Lab4 - Multimedia Lab Part II

The Multimedia Lab will be a two part lab designed to discuss each aspect of the life-cycle of a multimedia file. This story begins with generation of the file, which is the digitization of some analog phenomenon of interest (e.g. sound -> music, light -> image, both -> film). Once this process is complete, the generated digital media file can be operated on (stored, read, copied, etc.) by a finite precision computer. Each of these operations, however, would be quite a bit faster if the file were small, so we'll investigate compressing the file into a more compact representation. As the honorable Snoop Dogg knew well, it "ain't no fun if the homies can't have none," so you'll naturally want to share this media file with your friends. We'll look at the complications that may arise in transmission, and investigate techniques to ensure that your friend recieves the file error-free. Sadly, all good things must come to an end, and if your file is stored on some physical medium, it will eventually be corrupted.

Naturally, the process of capturing, storing, transmitting, and eventually deleting a media file is composed of a harmony of instruments across **all** EECS disciplines, but this lab will focus in particular on the role of probabilistic analysis, reasoning, and algorithm design in this symphony. Probability in EECS does not exist in a vacuum but, as you will see in this lab, often forms the theoretical underpinning to analyze the efficacy of an idea.

This notebook covers Part 2 of the lab, in which we'll recap some of the concepts from compression and continue to discuss the transmission of a file. Hope you enjoy the lab!

Recap: Source Coding and Compression

In the last lab, we looked at the Huffman compression algorithm as applied to a tweet, so let's revisit that. In this iteration, we'll be applying a Huffman code to a song. In fact, the last step in encoding audio into an mp3 file is to apply a Huffman code, so this is not an impractical application. Walk through the following code blocks (which use function imported from the file `utils.py`, so make sure it is in the same directory as this notebook). The code will compress the song `School Bell Ringing`. Each code block has a comment describing its functionality at the top.

Imports:

In [ ]:
```
from utils import *
from __future__ import division
from scipy.io import wavfile
from IPython.core.display import HTML
from IPython.display import display
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

Generate a quantized version of `School Bell Ringing`:

In [ ]:
```
num_of_bits = 4 # number of bits used by the quantizer
```

```
time, quantized_song = generate_quantized_song(num_of_bits) # generate song qu
antized with num_of_bits
fs = 3000 # I want the song to playback faster an with a higher pitch, so I'm
increasing playback sampling frequency
```

Visualize the first 0.01 seconds of the song and play it:

In []:
```
plt.plot(time[:120], quantized_song[:120]) # Plot the first 0.01 second
plt.title("First 0.01 seconds of 'School Bell Ringing' Quantized")
playMusic('sbr_' + str(num_of_bits) + '_bit_quantized', fs, quantized_song)
```

Plot a histogram of song values (i.e. how often each of the \(2^{num_of_bits}\) points occur) and use this as a frequency dictionary to encode the song:

In []:
```
# Plot Histogram
plt.hist(quantized_song, 2**num_of_bits)
plt.title("Historgram of Song Values")

# Set frequency dictionary values based on histogram
L = 2**num_of_bits
hist, bins = np.histogram(quantized_song, L)
quantized_pt = -1
freq_dict = {}
for value in hist:
    freq_dict[ quantized_pt ] = value
    quantized_pt += 2 / (L-1)
```

Encode the song and compare the differences in file size:

In []:
```
huff = HuffEncode(freq_dict)
print "Symbol\tWeight\tHuffman Code"
total_num_of_bits = 0
for p in huff:
    print "%.3f\t%s\t%s" % (p, freq_dict[p], huff[p])
    total_num_of_bits += freq_dict[p] * len(huff[p])

encoded = encode_song(quantized_song, huff)
print 'First 50 bits of huffman coded song file: ',  encoded[:50]

original_num_of_bits = num_of_bits * quantized_song.size
original_size = round(original_num_of_bits/1000/8) # original file size
print "Original file size with a %d-bit quantizer: %d KB" % (num_of_bits, orig
inal_size)
print "Compressed file size with a %d-bit quantizer: %d KB" % (num_of_bits, ro
und(total_num_of_bits/1000/8))
print "Wow! Look at that compression!"
```

Optimality of Huffman Codes

In Claude Shannon's landmark paper `A Mathematical Theory of Communications`, he established the theoretical limit to possible data compression of a stream of I.I.D random variables. This is known as Shannon's source coding theorem (http://en.wikipedia.org/wiki/Shannon's_source_coding_theorem), which states that an I.I.D. sequence of random variables $X_1, X_2,...,X_N$ can at best be expected to be represented by $NH(X)$ bits, where $H(X)$ is the binary entropy function

which states that given a sequence of I.I.D random variables $X_1, X_2,...,X_N$, the entire sequence can be represented by $H(X)$ bits per symbol or more, where $H(X)$ is the binary entropy function. If you attempt to represent the sequence with fewer than $NH(X)$ bits, you will lose information.

$$H(X) = \sum_{x} P(X=x)\log_2\left\{\frac{1}{P(X=x)}\right\}$$

Although we do not yet have the tools to derive this result, we can explore some of the ideas behind how it was derived, as well as how this translates to compression algorithms. Let us define L to be the the *expected length of a codeword*, which is a measure of how much compression a given symbol code is achieving:

$$L = \sum_x P(X=x)\ell(x)$$

where $\ell(x)$ is the length of x once it has been encoded. For example, consider a source alphabet {a,b,c} encoded as {01,00,1}, respectively, where each symbol occurs with probability {0.1,0.4,0.5}, respectively, then

$$L = 0.1*\ell(a) + 0.4*\ell(b) + 0.5*\ell(c) = 0.1*2 + 0.4*2 + 0.5*1 = 1.5 \text{ bits / symbol}$$

In order to derive a lower bound on this quantity, Shannon formalized the problem as follows: minimize the expected length of a codeword L subject to the constraint that your encoding must be uniquely decodable (i.e. it must be a prefix code). It turns out that this constrained minimization problem is convex, so you can use variational calculus methods to determine an exact solution. This solution turned out to be the Shannon entropy.

$$\min_{\text{subject to decodability}} L = \min_{\text{subject to decodability}} \sum_x P(X=x)\ell(x) = \sum_x P(X=x)\log_2\left(\frac{1}{P(X=x)}\right) = H(X)$$

We have not learned in this class how to formally state the constraint that an encoding is uniquely decodable, so we will not be able to derive this result. The derivation is covered in any introductory course on information theory (EECS 229a at Berkeley).

In fact, this derivation led Shannon almost immediately to his next conclusion, which was one can always create an encoding of the source alphabet which achieves

$$\large H(X) \leq L \leq H(X) + 1$$

This is the insight which results in Shannon-Fano codes (http://en.wikipedia.org/wiki/Shannon%E2%80%93Fano_coding).

$$1. How can you manipulate the expression \(\sum_x P(X=x)\log_2\left(\frac{1}{P(X=x)}\right) \leq L\) into the expression \(H(X) \leq L \leq H(X) + 1\) (you will need to assume the existence of a certain prefix code in order to do so). This is showing that the expected length of a codeword can always be less than the entropy + 1

Q1 solution here

Shannon and Fano used a simple rounding method on the entropy function to generate their compression algorithm, but they could not prove that this was the optimal compression achievable with an integer length code (because it was not, in fact, optimal). At the time, Huffman was a student in one of Fano's courses at MIT, where Fano gave students the option of taking a final or solving the problem of finding an optimal binary representation of an I.I.D. sequence of elements from some distribution. Huffman decided to try his hand at this problem, eventually leading to his discovery of the Huffman compression algorithm. He was also able to prove that this scheme was optimal (http://www.cs.utoronto.ca/~brudno/csc373w09/huffman.pdf).

Huffman has stated on many occasions that if he were aware that both Shannon and Fano had struggled with this problem, he probably wouldn't even have tried and simply studied for the final exam. Absent this information, however, he was able to make one of the most significant EECS discoveries ever. One of the reasons it was likely so difficult for many researchers to come up with the Huffman algorithm was that they were taking their cues from Shannon-Fano codes, which resulted from a rounding estimation of the entropy function. Most researchers investigating this problem (including Shannon) were looking for better, tighter rounding strategies that would somehow result in an optimal code. The idea of this kind of simple greedy algorithm wasn't even on their radar.

Huffman Code vs. Entropy in Our Example

The following code block computes the entropy of the distribution over quantized values in the song, and then shows the bits/symbol achieved by the Huffman algorithm. Notice that it is smaller than the entropy + 1, as it must be since it is more optimal than a Shannon-Fano code, which is guaranteed to have expected codeword length between the entropy and entropy + 1.

In []:
```
probabilities = freq_dict.values()
probsum = sum(probabilities)
probabilities = probabilities / probsum
entropy = sum([p*np.log2(1/p) for p in probabilities])
print "Entropy of Distribution: ", entropy
print "Achieved Bits/Symbol of Huffman Code: ", sum([len(huff[val])*(freq_dict
[val]/probsum) for val in huff])
```

\(\mathcal{B}\)onus Question: Suppose I have some source alphabet \(\mathcal{A}\) with a distribution \(\mathcal{P}\) over it, and I create a Huffman code over \(\mathcal{A}\). If I generate an infinitely long, random binary string (i.e. infinite sequence of IID Bernoulli 1/2's), and decode it using the generated Huffman code, what will be the observed output distribution over symbols?

Bonus Solution

While this discussion of compression algorithms was somewhat informal and not very rigorous, if you followed the links and did a little of your own reading you should now have enough information to understand the following clip from the first season of HBO's `Silicon Valley`:

In [1]:
```
from IPython.display import YouTubeVideo
print "Middle out?"
YouTubeVideo('l49MHwooaVQ')
```

```
Middle out?
```

Out[1]:

Transmitting Across A Channel

As we've seen, source coding is an attempt represent a file as minimally as possible, compressing it to the smallest possible size. When transmitting a file, however, the opposite idea comes into play. We want to increase the redundancy in our file to make sure it gets through correctly, even in the presence of erasures or errors. We will explore attempting to send our Huffman coded audio file across a communication channel.

In general, a communication system is as follows:

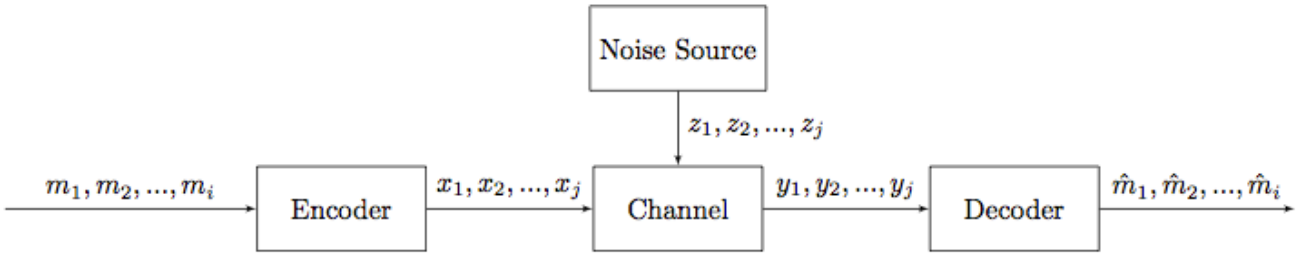

Figure 1

You are interested in transmitting a message \(m_1,m_2,\ldots\), which in our case will be a binary string representing our compressed audio file. Next, this file will be encoded in some way which adds redundancy to the file. When the encoded message \(x_1,x_2,\ldots\) is sent through the communication channel, it is corrupted in some way by a noise source, before being received as output \(y_1,y_2,\ldots\). This output is passed through a decoder, which will take advantage of the redundancy added by the encoder in order to accurately reconstruct the file. The decoders estimate for the input message will be \(\hat{m}_1,\hat{m}_2,\ldots\), and we would hope that it is the exact same as the input message.

There are dozens of models for different communication channels, but let us look at one in particular: the Binary Symmetric Channel

BSC

The binary symmetric channel is a very simple model of a communication channel, and is widely used in probabilistic analysis, information theory, and coding theory. In this model, a transmitter wishes to send a bit (a zero or a one), and the receiver receives a bit. The bit can either be transmitted correctly or can be flipped during tranmission. The "crossover probability" is the probability with which the bit is flipped, which we will denote as \(p\). Thus the bit is transmitted correctly with probability \(1-p\). A BSC with these properties is depicted below in Figure 2.

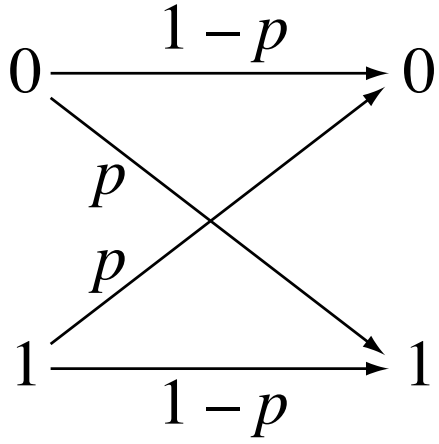

Figure 2: The Binary Symmetric Channel (BSC) is one of the simplest communication channels to analyze, and comes up very often in communications theory.

If we send our file across this channel, each bit will be flipped with a probability \(p\). Let us take a look at what happens when we try to send our file through this channel without encoding it with any redundancy. In this example we set \(p=0.01\).

In [ ]:
```
from random import random

def simulateBSC(message,p):
    output = [message[i] if random() > p else str(1-int(message[i])) for i in
range(len(message))]
    return output

output = simulateBSC(encoded,0.01)
```

In [ ]:
```
decoded = decode_song(output,huff)
```

In [ ]:
```
plt.plot(time[0:120], decoded[0:120]) # Plot the first 0.01 second
plt.title("First 0.01 seconds of decoded version of 'School Bell Ringing'")
plt.xlabel("Time")
plt.ylabel("Amplitude")
```

In [ ]:
```
playMusic('decoded_song', fs, decoded)
```

The song doesn't sound so good anymore does it? Let's explore using a repititon code to make sure that it is not corrupted during transmission.

\(\mathcal{B}\)onus Question: Why does the song still even sound remotely like the original? Once you flip a bit, the whole Huffman decoding algorithm is thrown off from there on out, and you really could be anywhere in the Huffman tree when decoding. In fact, you're very likely to have an output which doesn't even have the same length as the original file before the Huffman code was applied. Yet, the song still sounds similar to the original. Hypothesize as to why that is the case. Run some experiments and simulations to support your hypothesis. (Note: Your instructors have come up with differing explanations for this phenomenon, so it would be interesting to see if any of the students could convincingly justify a hypothesis)

Bonus Solution

Repetition Code Analysis over a BSC

We know that if we send one physical bit through this channel, the probability of receiving it correctly is \(1-p\), and that this is the best we can do (after all, this is the nature of the channel). However, we can do better if we send only one bit of information through the channel by encoding it in more than one physical bit (it is important to note the difference between physical bits and information bits). The most obvious way to do this is through a repetition code. That is, each time we want to send a bit of information, we repeat the bit \(r\) times and send them all through the channel. At the receiving end, we use a majority decoding strategy (i.e. decode to whichever symbol, 0 or 1, has been received greater than \(\frac{r}{2}\) times).

$$2. Let us suppose we use a repetition code to send one bit of information through a BSC, repeating the bit \(r\) times. Let \(Z_i = \mathbb{1}_{\{\text{\)i^$ bit is flipped}}}$ (indicator that the \(i^{\text{th}}\) tranmitted bit is flipped by the channel) and let us denote \(Z = \sum_{i=1}^r Z_i\).

a. What is the distribution of \(Z\)? What is its mean and variance?

Solution Here

b. Write an expression for the exact probability of decoding error when transmitting one information bit, given some \(p\) and \(r\).

Solution Here

c. Use Markov's inequality to provide an upper bound on this probability of error in terms of \(p\) and \(r\). Rewrite this in terms of \(r\) if you wish to do better than some fixed probability of error, \(\epsilon\).

Solution Here

d. Now, use Chebyshev's inequality to bound the same probability of error. Again, rewrite this in terms of \(r\) if you want to have probability of error less than \(\epsilon\).

Solution Here

e. Given some \(p < \frac{1}{2}\), use the Gaussian approximation of \(Z\) to determine \(r\) such that the probability of error in the transmission goes to \(\epsilon\). That is, use the CLT in order to obtain an \(approximation\) for \(r\) that will give you the desired probability of error. You may leave your answer in terms of an inverse \(\mathcal{Q}\) function.

Solution Here

f. As we said before, the CLT is not a bound. If we have high reliability systems that we wish to model, we can't rely on approximations that may or may not result in choices that trash our system. We need a good, solid upper bound that we can rely on. Enter Chernoff bounds. They do exactly that for us. Use the Chernoff bound to calculate a lower bound on \(r\) in order to achieve the desired probability of error \(\epsilon\).

Forgot what chernoff bounds (http://inst.eecs.berkeley.edu/~cs70/fa14/notes/n19.pdf) are? Here's a quick summary on how to go about it. Given Markov's inequality, we know that \(P(X > \alpha) < \frac{\mathbb{E}[X]}{\alpha}\). This also holds for any positive, nondecreasing function \(f \rightarrow \mathbb{R}^+\). So \(P(f(X) > f(\alpha)) < \frac{\mathbb{E}[f([X])}{f(\alpha)}\). Remember why CLT does so much better than Chebyshev (Look up why if you don't)? Well, let's use a clever choice of \(f\) to get a similar advantage. Let \(f(x) = e^{sX}\), where \(s > 0\). Note that the wiggle room we have in \(s\) is effectively acting like a change of base. Since we're trying to find a nice, tight upper bound on the probability, we should look for \(P(X > \alpha) \leq \min_{s>0}\frac{\mathbb{E}[e^{sX}]}{e^{s\alpha}} = e^{-\phi_X(\alpha)}\). We note that the negation flips the min to a max, and define \(\phi_X(\alpha) = \max_{s>0}(s\alpha - \ln{\mathbb{E}[e^{-sX}]})\). Come to HW party for more information!

Solution Here

g. You now have many different bounds and approximations to see how large \(r\) must be in order to attain a probability of error. For \(p = .2\), calculate what \(r\) should be to attain an error probability of \(10^{-6}\), when using Chebyshev, Chernoff, and the CLT.

Solution Here

h. Simulation time! For various values of \(p < .5\), plot the probability of error you obtain from each of the bounds as you vary \(r\) (Chebyshev, CLT, Chernoff). Make the \(y\)-axis log-scaled. What do you see? Play around with the values of \(p\) to explore when each of the bounds are of most use. With the right parameters, you should be able to get a sense for when you can trust in the central limit theorem (hint: \(central\)).

If you need a starting point, try the following values for \(p\): .2, .4. If any line other than the exact probability of error is jagged, you're dealing with an integer effect.

Solution Here

i. Suppose instead of sending a 1-bit (information) message, we are sending a message with \(s\) information bits. Then an example repetition code with \(r=2\) and \(s=5\) would map the message \(10110 \mapsto 1100111100\). If \(r\) is chosen such that the probability of error on each information bit is \(\epsilon\), then determine an expression for the probability of error of the entire message. Justify.

Solution Here

j. Create a Monte Carlo simulation of sending a repetition coded message through a binary symmetric channel.

Empirically determine what the probability of error, \(\epsilon\) is given a message size \(m_{\text{size}}\), repetition parameter \(r\), and crossover probability \(p\). Plot \(\epsilon\) as you vary \(r\) (be sure to plot against the total additional bits on the x-axis, so \(r\cdot m_{\text{size}}\). Repeat for different values of \(p\). In particular, explore the effects of increasing \(r\) in the case of \(p > \frac{1}{2}\) and \(p=\frac{1}{2}\).

What do you notice? Do you ever have to change the method used to reconstruct the original message from the given codeword? (assume channel conditions are known at all stages—i.e. the crossover probability is known at the decoder).

Which of your estimates is closest to your empirical results?

Solution Here

k. Send the song through the BSC again, but this time use a repitition code. Set \(p=0.01\) as before, and try to achieve a probability of error of \(10^{-6}\). Use a majority decoding strategy. Does the song sound better this time?

In [ ]:
```
# Code here
```

Hooray! Lab Done!