Source Coding HW01

Program to compute Entropy and Huffman Codes for given probability distributions.
Author: Siffi Singh
Student ID: 0660819
Dated: 24/10/2017

The file HW01_Sc.cpp contains the C++ code for computing the Entropy and Huffman Codes for given probability distributions.

Here in this report, I will be explaining:

I. Steps to run this file
II. Explanation of Code Logic
III. Results
IV. Inferences from result
V. Conclusion

I. Steps to run the file:

---

1. Keep the iid.dat and markov.dat in the same folder as the SCHW01.cpp file.

2. *Enter the **name of the file you wish to run:***
   - IID.dat
   - MARKOV.dat

3. As you run, you see the Entropy values being printed first, which have specific details about which case (e.g. N=1, N=2, N=4, N=8) is it printing the entropy value.

4. Also if you want to see the intermediate entropy values, you can simply Uncomment the printing the frequency and probability values portion.

5. Secondly, you can see the Huffman codes being printed for case N=2, where each probability value (say, 00, 01, 10, 11) have been associated with an identifier ('1', '2', '3', '4') for better understanding of which node has been assigned which Huffman code.

6. Again after a line separation, you will see the Huffman codes being printed for case N=4, and for N=8 case respectively.

7. For case N=8, where the no. of different probabilities is 256, we cannot use values from '1' to '256' to know which node has been assigned what Huffman code.

8. Therefore, running this C++ code is pretty smooth, all the user needs to do is to enter the name of the file that it wants to use, when asked at run time.

II. Explanation of code logic

---

1. As explained above, in the beginning the user inputs the name of the .dat file to be opened.

2. This file is then read using ifstream function, which provides us with. eof(), .read(), and .is_open() functions. With the help of which we start reading from the file, character by character.

3. After reading it in char, we convert it to binary format for further computations. This converted binary is stored in an array of 8 ints, and as we proceed, we also keep calculating the P(0) and P(1) and p(00), p(01), p(10), p(11) frequencies.

4. Once the length of the read number of bits reached four, we store it a different array of 4 ints, and pass it to the function compare4() for counting the frequencies of the values occurred.

5. Similarly, after 8 characters are read, we store it again into a different array of 8 ints and call a function compare8() to calculate the frequencies of the values occurred.

6. We make use of functions like sprintf, to convert array of int to a string value so that the comparison and the calculation of frequency
can be made easier.

Part 0: Basics

7. After all the frequencies of unique values have been computed, we calculate the probabilities for each case of N=1, N=2, N=4 and N=8.

8. Just after the step of calculating the probability values, we calculate the Entropy values, the entropy value is calculated using the formula:

E(X)= -sum of(p(i)*log(p(i)));

9. The respective values of entropy for cases of N=1, N=2, N=4 and N=8 are displayed for the file entered.

Part 1: Huffman Coding

10. There are mainly two major parts in Huffman Coding:

1) Build a Huffman Tree from input characters.

2) Traverse the Huffman Tree and assign codes to characters. Now, Huffman codes for the specific cases of N=1, N=2, N=4 and N=8 are considered. Now the Huffman codes function is called by passing three arguments, the charter array that contains the list of identifiers for nodes. Second is the float array of probabilities, and third is the size of the array being passed.

11. As we call this Huffman code function, we start building the process of creating the huffman tree. Our main aim is to create the huffman tree efficiently and then traverse it to print the huffman codes. We create a struct of Min Heap that contains A Min Heap: Collection of min heap (or Huffman tree) nodes. This struct contains Current size of min heap, capacity of min heap and Array of min heap node pointers.

12. For doing so, we call the HuffmanCodes function which takes in an argument as the char array of identifiers, float array of frequencies and size of the array being passed. This function returns a struct node. Our approach is to create a minimum heap of capacity equal to size. Initially, there are nodes equal to size. We Iterate while size of heap doesn't become 1. We then Extract the two minimum frequency items from min heap and create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the two extracted node as left

and right children of this new node. Add this node to the min heap '$' is a special value for internal nodes, not used. The remaining node is the root node and the tree is complete.

13. After the huffman tree is constructed using the above method, we Print Huffman codes using the Huffman tree built above, by calling function printCodes(), that takes in four arguments, struct pointer for root node, int of array(frequencies), the top value, and the size of the array.

14. In this function, we Assign 0 to left edge and recur, and Assign 1 to right edge and recur. Assign 1 to right edge and recur characters, print the character and its code from arr[].

15. As we create the minHeap, we want to extract the minimum node from the heap, for choosing the two minimum nodes and creating a node and making it as their children. A standard function to extract minimum value node from heap.

16. The main task is to check whether or not the node is a leaf. it is a small utility function to check if this node passed as an argument is leaf to further heapify the built Huffman Tree.

17. Hence, by passing the float array of frequencies, with the list of identifiers and size, we print the huffman codes for the various nodes constructed and calculated after constructing the huffman tree and extracting minimum and inserting new nodes, and assigning '0' and '1' values to it.

18. Hence, for the output, we get the identifier's name(for the node), followed by the Huffman code corresponding to it.

III. Results

Part 0: Basics

Following is the probability distribution functions for N=1, N =2, N=4 and N=8 block size. The entropy values and the distribution follows:

For N = 1

IID: H(x) = 0.693104
~ 1 bits
P(0) = 0.502563
P(1) = 0.497537

MARKOV:
H(x) = 0.693075
~ 1 bits
P(0) = 0.495488
P(1) = 0.504613

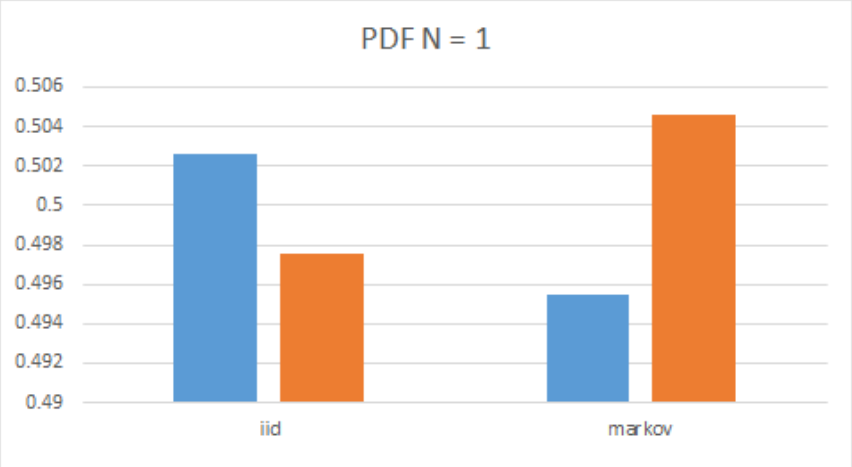

Figure 1: PDF for iid and markov for N = 1

For N = 2
IID: H(x) = 1.938614
~ 2 bits
P(00) = 0.5083
P(01) = 0.49685
P(11) = 0.496825
P(10) = 0.496825

MARKOV: H(x) = 0.834937
~ 1 bits
P(00) = 0.8439
P(01) = 0.147075
P(11) = 0.147075
P(10) = 0.86215

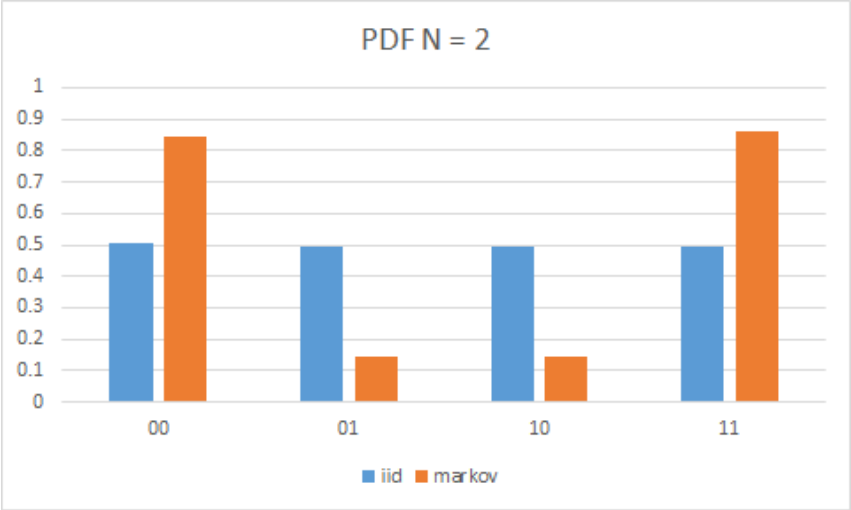

Figure 2: PDF for iid and markov for N = 2

For N = 4:

IID:
H(x) = 2.77241
~ 3 bits

MARKOV:
H(x) = 1.6676
~ **2 Bits**

The respective probability values are given in Table3.

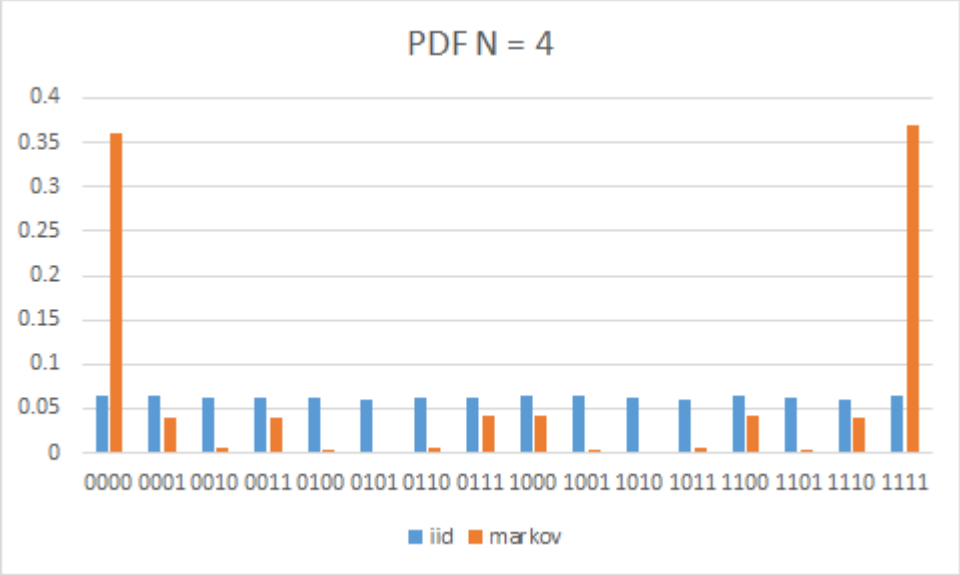

Figure 3: PDF for iid and markov for N = 4

Based on the observation for N=1, we know that there is not any significant difference regarding the frequencies of zero and one outcome between both datasets. The i.i.d dataset and Markov dataset has a similar number of times where the outcome is zero and one. Without any further knowledge, we might say that both of those dataset are the same and behave similarly to a fair coin.

Based on the observation N=2, now we know that there's now a noticeable difference in the probability distribution between the i.i.d dataset and Markov dataset. The i.i.d dataset is behaving like a uniform distribution. Meanwhile, the Markov dataset is not a uniform

distribution. We can see this by looking at Markov's PDF. It has a higher probability of having 00 and 11 as one of the outcomes rather than 01 and 10. On the other hand, the i.i.d dataset has an almost equal distribution for its entire outcome.

In both observation N=4 and N=8, we can fully confirm that the distribution between those two dataset is not the same. The i.i.d dataset forms a uniform distribution where the probability of each categories is the same. Hence, the randomness of the i.i.d dataset's outcome is high. This means the encoder cannot easily predict the outcome of the i.i.d dataset because there's an equal amount of probability for each outcomes. Meanwhile, the Markov dataset has a different probability for each categories. Thus, the encoder can make a better prediction for the next outcome with the Markov dataset than the i.i.d dataset.

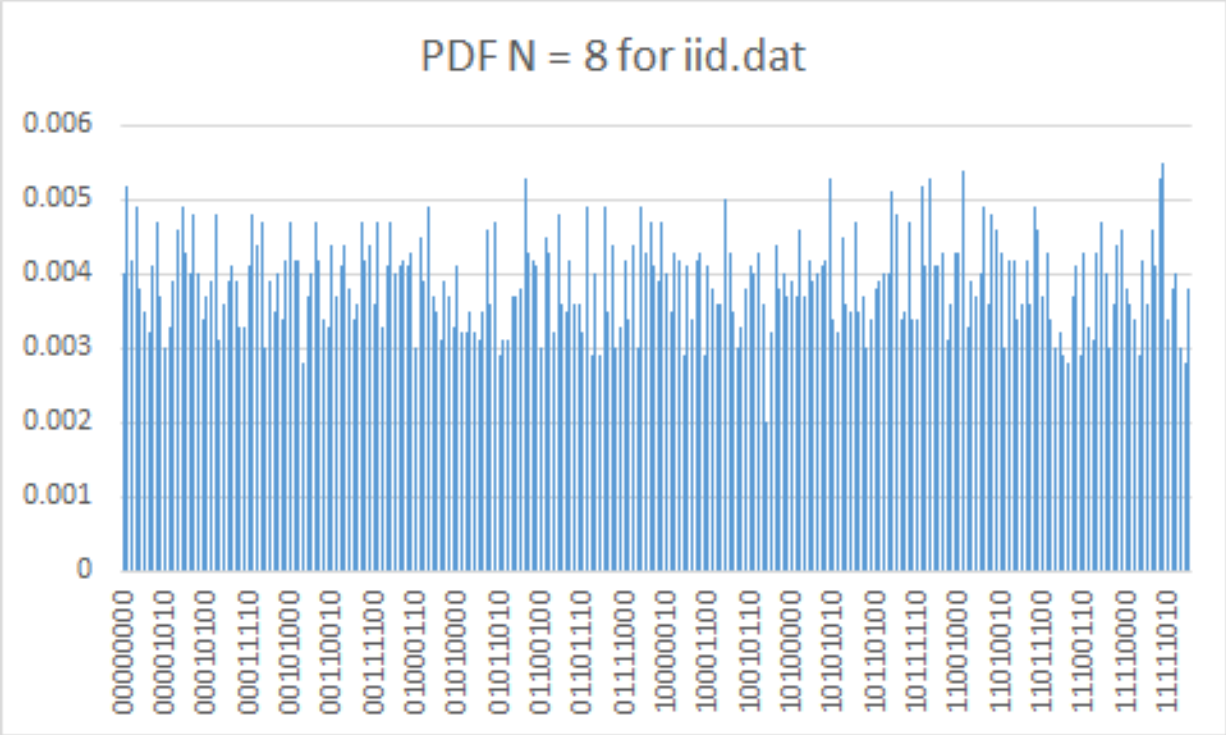

Figure 4: PDF for iid N = 8

IID: H(x) = 6.53319 ~ **7 bits**

In reality, the Markov dataset has a different probability distribution than the i.i.d dataset because the Markov dataset is a not an independent distribution. This dependence on the previous outcome can also decrease/increase the existing probability for the next outcome. This can be seen during observation 1 and 2. We can proof Markov's dependence and i.i.d's independence by this equation.

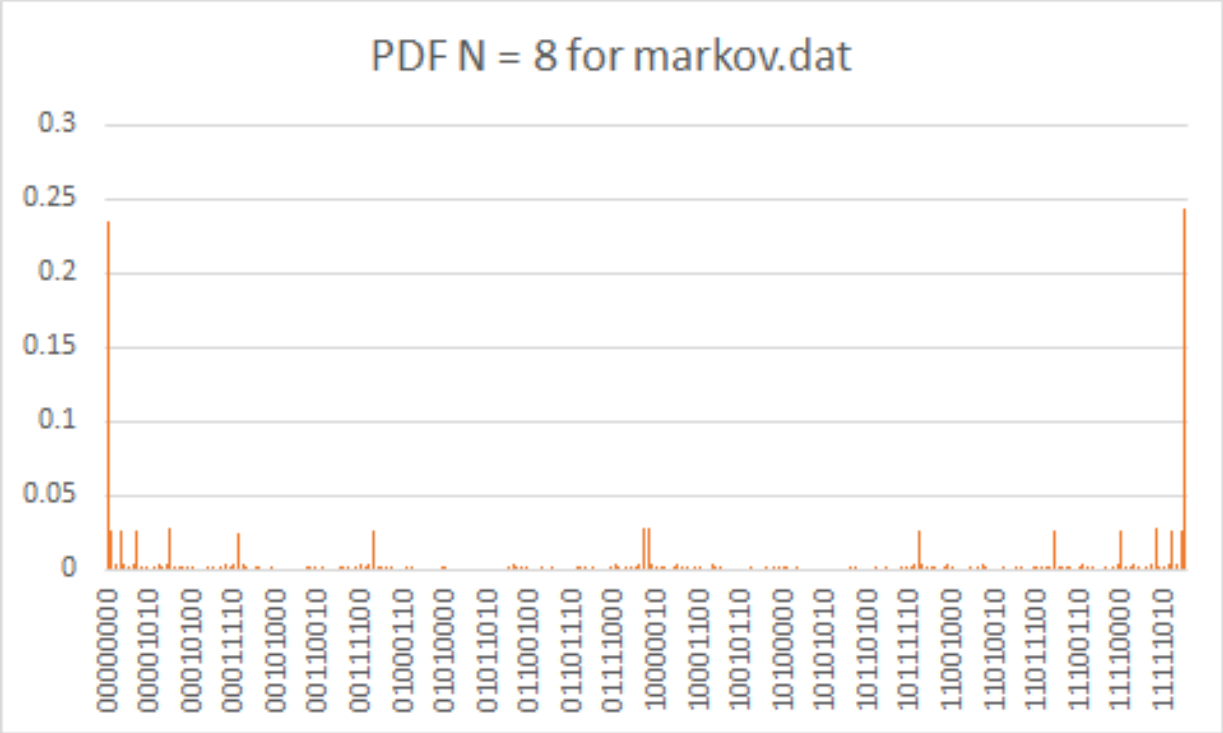

Figure 5: PDF for Markov for N = 8

MARKOV: H(x) = 3.95093 ~ **4 bits**

ENTROPY				
	N=1	N=2	N=4	N=8
iid	0.6931	1.3861	2.77241	6.5332
markov	0.6931	0.8349	1.6676	3.9509

Figure 6: Table for entropy for iid and markov

Hence, the entropy values are given for each dataset in the table above. The comparison and inference has been explained in the section Inferences from result.

Part 1: Huffman Coding

IID for N=2		
CODE	MAPPED CODE	PROBABILITY
00	11	0.5083
01	01	0.49685
10	00	0.496825
11	10	0.496825

Figure 7: Table for Huffman Code for iid for N = 2

IID for N=4		
CODE	MAPPED CODE	PROBABILITY
0000	1111	0.06515
0001	1101	0.0645
0010	0100	0.06165
0011	0011	0.0615
0100	1000	0.0621
0101	0000	0.05935
0110	0111	0.0621
0111	0101	0.0619
1000	1100	0.06425
1001	1110	0.0649
1010	0110	0.06195
1011	0001	0.06005
1100	1010	0.06365
1101	1001	0.0627
1110	0010	0.0606
1111	1011	0.06375

Figure 8: Table for Huffman Code for iid for N = 4

MARKOV for N=2		
CODE	MAPPED CODE	PROBABILITY
00	11	0.8439
01	100	0.147075
10	101	0.147075
11	0	0.86215

Figure 9: Table for Huffman Code for markov for N = 2

To measure the randomness of each dataset, we can use the entropy formula. Entropy can also be used as the minimum representation of source in bits.

A high entropy means a high amount of randomness in the data. Thus, if we are representing a high entropy dataset, it need higher amount of bits than the ones with a low entropy.

This entropy logic follows an intuition in which, when we are compressing some data, we want to assign a lower amount of bits to a more frequent outcome. That way, we can reduce some bits. However, if the outcomes we are representing has an equal probability to be in the data, it is hard to save bits. The worst case is that we cannot save any bits at all.

MARKOV for N=4		
CODE	MAPPED CODE	PROBABILITY
0000	11	0.3602
0001	10011	0.0386
0010	1001001	0.0054
0011	10100	0.0388
0100	10010100	0.00405
0101	1001010100	0.0002
0110	1001000	0.005
0111	10111	0.0417
1000	1000	0.04235
1001	100101011	0.00355
1010	1001010101	0.00045
1011	10010111	0.00495
1100	10110	0.0412
1101	10010110	0.0045
1110	10101	0.0404
1111	0	0.36875

Figure 10: Table for Huffman Code for markov for N = 2

We can see how the result of generated codebooks reflected the entropy value analysis. Both of the i.i.d codebooks have the same bit length as the original mapped input. On the other hand, both the Markov codebooks have assigned different mapped bit length according to its probability. Huffman Coding assigns fewer bit to more probable outcome with the loss of adding more bits to less probable outcome. This result can be more clearly in the Markov N=4 codebook, where the mapped bit length of 0101 is 10 bits but its probability to show in the data is only less than 0.02%.

IV. Inferences from result:

---

1. From the probability values, we can see that the IID dataset is an independent distribution, as it fulfills the requirement for independent events, while the Markov dataset is not an independent dataset.

2. Huffman coding is a lossless data compression algorithm. We assign variable length codes to the probabilities computed from the .dat file read. The most frequent binary block size gets the smallest code and the least frequent binary block size gets the largest code.

3. The variable-length codes assigned to input characters are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character.

4. Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

V. Conclusion:

---

1. Huffman's algorithm is an example of a greedy algorithm. In general, greedy algorithms use small grained, or local minimal/maximal choices in attempt to result in a global minimum/maximum.

2. At each step, the algorithm makes the near choice that appears to lead toward the goal in the long-term.

3. In this case, the insight for its strategy is that combining the two smallest nodes makes both of those character encodings one bit longer (because of the added parent node above them) but given these are the rarest characters, it is a better choice to assign them longer bit patterns than the more frequent characters.

4. The Huffman strategy does, in fact, lead to an overall optimal character encoding. Even when a greedy strategy may not result in the overall best result, it still can be used to approximate when the true optimal solution requires an exhaustive or expensive traversal.

5. In a time or space constrained situation, we might be willing to accept the quick and easy-to-determine greedy solution as an approximation.