied by 2 and shifted back to the $[0, 1)$ range by subtracting 1.

2. The procedure above works well when the data is such that the interval goes below or above the $1/2$ boundary. However, you may be unlucky and the data may be such that the interval will become smaller while straddling the $1/2$ boundary, thereby losing precision just as in the examples we tried above. This is not as bad as it sounds: if the interval is such that `1/4 < lo < hi < 3/4`, you know that when the straddling is finally resolved, any dyadic interval within the resulting interval will have either `10` at its beginning if the straddling is resolved above the $1/2$ boundary, or `01` at its beginning if the straddling is resoved below the $1/2$ boundary. Hence you can stretch the interval end-points by multiplying them by 2 and substracting $1/2$ while keeping a count of how many times you've done that before straddling is resolved. If straddling is resolved above the $1/2$ boundary after σ straddling counts, you need to output a 1 followed by σ zeros, while if it is resolved below the $1/2$ boundary, you output a 0 followed by σ ones.

3. The lack of infinite precision in the interval calculation still has one minor pitfall that would no longer hit us for `n` larger than about 10 as it did in the examples we tried above, but would probably cause problems once we tackle source strings in the thousands: because the interval end-points are subject to automatic rounding by the finite-precision floating-point calculation done by Matlab/Octave, there is a danger that the intervals corresponding to all possible source strings will no longer be *disjoint*. In the lecture, it was shown that this is the condition for the code to remain prefix-free and hence uniquely decodable. In order for the code to remain prefix-free, we have to take control of the rounding of interval end-points and ensure that any rounding is always *inwards*, i.e., `lo` can be rounded up in calculations and `hi` can be rounded down, but never the other way around.

The easiest way to go about controling the automatic rounding of interval end-points is to revert to integer arithmetic. We can define a number `max_integer` that we will consider to be the 1.0 boundary or initial interval high end-point, while integer 0 will remain the minimum boundary or initial interval low end-point. All arithmetic is done in full-precision floating

point, but by rounding the resulting interval end-points to either the integer above (using `ceil()`) or the integer below (`using floor()`), we can control all the rounding that happens in our finite precision implementation.

We have provided a partial implementation of the procedure described above in a skeleton function `y = arith_encode(x,p)`. This implementation is heavily inspired by the 1987 paper by Witten, Neal and Cleary [**?**]. The function again takes an optional third `alphabet` argument as above, and encodes an input string of bytes `x` to an output binary string `y`. Your task is to complete the function as instructed.

Implementing an arithmetic encoder is a delicate task. There are a few subtly different ways it can be implemented and it is essential for the decoder to exactly mirror the operations of the encoder if you want compressed files to be decompressed successfully. We have provided a full arithmetic decoder. Examine the code for `arith_decode(x,p,ny)` and you'll find that it follows the same outline as the encoder. The function takes as its input the binary compressed string, the probability distribution and the length of the original encoded string. When completing the encoding function, make sure you either mirror the operations in the decoder exactly, or if you opted for slightly different operations in your encoder, make sure you modify the decoder supplied so it mirrors your operations in the encoder.

Once you've completed the function, try to encode `hamlet_uppercase` ensuring that you still have the appropriate probability distribution `p` for it. Calculate the compression rate in bits per byte and compare it to the entropy `H(p)`. What do you observe? Use the decoder and verify that the file was decoded properly by viewing its first 1000 or so characters and checking its length.

We have put it all together for you in functions `camzip3` and `camunzip3` that call your arithmetic encoder and the arithmetic decoder we've provided. The function stores the original file length and the probability distribution used for encoding in the associated file with extension `.cz3c`. Have a go at compressing the same files you compressed with your Shannon-Fano and your Huffman codes and add them to your comparison. Comment on the compression rates achieved with arithmetic coding as compared to the other algorithms. Explain why the differences in performance are as they are. How could you improve on the performance of the arithmetic encoder?

7 Carrying on: adaptive and context-based compression

(This section only needs to be done for the Full Technical Report)

In this final section, we will not provide any function skeletons or tools and you are expected to write these by yourself. There are several ways you could go beyond the work in the 3F7 lab to bring it to FTR level:

1. Rather than measuring the frequencies in a file you want to compress and storing the resulting code or probability distribution in a separate file, you can calculate frequencies on the fly as you encode. As long as you only use frequencies measured up to and not including the current symbol in the encoding operation, the decoder will have processed the same string and worked out the same frequencies, so can reconstruct the code without the need to supply it separately. This approach is called "adaptive" or sometimes "universal" coding[10].

[10]Note that the word "universal" implies a proof that the adaptive approach is optimal in the sense that it

2. In the lecture, you've already seen that compression can be based on the conditional probability distribution given a number of past symbols. The compression rates we achieved in the previous section are limited to an independent and identically distributed model of text. Text is obviously not independent and identically distributed (e.g. $P_{X_{k+1}|X_k}('u'|'q') \approx 1$) so the string probabilities we based our encoder on are not the true string probabilities of the text. You can improve performance a lot by measuring the conditional probability distribution of the next symbol based on a context.

For the first option, note that simply basing your probability model on the frequency counts of the symbols you've seen so far is not a good idea: if you have never seen an 'a' until now, the frequency of 'a' will be zero and your Huffman or arithmetic code will not have a codeword for it. As a result, if 'a' occurs, you will not have a codeword to "tell" the decoder that this new symbol needs to be added to its statistics. There are several remedies for that: the Predicition by Partial Matching (PPM) family of algorithms (see `http://en.wikipedia.org/wiki/Prediction_by_partial_matching`) adds an "escape" character to the alphabet in every context to indicate that a new symbol needs to be added, and encodes the new symbol using a block code. The same approach can be adopted in a non-contextual adaptive compression algorithm. Other approaches consist in giving every frequency count an initial bias of a set number ε so that none of them are ever zero. When $\varepsilon = 1$, the resulting probability estimator is called the Laplace estimator. You should modify `camzip3` and `camunzip3` to use an adaptive probability estimator of your choice, either using an escape symbol or a biased estimator. You may want to embed your probability estimator or escape mechanism into `arithmetic_encode` and `arithmetic_decode`, or, alternatively, write a separate function to estimate probabilities. If writing a separate function, it would be helpful to modify `arithmetic_encode` to process one symbol at a time rather than the whole file, and ditto for `arithmetic_decode`. This would allow you to supply a different probability model for every call of the function. You would want to make use of Matlab/Octave's `persistent` variable definitions so that the interval end-points and straddle counter remain available when calling the function repeatedly. Your new `camzip4` and `camunzip4` should now be fully independent compression and decompression modules that map a file to a compressed file and back without the need for a separate code description. Compare the performance of `camzip4` to the performance of the previous algorithms and explain your observations. This step completes the minimum requirements for an FTR and you may either stop here or pursue the optional steps below if you want to learn more about the topic.

You may also want to look at adaptive Huffman coding. The direct approach to this would be to run Huffman's algorithm at every encoding step on the current estimated probability distribution. The decoder can do the same assuming it's decoded all the past symbols correctly and has access to the same partial statistics as the encoder had at this point. However, there is a much more elegant way to implement an adaptive Huffman encoder based on the observation that trees cannot change drastically from one step to the next assuming that only one frequency count will increase by 1. If you want to study and implement this algorithm, you could read the paper [**?**] by Robert G. Gallager "Variations on a theme by Huffman". This paper is available for download from the 3F7 lab moodle page and it's easily one of the prettiest and most accessible scientific papers in the information theory literature. Sections I to III of the paper set out the mathematical basis for the algorithm, proving three "theorems" that establish useful properties of a Huffman code. Section IV describes the actual algorithm

achieves the source entropy asymptotically.

implementation.

Finally, if you want to "break the world compression record", you will need to look at contextual arithmetic coding. To visualise how probability models for compression are improved by taking context into account, we suggest that you play with the Dasher information-efficient text-entry interface (`http://www.inference.phy.cam.ac.uk/dasher/`). To embed context into your arithmetic encoder/decoder, try at first to estimate conditional distributions for every context of length κ as you did for the adaptive arithmetic encoder above. Try $\kappa = 1, 2, 3$. Measure the performance obtained in function of the length of context for files of different length. One "cheat" consists in taking *hamlet.txt* and concatenating 10 or 100 copies of it into one file, to see how the compression improves once the algorithm is able to accumulate more statistics. There are several ways to juggle the advantages of deep contexts (= better statistics in the long term) with its disadvantages (= more time/data needed for the model to converge). One way consists in using an escape symbol as hinted above. Another way consists in combining estimates from contexts at different lengths. A full study of optimal contextual statistical measurement is beyond the scope of this lab and you should refrain from spending too much time on this, but if you find the topic interesting, consider taking the Machine Learning module 4F13, the Speech and Language Processing module 4F11, in addition to the Advanced Information Theory module 4F5 which we trust that you will definitely take since your interest for 3F7 led you to do an FTR for this lab.