**Module: Information Theory – 1 Credit / 15 hours**
**Competency: Mathematical Reasoning of Data**
Author: Jack Pope

**Instructor overview & objective**
This course module is about Information Theory.

Given that this module is to be worth 1 credit, or 15 hours of coursework, the instructor should prepare three to five subtopic discussions. These should account for at least 5 hours of instructional material and might be in the form of lecture notes or lecture videos. The additional hours should be comprised of student labs or assignments.

Traditionally, a course in Information Theory takes a theoretical approach, on paper. However, the field of Data Science is an empirical endeavor that programmatically emphasizes empirical evidence.

Axioms which we accept as true are what support logical proof. Evidence (data) supports empirical proof, certainty based on a measurement and its statistical significance. A weakness of logical proof occurs when erroneous accepted “facts” support an axiom. A weakness of the empirical proof occurs when the data sample does not represent a population.

To accommodate our purposes, we will use the Python programming language. The choice of programming language is ultimately left to the instructor.

**Schedule**

| Unit | Topic |
|---|---|
| 1 | Overview of Information Theory |
| 2 | Entropy<br>Exercise |
| 3 | Transmission of messages<br>Joint entropy<br>Exercise<br>Conditional entropy<br>Exercise<br>Mutual information<br>Exercise |
| 4 | Noise, Error detection & correction<br>Exercise<br>Channel capacity<br>Exercise |
| 5 | Data compression & Huffman encoding<br>Exercise |

**Overview of Information Theory**

Information theory, also called Communication Theory, is concerned with the processing and transmission of data. Sometimes Information Theory is referred to as Coding Theory, which is actually more focused on the quantification of data. Related studies include Statistical Signal Processing and Digital Communications (digital-to-analog conversion).

Information theory has three general components: a transmitter, a channel and a receiver. These components apply whether we are communicating over amateur radio or by smoke signal.

In studying Information Theory we are interested in two fundamental limits: the maximum amount of data that can be transmitted over a medium (per second) and the degree of compression that can be achieved without a loss of information.

In the formal context of Information Theory, more information does not equate to clarity, but just the opposite. Information Theory seeks to minimize the amount of data required to convey a message.

"Information: the negative reciprocal value of probability." – Claude Shannon

A "message" is a series of symbols drawn from a known set. Within a message, a "symbol" can refer to an alphanumeric character, a word, phrase, object, etc. Usually a symbol is represented by a string of 8 bits or one byte.

Each of the symbols used in a communication have a certain quantity relative to the overall message. For example, we typically expect to use the word "the" more often than we use the word "dinosaur." The word "the" will usually have the higher frequency of use. Usage statistics also apply to characters of the Unicode set, as well as phrases.

Given such expected statistics, methods of data compression can be devised so that fewer bits need to be transmitted. Additionally, symbol statistics will corroborate error correction in messages containing the noise of random information.

Simplistically speaking, given a transmission of eight-bit binary data, there is a frequency for how often a null string such as "00000000" occurs. Lets say that the frequency of the null string is 50% of the data. As a form of compression, to reduce the amount of bits to transmit and store, we can make the null string simply "0," thereby saving 7 bits. We would then pre-append a 1 in the left-most bit of all the 8-bit strings, making them 9 bits in length. The amount of data saved is then:

Total message bits: 8 x (all null bits + all non-null bits), a multiple of 16 == 128 bits

Total message bits after compression: 8 x [(8 bits – 7 bits) + (8 bits + 1 bit)] == 80 bits

That is a savings of 37.5%, given a transmission of 5 bits per symbol; a savings of 3 bits per byte.

**Entropy**
In the context of Information Theory, what is meant by entropy is really “Shannon information,” which is used synonymously as entropy and uncertainty. (We will not address thermodynamic entropy here.)

In general terms, there is “low entropy” if a message is succinct and high entropy when a message is uncertain. Given that more entropy (more Shannon Information) means less certainty, a measure of entropy can guide data compression, as we will see.

As mentioned, symbol frequencies in a message can be used to reduced the number of bits to transmit a message. This requires that both the sender and the receiver share the same symbols statistics. It could be that frequencies are from a single message, or a type of message, or from a sample of prior messages. Perhaps the sender and receiver agree to use a commonly recognized source of word and letter frequencies (such details may be transmitted in advance of the relatively longer communication). For a source see: http://www.norvig.com/mayzner.html . There are also statistics on successive word pairs, trios, etc. For example the letter 't' usually proceeds the letter 'h' in two letter sequences.

There are $26^{26}$ possible letter sequences and, with 97,565 words in then English dictionary, we might consider five million pairs to analyze. However, many of these combinations are not used in normal non-encrypted communication. High frequency words will include conjunctions and propositions. We are interested in the related nouns and verbs.

Lets summarize some statistics based on what we know so far about entropy and the English language. We will assume independence between successive characters and successive words.

English language quantifications:
- Characters (A to Z and blank space): 27
- Words: 97,565
- Average word length: 4.79 chars
- Uniform character distribution: $\log_2(27) == 4.75$ bits
- Uniform word distribution (incl space): $\log_2(97{,}565) / 5.79 == 2.86$ bits

Consider the characters of the English alphabet, with 26 letters and a blank space (a hidden \s character) for separating words. So, we really have 27 characters.

Given that the probability distribution of a message is known, the length of the message can be known. Therefore, for all bits of message S, the sender and receiver have the same information, or $\text{decode}(\text{encode}(S_i)) = S_i$

The computed entropy depends on the basis of symbol probability that is adopted by the sender and receiver.

Minimum bits to represent a symbol:
The essence of Shannon's 1948 “Noiseless Coding Theorem” is that no redundant data be used in a message. Accordingly, the theoretical minimum length of a message is given by the entropy of the message.

<u>The expected length of message is equal to the entropy of the message.</u>

Entropy, also known as Shannon Entropy or Shannon Information, can be expressed as:

$$I = -\log_2 P_0$$

… where $P_0$ is the probability that a message will be received.

Single event:
The Shannon Information I may be expressed for a single event e as inversely related to the probability of the event. That is:

$$I_e = -\log_2 P_e == \log_2 (1/ P_e)$$

You can use either form.

Thus, with 100% certainty,

$$I = -\log_2 1 = 0$$

So, 0 bits of “Shannon information” (Same as 0 bits of Shannon entropy).

Note that logarithms in base 2 count in magnitudes of 2, readily accommodating bits (binary digits). If the values that we are dealing with are in another base, we can still reflect these as bits by computing the amount of information as the log_baseX(someNumber) / log_baseX(2).

Working with distinct 8-bit string lengths, from 00000000 to 11111111, the receiver will not confuse 00000111 with 1110. Binary can also corresponds with some decision trees and networking protocols.

Apart from reflecting binary magnitudes, using a logarithm relegates output to a narrower range of integers.

Consider the decimal value 0.5 in terms of bits:

$$I = -\log(0.5)/\log(2) == -\log_2 0.5 == 1 \text{ bit}$$

Supposing that a message has a 0.5 probability of being sent (like flipping a coin), the receiver cannot be certain which message will be received.

Lets consider a scenario where the transmitter can send any one of 10 possible, equally probable, messages.

$$I = -\log_2 (1/10) = 3.3$$

This is 3.3 bits of Shannon information or Shannon entropy.

If it is known that the symbols 'B' occurs 8 times in 64 symbols, then the number of bits required to represent 'B” is:

$$-\log_2 8/64 == 3 \text{ bits}$$

In the above example, the expected length is 3, which is also its entropy. Using more than 3 bits to represent 'B' suggests opportunity for compression.

Weighted average entropy:
We can define entropy as the average uncertain information received over a range of values or events.

For a set of symbols comprising a message, weighted average entropy may be denoted by the following equation:

$$H = -\sum P_i \log_2 P_i$$

That is, $P_i$ is the probability of a symbol. While all symbol frequency probabilities will sum to 100%, entropy may be computed higher and lower than 1. Entropy of 0 indicates certainty.

Note the probabilities now also serve as weights, as now needed to give relative weight to each parameter:

$$I = -(P_0 \text{ x } \log_2 P_0 + P_1 \text{ x } \log_2 P_1)$$

Given a set of two symbols, with probabilities of 0.60 and 0.40, the entropy is 0.971, indicating entropy or uncertainty. In a spreadsheet, the formula can be entered as:

=-(0.6*LOG(0.6,2)+0.4*LOG(0.4,2))

It would be the same result if 4 of 10 messages share had one expectation while the other 6 shared another expectation of being received. Suppose the message was simply “AAAABBBBBB.” The string is four tenths As and six tenths Bs. The formula is like that in the above scenario:

$$I = -(0.40 \text{ x } \log_2 0.40 + 0.6 \text{ x } \log_2 0.6) == 0.971 \text{ bits of Shannon information}$$

Dealing with unknowns:
We can limit uncertainty by imposing a maximum bound of likelihood of receipt for a set of symbols or messages. We may apply such a bound to a range of symbols within a set of unknown or random information, treating all equally likely. That way, we get results for groups of symbols, like the strings “AAAA” and “BBBBBB” in the example above. We can similarly have categories of symbols with various weights. Symbols of unknown probability can be assigned 0.5 as a neutral weight.

Consider for example that 9 out of 10 messages are equally expected by the receiver. It could be that a “constraint” was imposed on this set of 9. Therefore, there is only a 10% expectation of receiving the one unique message. Now the Shannon entropy is:

$$I = -(0.90 \text{ x } \log_2 0.90 + 0.10 \text{ x } \log_2 0.10) = 0.20$$

This means that there are 0.20 bits of Shannon entropy or uncertainty of 0.20 bits. The constraint effectively limited the amount of information, thereby reducing uncertainty.

If we have only a small set of symbols, the number list [1, 3, 5, 7]. The entropy calculation is:

Element, e from index 1 to 4
Number of elements, N = 4
Probability of e, $P_e = P_e / N$
Shannon entropy, I = -1 * (average sum of $P_e * \log_2(Pi)$ ) == 2 bits

What if the the sample numbers are [2, 4, 6, 8], the result is the same, 2 bits. If you are confused, think though this general algorithm (program code is AWK):

```
bitstr = "1111222233334444"
N = split(bitstr, arr, "")        # create array arr N = length of bitstr

for (i = 1; i <= N; i++) {        # assign value of digit to a key in array arr2
  arr2[arr[i]]++;                 # Note! Duplicates == single key; yet +1 for each digit
}

for(e in arr2) {
  # now we want count of element relative to total element count
  p = arr2[e] / N;                # key e for val = relative count in numerator / total count
  I -=  p * log(p);
}
print I / log(2);
```

In Python the same algorithm can look like this:

```
#!/usr/bin/python3
import math

def entropy(bitstr):
  elist = list(bitstr)
  N = len(elist)
  I = 0.0
  S = set(elist)  # creates set of distinct elements

  for s in S:
    frequency = 0.0

    for e in elist:
      if(s == e):
        frequency = frequency + 1
        p = (1.0 * frequency) / (1.0 * N)
    I = I - p * math.log(p)

  return I / math.log(2)

#Run a test:
bitstr = "1111222233334444"
print(entropy(bitstr))
```

Whatever language you use, when computing $P_e$ be careful not to confuse the count of distinct elements with the elements themselves. If you use an associative array to assist with this task, the manner of extracting a collection's keys and values depends on the language.

**Transmission of messages**
Our message symbols can be comprised of characters or words.

The Shannon information for drawing a specific character, such as 'A':

$$I = -\log_2 (1/27) = 4.76$$

The Shannon information for drawing the three character sequence 'ABC':

$$I = -\log_2 (1/27) + -\log_2 (1/27) + -\log_2 (1/27) == 3 \text{ x } ( -\log_2 (1/27) ) = 14.28$$

That same value of Shannon information exists if we seek 'XYZ' or 'ABC' or whether the characters were received without error or random.

Using large volume for symbol frequency counts:
In reality, a message will have some characters that are more frequent than other characters. Based on the statistics shown earlier, a blank space typically occupies 20% of a word symbol. E is the most used letter, with 12.49% of the total letter count, and Z is the least used letter, with 0.09% usage (Source: http://www.norvig.com/mayzner.html ).

Computing Shannon Information can take these statistics into account. For example, different parameters of the weighted entropy formula can represent each character with a different weight based on its probability. That is:

$$I = -(P_{blank} \text{ x } \log_2 P_{blank} + P_A \text{ x } \log_2 P_A + P_B \text{ x } \log_2 P_B + \ldots + P_Z \text{ x } \log_2 P_Z )$$

Based on the relative parameters of this weighted average, the expectation of drawing any one character, the Shannon information, is 4.16 bits.

Supposing communication of three specific characters, in any order, lets consider the probability of their being drawn. For the characters “HOP” we have:

$$I = -(P_H \text{ x } \log_2 P_H + P_O \text{ x } \log_2 P_O + P_P \text{ x } \log_2 P_P )$$

Or,

$$I = -(0.05 \text{ x } \log_2 0.05 + 0.076 \text{ x } \log_2 0.076 + 0.02 \text{ x } \log_2 0.02 ) = 0.62 \text{ bits}$$

The likelihood of receiving a message “HOP” is not different than receiving “POH.”

In Information Theory we are interested in Shannon entropy given a set of symbols in any order.

If we require exact order, such as H-O-P, then we can look to Probability Theory. As an aside, consider that each character H-O-P has its own probability of occurrence. To receive these characters in that exact sequence is less probable than any one of their probabilities. That overall probability would be:

$$0.05 \times 0.076 \times 0.02 == 0.000076$$

Based on that result, one might compute the Shannon entropy of a single draw of these three characters as:

$$I = -\log_2 0.000076 = 13.68$$

… indicating a high amount of entropy or uncertainty for receiving the exact symbol sequence H-O-P.

However, for Information Theory, whether a symbol sequence is recognizable or random is irrelevant. What matters is the weighted average number of bits. "HOP" has the same information as "POH."

The meaning of information is of no concern to Information Theory. In that context, "more information" does not convey more meaning, but greater uncertainty with a greater number of bits that must be processed.

**Exercise**
Using Python, write an algorithm to determine the entropy of the following lists:

[0, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1]
[1, 0, 1, 0, 1, 0]

Submit your coded algorithm and your output.
What do you deduce about the impact of various element patterns on entropy?

**Joint entropy**
Given two sets of symbols, joint entropy is the entropy of their intersecting set. That is, it is the count of each element within the intersection relative to the total number of intersecting elements.

Formally, joint entropy can be expressed as:

$$H(S1 , S2) = -\sum_i P(S1_i , S2_j) \times \log_2(P(S1_i , S2_j)), \text{ where } (S1_i , S2_j) \in S1 \times S2$$

Determining joint entropy is not much different than finding entropy, except that you must first determine the intersecting set.

Python gives us some fairly easy to use set operators:

```
s1, s2 = {1, 3, 5, 7, 8}, {1, 2, 4, 6, 8}          # define 2 sets
print("union s1 s2: ", s2 | s2)
print("intersection s1 s2: ", s1 & s2)
print("difference s1 s2: ", s1 - s2)
```

However, for you to better understand the set operations involved, the following Python code is more explicit about the set intersection:

```
#!/usr/bin/python3
import math

def jointEntopy(bitstr1, bitstr2):
  S1 = set(bitstr1)
  S2 = set(bitstr2)
  L1 = list(bitstr1)
  L2 = list(bitstr2)

  N = len(L1)
  I = 0.0

  for s1 in S1:
    for s2 in S2:
      frequency = 0.0

      for i in range(N):
        if(s1 == L1[i] and s2 == L2[i]):      # set intersection
          frequency = frequency + 1
      p = (1.0 * frequency) / (1.0 * N)
      if p > 0.0:
        I = I - p * math.log(p)

  return ( I / math.log(2) )

#Run a test:
bitstr1 = "13578"
bitstr2 = "12468"
print( jointEntopy(bitstr1, bitstr2) )
```

**Exercise**
Using Python, write an algorithm to determine the joint entropy of the following lists:

```
          S1                    S2
[0, 0, 0, 0, 0, 0] and [1, 1, 1, 1, 1, 1]
[1, 0, 1, 0, 1, 0] and [0, 1, 0, 1, 0, 1]
[1, 2, 2, 3, 3, 3] and [1, 1, 1, 2, 2, 3]
```

Submit your coded algorithm and your output.
What do you deduce about the impact of any particular bit pattern on joint entropy?
What types of patterns increase or decrease joint entropy?

**Conditional entropy**
There are cases in which one variable depends on another. Coincident signals may be depicted as cause-affect relationships.

The entropy of S2 conditioned on S1 is written as: H(S1 | S2). More formally: If H ( S2 | S1 = s1 ) is the entropy of the discrete random variable S2 conditioned on the discrete random variable S2 taking a certain value s1, then H ( S2 | S1 ) is the result of averaging H ( S2 | S1 = s1 ) over all possible values s1 that S2 may take.

Formally, conditional entropy can be expressed as:

$$H(S1 , S2) = -\sum_i\sum_j P(S1_i , S2_j) \times \log_2(P(S1_i , S2_j)), \text{ where } S1_i \in S1 \text{ and } S2_j \in S2$$

We can account for their joint entropy, comparing it to the entropy of the other, perhaps dependent, set. That is:

$$H(S1 \mid S2) = \text{jointEntropy}(S1, S2) - \text{entropy}(S2)$$

Presented with dependence on another variable we have a state of ambiguity. That is why conditional entropy is also called “equivocation.” In the formula above, S1 may be interpreted as a “cause” while S2 is considered an “effect.” H(S1 | S2) is the signal lost between S1 and S2.

Cause and effect exist when the sender can confirm the receiver's information, or if the receiver can confirm what the sender actually sent. Given uncertainty, where S2 is not likely to reflect a known S1, there might be set independence or channel noise (see below).

In Python, using our algorithms above for entropy and joint entropy, conditional entropy may be described as follows:

```
#!/usr/bin/python3
exec(open("entropy.py").read())
exec(open("jointEntropy.py").read())

def conditionalEntropy(bitstr1, bitstr2):
  return jointEntropy(bitstr1, bitstr2) - entropy(bitstr2)

# Run a test:
bitstr1 = "13578765"
bitstr2 = "12468765"
print( conditionalEntropy(bitstr1, bitstr2) )
```

**Exercise**
Using Python, write an algorithm to determine the conditional entropy of the following lists:

| S1 | | S2 |
|---|---|---|
| [0, 0, 0, 0, 0, 0] | and | [1, 1, 1, 1, 1, 1] |
| [1, 0, 1, 0, 1, 0] | and | [0, 1, 0, 1, 0, 1] |
| [1, 2, 2, 3, 3, 3] | and | [1, 1, 1, 2, 2, 3] |

Submit your coded algorithm and your output.
What do you deduce about the impact of any particular bit pattern on conditional entropy?
What types of patterns increase or decrease conditional entropy?

**Mutual information**
Mutual information can tell us how much uncertainty is removed from one message given (prior) knowledge of another message.

Consider the following formula for mutual information:

$$H(S1 \mid S2) = \text{entropy}(S1) - \text{conditionalEntropy}(S1, S2)$$
$$= \text{entropy}(S1) - \text{jointEntropy}(S1, S2) - \text{entropy}(S2)$$

The mutual information of two sets of symbols gauges in terms of Shannon Information how much dependence set S1 has on S2 and how much dependence set S2 has on S1.

In Python, mutual information might look like this:

```
#!/usr/bin/python3
exec(open("entropy.py").read())
exec(open("conditionalEntropy.py").read())

def mutualInformation(bitstr1, bitstr2):
  return entropy(bitstr1) - conditionalEntropy(bitstr1, bitstr2)

# Run a test:
bitstr1 = "13578765"
bitstr2 = "12468765"
print(mutualInformation(bitstr1, bitstr2))
```

**Exercise**
Using Python, write an algorithm to compute the mutual information for any two sets, S1 and S2. Show what happens when there is redundant data encoding for overcoming symbol errors. What is the implication for redundancy given mutual information?

Submit your coded algorithm and your output.

**Noise, error detection & correction**
Noise amounts to uncertainty. It adds more information, increasing entropy.

Simplistic approaches to dealing with error include sending redundant messages or symbols and using a parity bit.

Ignoring channel capacity, redundant transmission maintains the same probability of drawing any one character from a message.

A message can be sent twice and the two instances of received message can be compare. The information that these messages have in common provides some accepted degree of validation. For example, if there is no error, if some information corroborates that received symbols are correct, the mutual information will be equivalent to the joint entropy of the redundant sets.

Recall that for mutual information, H(S1 | S2) = entropy(S1) - conditionalEntropy(S1, S2). Through substitution, we have H(S1 | S2) = entropy(S1) - jointEntropy(S1, S2) – entropy(S2). If S1== S2, then H(S1 | S2) = jointEntropy(S1, S2).

To reduce the amount of redundant data for overcoming errors, there is another approach: use a "parity bit." That is to send one bit between each message that indicates an odd (0) or even (1) number of bits. However, this is only good for single bit errors.

The parity of a list can be computed as:

$$parity = sum(S_i) \% 2$$

The result can be compared to the received parity bit.

Redundant characters can be transmitted. For example, instead of transmitting "HOP" transmit characters as double characters, such as "HHOOPP."

The receipt of "HHOXPP" clearly has an error. This can trigger a "re-transmit" request, or the receiver may simply substitute the most common word having the pattern "H_P."

A more robust scheme is transmit several instances of each character:

H H H H H H H H
O O O O O O O O
P P P P P P P P

Each character actually has its ASCII code represented in binary, as 0 and 1 bits. The representation of a 1 being in the aforementioned redundant 8 bit scheme. These redundant bits provide "7 bit error correction." For example, a single 1 bit is expressed as:

1 1 1 1 1 1 1

… and a single 0 bit is expressed as:

0 0 0 0 0 0 0

Following each of these bit strings with parity bit, such as 00000000 or 00000001, provides “n-bit error detection.”

The receiver can decode the value S2 = 11111111 as 1 if the sum(S1) > 4 or if the subsequent parity bit is 0.

Additionally, there could be a certain parity for certain vertical sequences of bit strings for “m-bit error detection.”

To a certain extent, the transmitter and receiver must share some a priori information, if not about redundancies and parities, at least about basic communication protocols, such as encoding and modulations. Otherwise, the there will be no receipt of signal, just noise. The medium does not need to know this information.

**Exercise**
Code an algorithm that demonstrates use of redundant symbols for overcoming errors. Test your algorithm and show its output. Does symbol redundancy change affect Shannon entropy? Explain why or why not.

**Channel capacity**
The medium or route connecting a transmitter to a receiver may be called a channel. The channel may carry electrical pulses, representing bits according to a protocol, such as for networking or wireless communication. Bits may represent integers, and integers may represent characters which comprise words of communication.

A channel is limited by how many bits it can carry without signal loss. Suppose a electrically conductive medium maintains eight voltage fluctuations per second (a frequency of 8 Hertz), each with a consistent magnitude. The most data that could be effectively transmitted on this channel is 8 bits/s. The channel's bandwidth is 8 bits/s.

Regardless of the communication protocol, an increase in the rate of throughput will require either a higher rate of voltage fluctuation per second (higher frequency) or transmission on multiple channels in parallel. In either case, a higher rate of transmission requires more energy.

If errors received increase at an increasing rate as the rate of symbol throughput increases, then the probability of reception should be reduced until entropy resides within an acceptable tolerance for the error correction scheme in place.

A communication channel's capacity is the maximum rate for transmitting symbols in bits per second. That is:

Given the transmission of one message per cycle, information rate is defined as:

informationRate(S1, S2) = channelCapacity(S1, S2)

Accordingly, Shannon's Fundamental Theorem for a Noiseless Channel (1948) holds that there is a maximum rate S for symbol transmission based on channel capacity C divided by the entropy H:

$$S = C / H$$

We can consider channel capacity the maximum mutual information and compute it as:

$$channelCapacity(S1, S2) = mutualInformation(S1, S)$$

**Exercise**
Using Python, write an algorithm to determine the channel capacity of the following lists:

| S1 | S2 |
|---|---|
| [0, 0, 0, 0, 0, 0] | [1, 1, 1, 1, 1, 1] |
| [1, 0, 1, 0, 1, 0] | [0, 1, 0, 1, 0, 1] |
| [1, 2, 2, 3, 3, 3] | [1, 1, 1, 2, 2, 3] |

What values of 0s and 1s for S1 and S2 will maximize channel capacity? Does information rate matter?

**Data compression & Huffman encoding**
Data compression is an achievement of information theory. Compressing data allows us to represent information with fewer bits for transmission and storage. The receiver restores the original information using decompression.

Shannon entropy makes compression possible, as it tells us when data is represented with more bits than necessary. By representing a symbol from a message with an optimal number of bits to minimize entropy, the bit sequence representing a message will have no more bits than is required for its unique identity.

Depending on the compression method, more bits can be acceptably compressed than can be restored through decompression. Such “lossy” data compression maybe acceptable for sound and graphics processing. Conversely, archival storage of financial data must be “lossless.” Similarly, data transfer over a network may be either lossless or lossy, depending on the application.

The effectiveness of compression is the size of the compressed data relative to the original data.

We will look at lossless compression in terms of Huffman encoding, a “minimum redundancy coding” which encodes the most frequently used symbols by using the fewest bits. There are other clever techniques that deserve a place in a dedicated course of study on data compression, such as Dynamic Markov compression and LZ77 compression.

Compression:
Huffman coding is an elegant form of compression that decodes and encodes data using a binary tree data structure called a “Huffman tree.”

As an example, suppose we have a text message to send that consists of 120 characters, only made up of the characters A, B, C, X, Y, and Z. These characters have different frequencies of occurrence in the text. So, we itemize their probabilities and entropies as follows:

| Symbol | Probability | Entropy ea | Entropy total |
|---|---|---|---|
| A | 27 / 120 | 0.48 | 13.07 |
| B | 9 / 120 | 0.28 | 2.52 |
| C | 21 / 120 | 0.44 | 9.24 |
| X | 15 / 120 | 0.38 | 5.63 |
| Y | 12 / 120 | 0.33 | 3.99 |
| Z | 16 / 120 | 0.39 | 6.20 |

Building Huffman tree codes result in compression that approximates entropy. It cannot be exact because, as you can see, entropy does not compute to whole numbers.

To compress and decompress, start with one of the symbols, such as 'A.' That will signify the root node of the tree. A left branch from a node will be labeled with a 0 and a right branch will be labeled with a 1.

Order the symbols in terms of frequency:

```
B  Y  X  Z  C  A
9 12 15 16 21 27
```

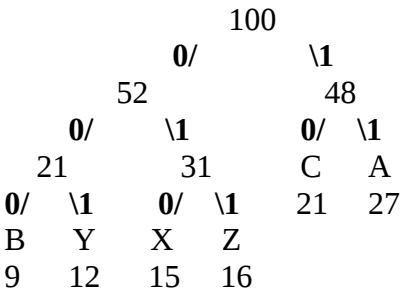
Sum the symbols pair-wise, pairing the symbols with the smallest probabilities, to create sub trees with their respective root nodes as their sums:

```
   21          31          48
  /  \        /  \        /  \
 B    Y      X    Z      C    A
 9    12     15   16     21   27
```

Continue to merge the trees, paring the smallest sums:

```
          52
        /    \
   21          31          48
  /  \        /  \        /  \
 B    Y      X    Z      C    A
 9    12     15   16     21   27
```

Repeat until there is one root. Then label 0s for left branches and 1s for right branches, as follows:




The Huffman code (the compression) for each symbol is found by tracing its path from the root node (top) to the symbol. Along this path make a list of the 0s and 1s. For the above tree, we get:

B = 000
Y = 001
X = 010
Z = 011
C = 10
A = 11

How much compression did this give us? It takes just 2 to 3 bits to represent all the symbols. If the original scheme were to transmit 8 bit symbols, the Huffman encoding, saves 5 to 6 bits per symbol.

| Symbol | Frequency | Bits | Frequency x Bits |
|---|---|---|---|
| A | 27 | 2 | 54 |
| B | 9 | 2 | 18 |
| C | 21 | 3 | 66 |
| X | 15 | 3 | 45 |
| Y | 12 | 3 | 36 |
| Z | 16 | 3 | 48 |
| | | | 267 total bits |

Saved bits = (120 x 8) – 267 / (120 x 8) == 72.2%

Decompression:
Decompressing entails rebuilding the Huffman tree. Consider the following Huffman codes and their associated values:

{ B:000, Y:001, X:010, Z:011, C:10, A:11 }

We can determine the placement of each symbol. Starting at the root node, the route to a symbol depends on the left-right direction signified by 0 or 1 respectively. Label the end of each such bit path with the associated symbol. Repeat this processes for all of the symbols, starting from the same root node.

In decompression, the sequence order of symbols:bits does not matter because each Huffman code is unambiguous, regardless of the bits in its parent node (ie: Huffman codes are prefix *free*).

**Exercise**
Using Python, write an algorithm to decompress the Huffman codes from our example: A:11, C:10, Z:011, X:010, Y:001, B:000. Does your algorithm verify that symbol sequence matters for decompression? Does it matter for compression? Please explain.

Create an additional function to compute the effectiveness of the Huffman encoding.
Submit your test output along with your code.

**Additional references**
https://www.amazon.com/Elements-Information-Theory-Telecommunications-Processing/dp/0471241954