COM4115 / COM6115: Text Processing (2017/18)

Assignment: Text Compression

NB: COM3110 students should NOT attempt this assignment { it is not part of the assess-ment for that module. It is for students of COM4115 / COM6115 only.

Task in brief: To implement the Hu man Coding text compression algorithm in Python and to investigate the advantages and disadavantages of character-based vs word-based encoding.

Submission: Submit your assignment work electronically via MOLE. Precise instructions for what les to submit are given later in this document. Please check you have access to the relevant MOLE unit (i.e. COM4115 Text Processing (AUTUMN 2017 18) or COM6115 Text Processing (AUTUMN 2017 18)) and contact the module lecturer if not.

SUBMISSION DEADLINE: 12 noon, Monday week 11 (4 December, 2017)

Penalties: Standard departmental penalties apply for late hand-in and use of unfair means.

Materials Provided

Data le: The Project Gutenberg version of Melville's novel mobydick.txt is supplied as a data le for development and testing (you may want to use a small portion of this text for initial development and testing).

Task Description

Your task is write a two Python programs, one called huff-compress.py and the other called huff-decompress.py.

huff-compress.py should take a UTF-8 encoded plain text le (we will refer to this as infile.txt) as its rst argument and should behave as follows:

1. Using infile.txt as data, the program should count the frequency of the symbols to be assigned Hu man codes (either characters or words { see Note 1 below) and should build a zero order model for symbol probabilities. To estimate the probabilities in the zero order model use the relative frequency of each symbol in infile.txt (the relative frequency of a symbol s in a text T is the proportion of symbol occurrences in T that are s). Take special care in dealing with non-printable characters.
2. The program should compute a Hu man code-tree for your infile.txt symbol set, using the zero order model computed in the previous step to assign probabilities to the nodes in your tree. One method to do this is described below in Note 2.
3. Once the code tree is complete the program should write two output les (see note 3):
   1. The rst, which should be called infile-symbol-model.pkl, should contain a serialized version of your symbol model (which is probably a dictionary), produced using the Python Pickle module. See Note 4 below.

COM4115 / COM6115 Page 1 Assignment: 2017/18 (v1)

1. The second, which should be called infile.bin, should contain the text of infile.txt compressed using your symbol model. Be careful to write to write this data out as binary data and not, e.g., as an ASCII representation of 1s and 0s. See Note

5 below. Also, to deal with the problem of your encoding potentially ending in the middle of a byte, be sure to add a pseudo-EOF (end-of- le) character to your symbol model and to add this character to the end of your encoded data. See Note 6.

huff-decompress.py should take a binary le encoded using your hu -compress.py progam (we will refer to this as infile.bin as its rst argument and should behave as follows:

1. It should look for a le called infile-symbol-model.pkl in the same directory as infile.bin and should use pickle (Note 4) to read in a recreate the symbol model.
2. Using the symbol model acquired in the previous step the program should read the binary data le infile.bin, decode it into plain text and then write it to an output le called infile-decompressed.txt. Obviously, infile-decompressed.txt should match the original infile.txt (the Unix/Linux utility diff can be used to test this).

(Normally compression programs would simply write the output to a le called infile.txt, overwriting any existing le with that name. This is not a good idea during development or assessment of such a program, however.)

What to Submit

Your assignment work is to be submitted electronically using MOLE, and should include:

1. Your two Python programs huff-compress.py and huff-decompress.py. Do NOT sub-mit any other Python codes les.
2. A short report (as a pdf le), which should NOT EXCEED 2 PAGES IN LENGTH. The report should include a brief description of the extent of the implementation achieved (this is especially important if you have not completed a su cient implementation for performance testing). It should also address the following questions:
   1. How does the compression performance of character-based versus word-based Hu man compare? Speci cally:
      1. How big (in bytes) are the compressed text les produced in each case?
      2. How big (in bytes) are the symbol models produced in each case?
      3. How long does it take to
         1. build the symbol model?
         2. encode the input le given the symbol model?
         3. decode the compressed le? (including unpickling the symbol model)

See Note 7.

1. How could various aspects of the performance, as identi ed in response to the previous question, be improved?

COM4115 / COM6115 Page 2 Assignment: 2017/18 (v1)

Any other observations you believe are pertinent to comparing character-based and word-based Hu man coding should also be made. Graphs/tables may be used in presenting your results, to aid exposition.

Assessment Criteria

A total of 25 marks are available for the assignment and will be assigned based on the following criteria.

Implementation and Code Style (15 marks)

How fully and correctly has the functionality speci ed in the \Task Description" section above been implemented? How e cient is the implementation (i.e. how quickly are results returned)? Have appropriate Python constructs been used? Is the code comprehensible and clearly com-mented? You can expect your program will be run on mobydick.txt and perhaps on other texts as well. Amongst other things it will be checked that the result of compressing and then uncompressing a le is identical to the original le prior to compression.

Report (10 marks)

Is the report a clear and accurate description of the implementation? How complete and accurate are the answers to the questions posed above in the \What to Submit" section? What conclusions can be drawn about character-based versus word-based Hu man coding from these results?

Notes and Comments

1. Tokenization: Your program should allow a user to decide whether the symbols to be as-signed Hu man codes will be characters or words. This should be speci able via command line arguments which follow the syntax: -s char for generating codes at the symbol level and -s word for generating codes at the word level. To identify the words in the input text you should:
   1. treat every continuous sequence of alphabetic characters (of either upper or lower case) as a word { i.e. use a regular expression that matches \a-zA-Z" and consider each match a word;
   2. treat every other character (numerals, punctuation, white space) as a word.
2. Building a Hu man Code Tree:
3. Create a class whose instances will serve as nodes in (binary-branching) code trees. Instances might have attributes for the symbol (for use in leaf nodes), the node's probability, and for storing left and right subtrees (all of which might be initialised to None).
4. Create a list of (leaf) nodes for the individual symbols (each also storing the node's associated probability).
5. Sort this list on the basis of node probability, so that the two lowest probability nodes can be found together at one end of the list.

COM4115 / COM6115 Page 3 Assignment: 2017/18 (v1)

* 1. Remove the two lowest probability nodes from the list, and combine them to form a new subtree, i.e. as the left and right daughters of a new node.
  2. Add the new node back into the list of nodes, maintaining sorted order (w.r.t. node probability). An easy way to do this is just to append it and then re-apply the sort method (which is not optimally e cient { you may want to consider a more e cient alternative).
  3. Repeat steps (d) and (e) until only a single node remains in the list.
  4. Finally, traverse the tree of the remaining node, to assign binary codes to the symbols and return them (e.g. as a dictionary).

1. Format of Compressed File(s): A more user-friendly approach to implementing text compression using Hu man coding is to put both the symbol model and the compressed text into a single output le. However this requires more e ort than is warranted for this assignment and makes it harder to independently assess the size of the symbol model and compressed text les for di erent choices of symbol, i.e. for word-based or character-based encodings.
2. Pickling your Data: pickle is a Python module for serializing data, allowing com-plex data structures, such as dictionaries, to be written to and read from disk (i.e. they may be \preserved"). In the current case after importing pickle you should be able get away with using just the pickle.dump and pickle.load methods. See https:// docs.python.org/3/library/pickle.html?highlight=pickle#module-pickle for full details. You might nd https://wiki.python.org/moin/UsingPickle and https:// pythontips.com/2013/08/02/what-is-pickle-in-python/ helpful as well.
3. Writing and Reading Binary Data: Consider the example of Hu man coding from the rst lecture on Text Compression where the string `eefggfed' is coded as in binary as: 10101101111111101001. If you naively write this to disk as a string each binary symbol (`0' or `1') will be written in one byte resulting in a string of length 20 being used to represent an original string of length 8! This is hardly text compression. To get around this problem your string representation of a binary number must be converted to a binary representation

where each 8 digits in the string representation (which at one byte each takes up 8 8 = 64 bits) are converted to into a single byte (8 bits). To do this Python you should use the int built-in which if you call it as

c = '11001100' b = int(c,2)

will take an 8-character string such as c and convert into an integer base 2 representation (i.e. one byte).

Thus you should think of buidling up the whole text to be encoded into a single string of `1's and `0's and then loop through this 8 characters at a time converting each 8 characters into a one byte binary representation. If these one byte representations are added to an array then a proper binary representation of your encoding may be constructed. Here are some code fragments to illiustrate this:

COM4115 / COM6115 Page 4 Assignment: 2017/18 (v1)

import array

...

codearray = array.array('B') # create array of bytes

...

* somewhere inside a for loop where 8 string 1's and 0's are being
* placed into c

codearray.append(int(c, 2)) # convert c to binary and add to array ...

* Write the array at once to a file

f = open('workfile', 'wb') codearray.tofile(f) f.close()

You may nd this discussion helpful: https://www.linuxquestions.org/questions/ programming-9/writing-binary-data-under-python-718165/. Also helpful, if you are developing on a Unix-based platform (Linux, Mac OSX), is the system command hexdump, which prints out a dump of a le's contents in hexadecimal, 8 bytes per line. The -C ag also prints the ASCII interpretation of the line. This lets you check, on small example les during development, that you are getting things right.

1. Pseudo-EOF: Your binary encoding of the input text may be an arbitrary number of bits in length. However, when you go to write it to le you will need to write an integer number of bytes. Returning to the example from the lecture mentioned above, the 8 character original string `eefggfed' will end up being encoded in 20 bits as 10101101111111101001. When this is converted to 8-bit bytes we end up with two bytes + 4 bits. To write the last 4 bits we need to add 4 more bits to make up an integer number of bytes. The problem is if we just add arbtirary bits to nish o the last byte then when we come to decode our decoder may interpret these as further characters to be decoded. How is it to know the di erence?

There are a number of solutions to this problem. The solution recommended here is to add a special symbol to your symbol set, call it , that serves as a pseudo-EOF marker. It gets used just once { it is the last symbol you add to your encoded output and anything can be placed in any residual bits in the last byte following it (e.g. `0's). On decoding, when  is encountered you know you are at the end of the text to be decoded. Further discussion of this approach can be found at https://stackoverflow.com/questions/14667681/ last-byte-in-huffman-compression and in the pdf le at https://web.stanford.edu/ class/archive/cs/cs106b/cs106b.1126/handouts/220%20Huffman%20Encoding.pdf.

1. Timing: There are various ways to time your code's execution. If you are on a Unix-based platform (Linux, Mac OSX) you can use the Unix system command time. There are Windows equivalents, but they are not so straightforward. Alternatively there are cross-platform timing commands available with Python, which you can use.

COM4115 / COM6115 Page 5 Assignment: 2017/18 (v1)