Lab: Information, Entropy and English

CSI 3305: Introduction to Computational Thinking

January 8, 2011

1 Introduction

How many binary digits per character should it take to encode English text? The seemingly straightforward question has deep connections to the field of information theory. Information Theory is a field of mathematics developed in nineteen forties by Claude Shannon, dealing with the quantification and encoding of information. Through Shannon's work, an objective measure of information was developed (*entropy*), which allowed questions like the one posed to be answered in a rigorous manner. The entropy of a source, measured in bits per symbol, is the average number of binary digits per symbol necessary to encode messages produced by that source, often for transmission.

Hardware developers and bandwidth providers would like to minimize costs by using efficient means of encoding and transmission. One way to do this is to use the minimum number of binary digits necessary for encoding English characters. The entropy of a message source gives us what this minimum number of digits actually is. We begin with a simplified model of English text that only contains the twenty-six uppercase English language letters plus a space symbol. One naïve approach to encoding such a source would be to simply use 5 binary digits per character, which allows for the encoding of 32 symbols in total. This would provide codes for all our 27 characters.

However, this approach is wasteful for two reasons. First, we only need to encode 27 characters, but are using enough digits for the encoding of 32, thus leaving some of the binary digit patterns unused. Second, according to Shannon's theory, the entropy of English text is less than 5 bits per symbol, and is actually closer to 1 bit per symbol. Therefore, by using insight and tools from information theory, we can aim to reduce the average number of binary digits necessary to encode the symbols of our message. To do this, we must calculate the entropy of our source, to know how many binary digits are necessary for encoding each symbol.

As a side note, information theory only applies to ergodic sources, and so holds approximately for English text (which is not truly ergodic.) For an in-depth explanation of information theory as well as some of the nuances, see Wikipedia: http://en.wikipedia.org/wiki/Information_theory

2 Problem Statement

Your task is to come up with an efficient method of encoding English text, based on statistics gathered from a sample document. You will create tables of letter frequencies from the provided file, and use these frequencies to estimate the entropy of English text. In the second part, you will create an encoding scheme for transmitting the text and show that your code uses less than one binary digit per symbol more than the estimated entropy of the text in bits per symbol.

3 Tools

You will be using the Python programming language and Python interpreter for this lab. The code will be written using a text editor, and the interpreter will be called using the command line.

4 Setup and Programming

4.1 Estimating Entropy

In this section, we will calculate both the unigram and the bigram entropy from a source file. This will serve as an estimate for the unigram and bigram entropy of English text.

1. To begin, create a new text file and label it entropy.py.
2. Open the text file in the text editor of your choice (such as Notepad or Vim.)
3. Enter the following code at the top of your file:

```
from __future__ import division
from collections import defaultdict
from math import log
```

   The first line of code will allow Python to handle division in a more intuitive way, doing floating point division by default. The other import statements will import modules from the standard python libraries, which will allow us to use the default dictionary data type and have access to a math function that calculates logarithms.

4. Next, we will create a function to calculate entropy from a file. Type the following code into your file:

```
''' Functions '''
def getNGramData(filename, n):
    f = open(filename, "r")
    content = f.read().rstrip()
    f.close()
    total = len(content) - (n - 1)
    ngrams = defaultdict(int)
    for i in range(total):
        ngrams[content[i:i+n]] += 1
    for key, value in ngrams.items():
        ngrams[key] = value / total
    return (ngrams, total)
```

   This function first opens a file by the name of `filename` in read-only mode, then extracts all content from the file and saves it in the variable `content`. After closing the file, the total number of positions in the file is calculated, which is based on the width of the sliding window, `n`. A default dictionary object, `ngrams`, is then instantiated, which sets a default initial value of zero for each key. After this, we loop through the content using a sliding window based on `n`, and each corresponding n-gram is counted in our `ngrams` table. After this, we divide each count by the total number of positions, to get the probability of a particular n-gram. Lastly, we return a tuple containing both our n-grams table as well as the number of positions in our file.

5. Next, we will declare a couple variables. Enter the following code, beneath what you've already entered:

```
''' Vars '''
alphabet = " ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
filename = "largefile.txt"
```

   The first variable is simply a string of all the characters we're considering in our simplified language. The second is the name of the file we will process to get estimates of English text entropy. You will need to download this file from the course website.

6. We then call the previously defined `getNGramData` function, passing the filename and desired n-gram value, to calculate our frequency tables from the file. Enter the following code:

```
''' Calculate NGram Data '''
unigrams, unigramCount = getNGramData(filename, 1)
bigrams, bigramCount = getNGramData(filename, 2)
```

7. Lastly, we calculate the entropy of the text using the unigram and bigram tables, based on Shannon's formula:

```
''' Unigram Entropy '''
H = 0.0
for key, value in unigrams.items():
    H += -value * log(value, 2)
print "H (unigram):", H, "bits per symbol"

''' Bigram Entropy '''
H = 0.0
for key, value in bigrams.items():
    unikey = key[:1]
    H += -value * log(value/unigrams[unikey], 2)
print "H (bigram):", H, "bits per symbol"
```

   This code calculates the entropy in both the unigram and bigram cases and displays them on the screen.

8. Save the file and open the command line (From the Windows menu bar: Start > Run... > cmd) Navigate to the folder where you created your file (use the following command: cd "C:\Folder\where file\is".)

9. From the command line, type the following to run your program and hit enter:

```
python entropy.py
```

   You should see the estimates for entropy of the text for both cases. Record these two values.

4.2 Huffman Encoding

Now that we have estimated the entropy of English text, we will devise a coding scheme based on letter frequencies that achieves an efficiency, as measured by bits per symbol, near the entropy of the text. We will do so by using Huffman coding, which is described in detail here: http://en.wikipedia.org/wiki/Huffman_coding.

1. To begin, create a new text file in the same folder as your entropy.py file, and label it huffman.py.

2. Open the text file in the text editor of your choice (such as Notepad or Vim.)

3. Enter the following code at the top of your file:

```
from __future__ import division
import heapq
```

   The first line of code was explained previously, and the second imports the priority queue (heap) data type.

4. Next, we will create a class object. Below the import statements, enter the following code:

```
class Huffman:
    def __init__(self, n):
        self.binary_to_symbol = {}
        self.symbol_to_binary = {}
        self.n = n
```

   This defines a class called `Huffman` and defines its initialization method. The method instantiates two dictionary objects, which will hold the coding information, and saves a variable `n` that tells us whether we're using unigrams or bigrams for encoding purposes. Notice the indentation, as you will get an error if things are not properly aligned, matching the indentation shown above.

5. Beneath the initialization method, you will add more methods to your class. Enter the following code:

```
    def recurse_subtree(self, tree, prefix):
        ''' Base Case '''
        if type(tree) == str:
            self.binary_to_symbol[prefix] = tree
            self.symbol_to_binary[tree] = prefix
            return
        ''' Recursion '''
        self.recurse_subtree(tree[0], prefix + "0")
        self.recurse_subtree(tree[1], prefix + "1")

    def build(self, keys, ngrams_table):
        tree = []
        heapq.heapify(tree)
        for key in keys:
            value = ngrams_table[key] if key in ngrams_table else 0.0
            heapq.heappush(tree, (value, key))
        while len(tree) > 1:
            subtree1 = heapq.heappop(tree)
            subtree2 = heapq.heappop(tree)
            p_combined = subtree1[0] + subtree2[0]
            sub_combined = (subtree1[1], subtree2[1])
            heapq.heappush(tree, (p_combined, sub_combined))
        root = heapq.heappop(tree)[1]
        self.recurse_subtree(root, "")
```

The first method, `recurse_subtree()`, is a simple recursive function for building the prefix codes for encoding and decoding. The second method, `build()`, uses a priority queue and an n-gram table (like the one we constructed in the previous section) to build our Huffman coding tree. It first adds all the n-grams with their probabilities in the queue. (Notice, it also adds unseen letter combinations as occurring with a 0 percent probability. This becomes important with n-grams of $n \geq 2$, since our source file may not have contained all possible combinations.) It then recursively builds the Huffman tree, by popping off the two items with the lowest probabilities, combining them into a "supernode", then adding this node to the queue. When the queue consists of a single item (the whole Huffman tree), the loop terminates and the method uses this tree to assign prefix codes for the letter combinations. It does this by calling the recursive method we defined previously.

6. Next, we will define our encoding and decoding methods. Enter the following code:

```
def encode(self, text):
    output = []
    total = len(text)
    for i in range(0, total - (self.n - 1), self.n):
        code = self.symbol_to_binary[text[i:i+self.n]]
        output.append(code)
    return "".join(output)

def decode(self, cipher_text):
    index = 0
    offset = 0
    length = len(cipher_text)
    output = []
    while index + offset <= length:
        prefix = cipher_text[index:index+offset]
        if prefix in self.binary_to_symbol:
            output.append(self.binary_to_symbol[prefix])
            index += offset
            offset = 0
        else:
            offset += 1
    return "".join(output)
```

These two methods encode plain text and decode cipher text, respectively.

7. Save the file and open the entropy.py file (if you do not already have it opened.) Change the import statements at the top of the entropy.py file to the following:

```
from __future__ import division
from collections import defaultdict
from math import log
import huffman
```

This will now import our Huffman class we created, as long as both files are in the same directory folder and are named as indicated previously.

8. Go to the bottom of the entropy.py file. After the existing code, enter the following:

```
'''******* HUFFMAN ENCODING **********'''
```

```
''' Vars '''
filename = "largefile2.txt"
keys = ["".join([i,j]) for j in alphabet for i in alphabet]
```

These lines define two new variables. The first stores the name of the file we will be encoding, which can also be downloaded from the course website. The second builds a list of all possible bigram letter combinations, using the alphabet variable defined previously (in Part I.)

9. After this, enter the following code:

```
''' Create Bigram Huffman Encoder for Source '''
huff = huffman.Huffman(2)
huff.build(keys, bigrams)
```

First, we instantiate an instance of the class we created, `Huffman`, which will be used for bigram encoding (hence the 2 passed into the initializer.) We then call the `build()` method, passing in the `keys` variable and the bigrams table we computed previously. This will build the encoding/decoding tables used for the encoding of text.

10. Next, we will calculate how many bits per symbol, on average, our Huffman encoder requires for encoding English text from our simplified alphabet. Enter the following code:

```
''' Calculate Average Number of Binary Digits Per Symbol Using Huffman Encoder '''
avgPerSymbol = sum([value * len(huff.symbol_to_binary[key]) \
        for key, value in bigrams.items()]) / 2
print "Average binary digits per symbol using Huffman encoding:", avgPerSymbol
```

We make use of the python `sum()` function, which adds together items in a list, and compute the average by multiplying the number of binary digits in a given symbol code by its probability of appearing, as stored in our bigrams table. Notice, however, that we must divide this number by bigram length, 2, since two symbols are contained in each bigram. To find the average number of binary digits *per symbol*, we must divide by two.

11. We must now test our encoder, to see how well it actually encodes English text. We will compare this to an estimate of how well it should encode text, based on the bigram entropy of the source. Enter the following:

```
''' Estimate Bigram Encoded Filesize '''
unigramCount = getNGramData(filename, 1)[1]
print "Estimated encoded filesize:", ((H * unigramCount) / 8) / 1024, "KB"
```

The first part gets the number of symbols in the file to be encoded. The second line multiplies this number (the number of symbols) by the bigram entropy of the source (calculated in Part I, and still stored in the variable `H`), which is measured in bits per symbol. This gives us the estimated total number of binary digits needed by an optimal encoder to encode the file. We divide by 8 and 1024 to get the estimated file size in kilobytes.

Then enter the following code, which gets the contents from the file, encodes it, and displays the number of binary digits required to encode the contents using our bigram encoder:

```
''' Encode Text to Find Actual Encoded Filesize '''
bits = huff.encode(open(filename, "r").read().rstrip())
print "Actual encoded filesize:", (len(bits) / 8) / 1024, "KB"
```

The variable `bits` contains the actual encoded text, as a string of binary digits. We take its length, divided by 8 and 1024, to find the number of kilobytes required to encode the file.

5 Questions

5.1 Part I: Entropy Estimates

1. What is the estimated entropy, in bits per symbol, for the text in the unigram case?
2. What is the estimated entropy for the bigram case?

5.2 Part II: Huffman Encoding

1. What is the average number of binary digits per symbol using bigram Huffman encoding?
2. How does this number differ from the bigram entropy of English text? (In other words, how many more bits per symbol does the Huffman code require than the bigram entropy?) HINT: If everything was done correctly, there should be less than one binary digit per symbol difference.
3. What is the estimated encoded file size? What is the actual encoded file size?
4. What is the ratio of the original file size (roughly 633 kilobytes) to the actual encoded file size?
5. Assume that the original file used eight bits for each character, and your bigram Huffman encoder uses an average of n binary digits per character, as found above. What is the ratio of this number n to the eight bits used in the uncompressed file? How well does this ratio accord to the ratio of original file size to compressed file size?