Information Theory ECE 231A — UCLA Fall quarter 2022-23
Prof. Suhas Diggavi — Handout # 10, Monday, 28th October 2022

Project Module # 1

Due 11th November 2022, before 11:59pm.

Submit your solution to Gradescope with Entry Code:**57DN5B**

Instructions

Please read the following instructions carefully before working in the project:

- The project must be done in groups. We have made the group allotments randomly and you can view your group in the information sheet posted on Campuswire.
- We will be making use of Jupyter notebooks to modularize the code. Please find the install instructions at the webpage:

  `https://jupyter.org/install`

  You may also find useful the following resources, which provide tutorials on how to use Jupyter notebooks and the python `numpy` library.

  `https://www.codecademy.com/article/how-to-use-jupyter-notebooks`

  `https://cs231n.github.io/python-numpy-tutorial/`

  `https://numpy.org/doc/stable/user/quickstart.html`

- The uploaded module has a directory `code` with the following structure:

```
code
├── Toy_Example_Arithmetic.txt
├── Toy_Example_Huffman.txt
├── Toy_Example_Lempel Ziv.txt
├── Data_Compression_Project_Arithmetic.ipynb
├── Data_Compression_Project_Huffman.ipynb
└── Data_Compression_Project_LZ.ipynb
```

You will only need to make changes to the main `Data_Compression_Project_Arithmetic.ipynb`, `Data_Compression_Project_Huffman.ipynb`, `Data_Compression_Project_LZ.ipynb` notebooks. You do not have to write commands to load any data as that has already been done in the provided file.

- **Important:** Any portions of the code that you must modify start with `YOUR CODE HERE` and end with `END YOUR CODE HERE`. Please do not change any code outside of these blocks.
- Submit your solution of any theory part (if applicable) as well as the code in a single report in a PDF format to Gradescope. You may use the option to convert the Juypter Notebook to a PDF for your code. If needed, you can add Markdown blocks to the Jupyter Notebook to answer any questions. Each group member must upload the prepared final PDF file individually to Gradescope.

- Include a small section towards the end of your report on the contributions of each group member in the project.
- Your codes will be tested using a different input-text. Therefore, make sure you implement the missing parts of the codes properly independent of the given input-texts.
- We **DO NOT** measure the efficiency of your code. We want a working code that can work on a general text.

1 Huffman Codes: Coding Problem

In this part, you will work to implement the Huffman code using Python. Our data input is a text file Toy_Example that contains a simple text to check your code. First, we read the text file as a string. Print the string text to see what is written.

(a) Complete the function compute_distribution that takes a string as an input and returns a list symbols in the following form. Suppose, we have a string with a ternary alphabet $\{A, B, C\}$ having 10 characters, where A appears two times, B appears three times, and C appears five times, i.e., the empirical distribution is $\{0.2, 0.3, 0.5\}$. Then the list symbols is given by symbols $= [('A', 0.2), ('B', 0.3), ('C', 0.5)]$, where each element in the list is a tuple $(Char, prob)$ such that the first element in the tuple is the character and the second element is its corresponding probability. Check your function on the string text and print the empirical distribution of it.

(b) Sort your list symbols in an increasing order according to their probabilities, and print the sorted list. **Hint:** you can use command sorted.

The next step to implement the Huffman code is to encode the Huffman tree. Consider $\mathcal{D}$-ary Huffman code construction. We start with the entire set of symbols symbols $= [(A, 0.1), (B, 0.1), (C, 0.1), (D, 0.3), (E, 0.4)]$ sorted according to their probabilities. Then, we combine the first $\mathcal{D}$ elements in the list that have the smallest probabilities to get another list of tuples. Furthermore, the first tuple in this list is not a character, but, it is a group of $\mathcal{D}$ elements as a single element with the sum of the probabilities of $\mathcal{D}$ elements. We repeat this process again by sorting the list in an increasing order according to their probabilities and combine the first $\mathcal{D}$ elements in the list that have the smallest probabilities to get another list of tuples. Observe that this process ends when the list has a single element. For example, take $\mathcal{D} = 3$ for the set of symbols symbols. In the first layer of ternary Huffman coding we obtain $\text{tree}_1 = [((A, B, C), 0.3), (D, 0.3), (E, 0.4)]$. Then we combine this tuple $((A, B, C), 0.3)$ with the other two elements. Finally, the list will be $\text{tree}_2 = [((((A, B, C), D, E), 1)]$.

(a) Complete the function Huffman_tree that takes the list of symbols and an integer $\mathcal{D}$ as an input and returns a list of a single element containing $\mathcal{D}$-ary Huffman tree. Test your function on the source symbols and print the output.

The last step in the $\mathcal{D}$-ary Huffman code is to convert the constructed Huffman tree into codes as follows. We write a function Huffman_coding that takes a tuple and a code as inputs and returns a dictionary containing the Huffman codes. We call the function Huffman_coding recursively.

(a) Complete the function Huffman_coding, test your function using the Huffman tree, and print the Huffman codes.

(b) Complete the function compute_expected_length that takes a list symbols and a dictionary Huffman_code as inputs and returns the expected length of the Huffman code. Test your code and print the expected length of the provided text.

(c) Complete the function encode_text that takes a string text and a dictionary Huffman_code as inputs and returns the code of the string text. Test your code and print the code of the provided text.

(d) Complete the function decode_text that takes a string txt_code, a dictionary Huffman_code, and a list symbols as inputs and returns the decoded text as a string. Test your code and print the decoded text. Compare it with the original text.

2 Huffman codes: Probabilistic individual letter guarantees

We have seen in the lectures that the Huffman code is optimal in the long run in the sense that on average the length of the Huffman code is minimal as compared to any other uniquely decodable code. Now, in this part of the project, you will show that the Huffman code is optimal in the short run as well in the sense that Huffman code guarantees higher probability for minimal length for individual letters of the source alphabet as compared to any other uniquely decodable code.

Problem 1 (Dyadic source distribution)

Consider a source $\mathcal{X}$ with finite alphabet size that has the probability mass function $p(x) > 0, \forall x \in \mathcal{X}$. Suppose, $p(x)$ is dyadic, i.e., $-\log_{\mathcal{D}}(p(x))$ is an integer for each $x \in \mathcal{X}$. Then show that the $\mathcal{D}$-ary Huffman code length assignment to each source alphabet

$$l(x) = \log_{\mathcal{D}}\left(\frac{1}{p(x)}\right)$$

is shorter than any other uniquely decodable code assignment length, $l'(x)$, more often than not. Mathematically, show that

$$\mathbb{P}\left[l(x) < l'(x)\right] > \mathbb{P}\left[l(x) > l'(x)\right]$$

Hint: The following identity might be useful

$$sign(t) \leq \mathcal{D}^t - 1, \quad t = 0, \pm 1, \pm 2, \ldots \text{and} \mathcal{D} = 2, \pm 3, \pm 4, \ldots$$

where,

$$sign(t) = \begin{cases} 1, & t > 0 \\ 0, & t = 0 \\ -1, & t < 0. \end{cases}$$

Problem 2 (Non-Dyadic Source Distribution)

Now, consider the source $\mathcal{X}$ with probability mass function $p(x) > 0, \forall x \in \mathcal{X}$ which is not dyadic. In this part of the problem, you will show (using steps in the following sub-problems) that the Shannon code length

$$l(x) = \left\lceil \log_{\mathcal{D}}\left(\frac{1}{p(x)}\right)\right\rceil$$

is shorter (up to one bit) than any other uniquely decodable code most of the time.

(a) If $l^*(x) = -\log_{\mathcal{D}}(p(x))$ (where $p(x)$ may or may not be dyadic) and $l'(x)$ is the code length for any other uniquely decodable code then prove that

$$\mathbb{P}\left[l'(x) \leq l^*(x) - \gamma\right] \leq |\mathcal{D}|^{-\gamma}$$

Hint: You can represent $l'(x) = -\log_{\mathcal{D}}(q(x))$ for an arbitrary uniquely decodable code that is drawn from some probability mass function $q(x)$.

(b) Using the proof in part (a), show that the Shannon code most of the time provides a code length

$$l(x) = \left\lceil \log_{\mathcal{D}} \left(\frac{1}{p(x)} \right) \right\rceil$$

that is shorter (up to one bit) than any other uniquely decodable code length $l'(x)$, i.e.,

$$\mathbb{P}\left[l(x) > l'(x) + 1\right] \leq \mathbb{P}\left[l(x) \leq l'(x) + 1\right]$$

(c) **[Bonus]** Let $l'(x)$ be the length of the code associated with the symbol x by an arbitrary uniquely decodable code. Let $l(x)$ be the Huffman code length for the same symbol x. Then, prove that

$$\mathbb{P}\left[l'(x) < l(x) - 1\right] < \mathbb{P}\left[l'(x) > l(x) - 1\right]$$

Note that the source probability mass function is not known to be dyadic in this case.

3 Lempel-Ziv Algorithm: Coding problem

In this part, you will work to implement the Lempel-Ziv Algorithm using Python. Our data input is a text file Toy_Example that contains a simple text to check your code. First, we read the text file as a string. Print the string text to see what is written.

(a) Complete the function compute_distribution that takes a string as an input and returns the entropy of the source text and a list symbols in the following form. Suppose, we have a string with a ternary alphabet $\{A, B, C\}$ having 10 characters, where A appears two times, B appears three times, and C appears five times, i.e., the empirical distribution is $\{0.2, 0.3, 0.5\}$. Then the list symbols is given by symbols $= [('A', 0.2), ('B', 0.3), ('C', 0.5)]$, where each element in the list is a tuple $(Char, prob)$ such that the first element in the tuple is the character and the second element is its corresponding probability. Check you function on the string text and print the empirical distribution of it. The entropy is computed by $H(\mathbf{p}) = -\sum p(i) \log_2(p(i))$.

(b) Sort your list symbols in an increasing order according to their probabilities, and print the sorted list. **Hint:** you can use command sorted.

(c) Complete the function intialize_dict that takes a a list symbols as an input and returns the initialized dictionary of both the sender and receiver. The dictionaries of both sender and receiver are initialized by adding the single characters in the list symbols and assign a code for each character as follows. Suppose that the list symbols has three characters A, B, C. Then, the sender dictionary should contains: $\{A : 00\}, \{B : 01\}, \{C : 10\}$, i.e., the first character A takes the binary representation of index 0, the second character B takes the binary representation of index 1, and the third character C takes the binary representation of index 2. In general, if there are n characters, assign for each character a code for index from $\{0, \ldots, n-1\}$.
Make sure that the length of each code is equal to $\lceil \log_2(n) \rceil$ if there are n characters.

(d) Complete the function Lempel_ziv_coding that takes a string text and the initialized dictionary TX_dictionary as inputs and returns the code of the string text and the dictionary of the sender after encoding the text. The encoding algorithm is as follows: Suppose that we compress the text until the $i-1$th character 1) Find the longest phrase $W = C_i \ldots C_{i+k}$ in the current text that is available in the sender dictionary TX_dictionary. 2) Add the code of W to the output code. 3) Add the code of the next character C_{i+k+1} to the output code. 4) Add the new string $WC_{i+k+1} = C_i \ldots C_{i+k+1}$ to the sender dictionary TX_dictionary with the binary code of the least available index (For example, if the dictionary has n elements, then the new string will have the binary representation of the index n). 5) Remove the string WC_{i+k+1} from the text and go to step 1.
Suppose that the dictionary has n encoded strings, then make sure that when adding a new string, the length of each code in the TX_dictionary is equal to $\lceil \log_2(n+1) \rceil$. See Example 1 for more details.

(e) Complete the function compute_expected_length that takes a string text and its corresponding code as inputs and returns the number of bits per character for the Lempel-Ziv algorithm.

(f) Complete the function decode_text that takes a string code and the initialized dictionary RX_dictionary as inputs and returns the decoded text of the input code and the dictionary

of the receiver after decoding the code. The encoding algorithm is as follows: Start with empty string $W =''$. 1) Suppose that the RX_dictionary has n strings, and hence, each code in the RX_dictionary has length $\ell = \lceil \log_2 (n+1) \rceil$. 2) Read ℓ bits from the code. 2) Add its corresponding string from the RX_dictionary to the decoded text 3) Also, add its corresponding string from the RX_dictionary to W. 4) If W is not in the RX_dictionary, add W to the RX_dictionary with the binary code of the least available index, clear $W =''$, and go to step 1. 5)If W is in the RX_dictionary, go to step 2. **Suppose that the dictionary has n encoded strings, then make sure that when adding a new string, the length of each code in the RX_dictionary is equal to** $\lceil \log_2 (n+1) \rceil$. See Example 1 for more details.

Example 1 *(Lempel-Ziv coding) Suppose that we want to send the text $ABBABB$. Then, we first initialize the sender and receiver dictionaries by* $\mathsf{TX\_dictionary} = [\{A :' 0'\}, \{B :' 1'\}]$.

The sender first reads A, and hence, she will add $'0'$ to the code and add $'1'$ to the code for the next symbol B. Therefore, the sender has encoded AB and we get $\mathsf{code} =' 01'$. *Then, the sender will update the* TX_dictionary *by adding the new string AB with code $'10'$ which is the binary representation of the index 2. Furthermore, the sender will update the codes of A and B to have length 2 as follows:* $\mathsf{TX\_dictionary} = [\{A :' 00'\}, \{B :' 01'\}, \{AB :' 10'\}]$. *The sender reads B and adds its corresponding code $'01'$ to the* code *and so on.*

The receiver will read the $\mathsf{code} =' 0101....'$. *Since the* RX_dictionary *has only 2 elements, the receiver reads the first bit of the* code *and adds its corresponding string A to the decoded_text and W. Since $W = A$ is in the* RX_dictionary*, the receiver reads the second bit of the* code *and adds its corresponding string B to the decoded_text and W. Since $W = AB$ is not in the* RX_dictionary*, the receiver adds $W = AB$ to the* RX_dictionary *with code $'10'$ which is the binary representation of the index 2 and clear $W =''$. Furthermore, the receiver will update the codes of A and B to have length 2 as follows:* $\mathsf{RX\_dictionary} = [\{A :' 00'\}, \{B :' 01'\}, \{AB :' 10'\}]$. *Since the* RX_dictionary *has 3 elements, the receiver reads the two bits of the* code *and adds its corresponding string B to the decoded_text and W and so on.*

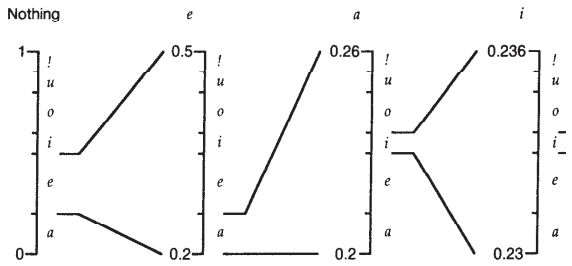

Figure 1: Arithmetic Coding example [**?**]

4 Arithmetic Codes

Consider the random variables X_i with a ternary alphabet $\{A, B, C\}$, having probabilities $\{.2, .3, .5\}$. The source produces a sequence of X_i's independently and identically distributed. As X_i's are i.i.d., let's call the sequence X^n from now on. Imagine that the source emits $ACCB...$ and this sequence is to be encoded using arithmetic coding.

As you observed, the Arithmetic coding requires the cumulative distribution function (CDF) of the source. Therefore, the first step in our algorithm is compute the CDF of a given distribution.

(a) Complete the function `compute_CDF` that takes a list `symbols` as an input and returns a list of tuples of the form $(char, CDF)$.
For example, consider `symbols` $= [(A, 0.2), (B, 0.3), (C, 0.5)]$. We expect the output of the function to be `CDF_symbols` $= [(A, 0.2), (B, 0.5), (C, 1)]$. Test your code and print the CDF of the provided text.

Observe that each symbol (character) has a unique range between 0 and 1 that starts from the CDF of the previous symbol and ends with the CDF of this symbols. For example, the range of the symbol A is $[0, 0.3)$ and the range of symbol B is $[0.2, 0.5)$.

Encoding part: The idea of the Arithmetic coding is to represent the sequence of characters (text) to a unique interval from $[0, 1)$. As the text becomes longer, this interval becomes shorter. Suppose a source with alphabet $\{a, e, i, o, u, !\}$ having distribution $\{0.2, 0.3, 0.1, 0.2, 0.1, 0.1\}$. Thus, the CDF is given by $[(a, 0.2)\,(e, 0.5)\,, (i, 0.6)\,, (o, 0.8)\,, (u, 0.9)\,, (!, 1)]$ The following example is from [**?**]. Suppose, we want to encode a message eai. Let `lower` be the low value of the interval and `upper` be the upper value of the interval. Initially, we set `lower` $= 0$ and `upper` $= 1$. After seeing the symbol e, the encoder narrows the interval to $[0.2, 0.5)$ by setting `lower` $= 0.2$ and `upper` $= 0.5$, where $[0.2, 0.5)$ is the range of the symbol e. After seeing the second symbol a, the new range will be narrowed to the first one-fifth of it, since a has CDF range $[0, 0.2)$. Thus, the new range will be `lower` $= 0.2$ and `upper` $= 0.26$. After seeing the third symbol i, the new range will be `lower` $= 0.23$ and `upper` $= 0.236$ (Please, see Figure 1). The update rule of the `lower` and the `upper` can be summarized as follows. Let `range` $=$ `upper` $-$ `lower` that is initialized by `range` $= 1$. At each step of reading a character, let u be the CDF of this character and l be the CDF of the previous character in the list. Then, the parameters can be updated

as follows:

$$\begin{aligned} \texttt{upper} &= \texttt{lower} + \texttt{range} * u \\ \texttt{lower} &= \texttt{lower} + \texttt{range} * l \end{aligned} \tag{1}$$

(a) Complete the function `decimal_encoding` that takes the `text` to be encoded and the list `CDF_symbols` as inputs and returns `lower` and `upper` values of the interval representing the code of the input text. Test your code and print the value of `lower` and `upper` of the provided text.

Observe that we encode the entire text to an interval between 0 and 1. However, we do not need to send the entire interval to decode the text. Instead, we only need to send the mid-point of the interval. Furthermore, to send a real number, we need infinite number of bits. However, by using Shannon-Fano-Elias code, we only need the first $\left\lceil \log_2(\frac{1}{\texttt{upper}-\texttt{lower}}) \right\rceil + 1$ bits to be uniquely decodable.

(a) Complete the function `Arithmetic_encoding` that takes the `lower` and `upper` intervals as inputs and returns the `code` of the interval. Test your code and print the `code`. How many bits required to encode the provided text?

Decoding part: The first step in the decoding part is to transform the code (a string of zeros and ones) into a real value between 0 and 1. This can be done using binary transformation $\texttt{decoded\_val} = \sum_{i=1}^{C} \texttt{code}(i)2^{-i}$, where C is the length of the `code` and $\texttt{code}(i)$ is the ith bit of the code.

(a) Complete the function `decimal_decoding` that takes the string `code` of zeros and ones as an input and returns the real value `decoded_val` between 0 and 1. Test your code and print the `decoded_value`.

Now, we get a real value between 0 and 1 that represents the mid-point of the interval representing the encoded text. In our example of the encoding part, the decoded value will be 0.233 representing the mid-point of the interval $[0.23, 0.236)$. To decode the text from this value, we reverse the process that we did in the encoding part. Observe that the value 0.233 is greater than the CDF of the symbol a and less than the CDF of the symbol e, i.e., this value 0.233 lies in the range of the symbol e ($[0.2, 0.5)$). Hence, the first symbol must be e. Now, we remove the effect of the decoded symbol e on the value 0.233 by subtracting the CDF of the previous symbol (0.2) and dividing by the range of the symbol e (0.3) to get a value $(0.233 - 0.2)/0.3 = 0.11$. Observe that the value 0.11 is less than the CDF of the symbol a, and hence, the second symbol must be a as 0.11 lies in the range of the symbol a ($[0, 0.2)$). By removing the effect of the decoded symbol a, we get a value $(0.11 - 0)/0.2 = 0.55$ that is greater than the CDF of the symbol e and less than the CDF of the symbol i. Thus, the third symbol must be i. Generally, we start with a real value `decoded_val` between 0 and 1 representing the encoded string. At each step, we return the symbol in which this real value lies with its range. Then, we remove the effect of the decoded symbol as follows. Let u be the CDF of the decoded symbol and l be the CDF of the previous symbol in the list. Then, the `decoded_val` can be updated as follows: let

$$\texttt{decoded\_val} = \frac{\texttt{decoded\_val} - l}{u - l} \tag{2}$$

and repeat the same process until decoding the entire text.

(a) Complete the function `Arithmetic_decode` that takes a real value `decoded_val`, and a list `CDF_symbols` as inputs and returns the decoded text as a string. Test your code and print the decoded text. Compare it with the original text.

5 Universal Data Compression: Coding with a Model class

The universal data compression deals with describing n-length data sequence x from the source alphabets $\mathcal{X}^n$, when the source distribution P is unknown[1]. Now, consider that the source distributions P are, instead of being completely unknown, restricted to a class of parametric distributions

$$\mathcal{M} = \{P(x|\theta) : \theta \in \Theta\}$$

which have corresponding codelengths $-\log(P(x|\theta))$[2]. Note that x is the data to be compressed, and the collection of data compressors is indexed by θ. Let $\hat{\theta} = \hat{\theta}(x)$ be the maximum likelihood estimate of the parameter θ after observing x, i.e.,

$$P(x|\hat{\theta}) = \max_{\theta} P(x|\theta)$$

However, this is not available to us, and without a prior knowledge of $\hat{\theta}(x)$, we do not know how to decode, since we do not know the code design (without knowledge of the distribution to which the encoder is designed). Therefore, we code the data using an arbitrary distribution $Q(x)$. For example, a possible choice of $Q(x)$ could be an empirical distribution after observing x. In this part of the project, we will find the optimal $Q(x)$.

(a) Explain mathematically and in words, what does the following term represent?

$$\log\left(\frac{P(x|\hat{\theta})}{Q(x)}\right) \tag{3}$$

(b) The worst case value of (3) over individual sequences x is

$$\max_{x} \log\left(\frac{P(x|\hat{\theta})}{Q(x)}\right)$$

And we want to minimize the worst case value with an optimal choice of the distribution $Q(x)$, i.e,

$$\min_{Q} \max_{x} \log\left(\frac{P(x|\hat{\theta})}{Q(x)}\right) \tag{4}$$

Then show that

$$Q^*(x) = \frac{P(x|\hat{\theta})}{C}$$
$$C = \sum_{x \in \mathcal{X}} P(x|\hat{\theta}(x))$$

is the optimal solution to (4). Q^* is also known as the normalized maximum likelihood.

(c) **[Bonus]** We are interested in minimizing the worst case value of (3) in part (b), however, one might be interested in minimizing the expected value of (3) instead. With respect to any distribution $P(x|\theta) = P_\theta(x)$ in $\mathcal{M}$, the expected value of (3) is

$$\mathbb{E}_{P_\theta}\left[\log\left(\frac{P(x|\hat{\theta}(x))}{Q(x)}\right)\right] \tag{5}$$

[1] We are denoting the sequences and joint probability distribution of the sequence using simpler notation, without indicating the length of the sequences; this simplifies notation.

[2] We will ignore integer constraints for code-lengths for this problem.

and we want to maximize the expression in (5) with respect to θ and then minimize it with respect to Q, i.e.,

$$\min_{Q} \max_{\theta} \mathbb{E}_{P_\theta} \left[\log \left(\frac{P(x|\hat{\theta}(x))}{Q(x)} \right) \right] \tag{6}$$

Show that (6) is equivalent to

$$\max_{w} \mathbb{E}_{Q^w} \left[\log \left(\frac{P(x|\hat{\theta}(x))}{Q^w(x)} \right) \right]$$

where w is *any* probability distribution[3] on Θ and

$$Q^w(x) = \int P(x|\theta) w(\theta) d\theta,$$

is a mixture distribution.

[3]This distribution has nothing to do with the actual generative model of the source itself. Also we are modeling this as having a probability density function $w(\cdot)$ but it can also be interpreted as a weighting of the model class.