Lecture 1:

Basics of Information Theory

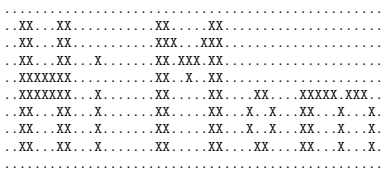
September 11, 2006

What is Information Theory? 1

- Information Theory is the formal study of the (related) problems of data compression and data communication (error correction).
- Data compression: the task of taking a sequence of symbols and re-representing them in a more compact way but so that we can still recover the original sequence. Examples:
  - Compressing computer files (e.g. gzip).
  - Compressing images (e.g. jpeg,gif).
  - Compressing audio/video (e.g. mpeg).
- Communication: transmitting and storing sequences of symbols reliably when the medium we are using introduces errors. Examples:
  - ECC memory
  - Error corrections on CDs.
  - Cell phone and space probe transmissions.

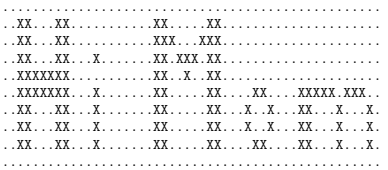
Example: compressing black and white images 2

- Say we are trying to compress an image of black and white pixels:

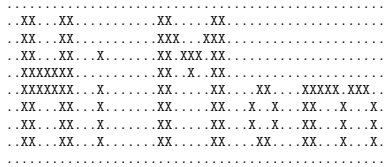


- e.g. the image above is 10 rows by 50 columns, so it has 500 pixels.
- If we encode white with 0 and black with 1, it takes 500 bits ("binary digits") to store this image.
- Can we do better? Hmm...
  Well, there are more white than black pixels.
  And both tend to come in clumps together.
  Can we exploit these observations to make our encoding shorter?

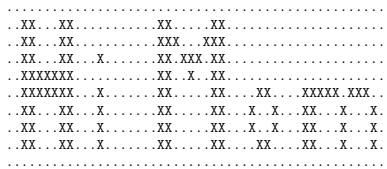
Send only black pixels 3



- One idea is to encode only the black pixels, by encoding the row and starting/ending columns for each "run" of black as binary #s.
- This requires (4+6+6)=16 bits per run (can you see why?).
- In the image above, there are 54 black runs, so it would cost 54x16 = 864 bits to encode this image.
- That's worse than the 500 bit encoding we started with!



- Another idea is to scan out the image (left to right, top to bottom), and encode "runs" of white/black pixels. (Assume the size of the image is known.)
- Say we use 8 bits per run, with the first bit to tell us the colour, and the other 7 bits the length of the run (from 0 to 127 in binary).
- In the image above, there are 55 white and 54 black runs, so the total size would be 8x(55+54)=872 bits.
- That's slightly worse than 864 and still worse than 500.
- Actually, we only need to send the colour of the first run (why?), but that only saves us 108 bits, still leaving 764.



- Rather than sending each row's runs on its own, we could encode the top left and bottom right coordinates of rectangles containing all black pixels.
- Each rectangle requires 2*(4+8)=24 bits.
- In the image above, there are 20 black rectangles, so it would cost 20x20 = 400 bits to encode this image.
- Finally, a bit better than 500!
- Other ideas: use fewer bits per run; don't allow runs of length zero; use different schemes for black vs. white runs; divide the image into sub-images and encode only those which aren't all white; ...

- Moral of the story: the more knowledge about our source we incorporate into our compression scheme, the better off we are (as long as the knowledge is correct).
- The most common way to incorporate knowledge is to build a *probabilistic model* of the sorts of inputs you expect to see. Given such a model, information theory can help us solve the mathematical and algorithmic problem of efficiently encoding sequences from the source.
- E.g. we might model black and white images using the *conditional probability* that a pixel is black given a certain number of pixels above it and left of it.
- If we somehow know enough about the source to specify these probabilities we can use a *static model*; more likely we'll have to learn them as we go through the image – an *adaptive model*.

- Can we say anything about the *best* scheme we could possibly come up with to compress images like "Hi Mom"?
- Surprisingly, if we model images (or any other sequence of symbols) as coming from a probabilistic source, and imagine using a fixed encoder over and over again to compress whatever the source produces, we can say something (quite strong) about the *best average compression* we could ever achieve.
- In particular, then there is a hard lower limit, called the *source entropy* below which no scheme can losslessly compress.
- This is Shannon's famous **Noiseless Coding Theorem** (1948).
- Limits performance, given a source model. Many algorithms exist which perform nearly optimally with respect to these limits.
- But often quite hard to get good source models; therein lies the art of applying information theory to real world problems.

Model all data/messages as strings of binary digits 8

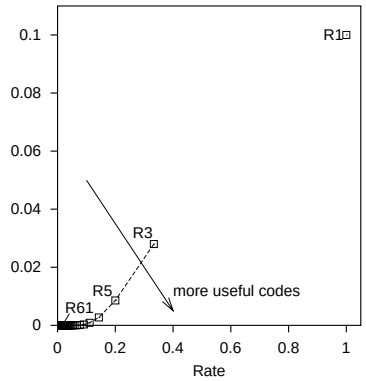
- In order to be able to perform algorithmic and mathematical analyses of the problems of data compression and communication, we will model *all* data/messages to be transmitted or compressed as strings of binary digits (bits).
- For our purposes, the *content* of the message is irrelevant; the bits could come from a file, as a stream of digitized audio over a wireless channel, as an email or HTML page over the web. The *physical medium* of the communication channel is also irrelevant; it could be a noisy radio link, a disk drive, or a phone line.
- We care only about *statistical structure* in message/noisy channel.
- For compression, we want to find patterns that occur non-randomly which we can exploit to achieve compression.
- For communication, we want to design an encoding scheme that is robust to the kind of distortions our channel will introduce.

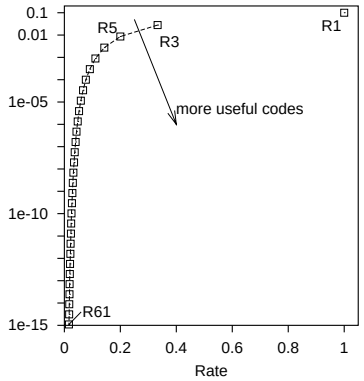
Example: Error Correction in Noisy Channels 9

- Suppose we are trying to send a sequence of bits through a channel that sometimes randomly flips a bit from 0⟶1 or vice versa.
- How can we encode the original sequence so that we can still recover it reliably at the other end? Simple idea: repetition code.
- Send each bit k times. Effective transmission rate is now 1/k.
  e.g. with $k = 2$ to send 01101 we actually send 0011110011
  with $k = 3$ to send 01101 we actually send 000111111000111
- The decoder looks at groups of k received bits and replaces each group with the bit that occurs most often in the group (which you can prove is the best thing to do) or requests resend.
  Example, with $k = 3$
  sequence 01101 → 000111111000111 transmission
  received code → 010111101100101 → 01101 decoded
- How much does this improve reliability (vs. rate reduction)?

Error Probability vs. Rate 10

Here are the plots are for $P = 0.1$.





Analysis of the Repetition Code 11

- Assume each bit of our transmission gets flipped with some probability P, independently for each bit. (May not be true.)
- Sender: repeat each bit k times. Receiver: take majority vote.
- For odd k, we make a decoding error if a group has $> k/2$ errors.
- Probability of decoding an original bit correctly $\propto \sum_{i=(k+1)/2}^{k} P^i$.
- If $P = 0.2$ and we are sending message of only 8 bits:
  $k = 1 \Rightarrow$ p(message-ok) = 0.851; rate=1.0
  $k = 2 \Rightarrow$ p(message-ok) = 0.996; rate$\leq$0.365
  $k = 3 \Rightarrow$ p(message-ok) = 0.999; rate=0.333
- Of course, our rate of transmission goes *down* by a factor of k.
- As we increase k, we can drive p(correct) to 1, but we also drive our transmission rate to zero at the same time.
- Is this inevitable?

The Surprising Answer: No! 12

- Amazingly, using smarter schemes than repetition, it turns out that we actually *can* transmit with arbitrarily small error at a finite (non-zero) rate called the *capacity* of the channel.
- This is Shannon's famous **Noisy Coding Theorem** (1948).
- For example, in our channel with $P = 0.1$, the capacity turns out to be 0.531, in other words we can obtain virtually zero errors while only transmitting less than twice as many bits are in the original message.
- But there is a catch: to achieve arbitrarily small errors, we need to encode the original message in long blocks of bits (not just one bit at a time). This creates a large delay at the decoder since we have to wait for a whole block to be decoded before we can see the bits at the start of the block.

Mathematical Setup 13

- We can formalize compression and communication as specific mathematical problems.
- We always start with a sequence of symbols $s = s_1, s_2, \ldots, s_N$ from a finite alphabet A. (e.g. ASCII strings).
- During compression/transmission we will convert this sequence into a new sequence $r = r_1, r_2, \ldots, r_M$ (using a possibly different alphabet B) and then back into s during decompression/reception.
- Our goal is to come up with two functions: an *encoder* $e(s) = r$ and a *decoder* $d(r)$.
- In both cases, we want to be able to recover the original string from its encoded version, while keeping the length of the encoding short.

Requirements on the Encoder and Decoder 14

- In *compression*, we require that $d(e(s)) = s \quad \forall s$ (i.e. the encoder and decoder functions compose to identity).
- Our goal is to make the "size" of $e(s)$ as small as possible, "on average" (averaged over our source distribution).
- We want the encoder to *remove redundancy* from input sequences.

- In *transmission*, r will have noise added to it by some (stochastic) noise function $n(r) = r'$ called the *channel*.
- Our goal is to make $d(n(e(s))) = s$ "almost always", without making $e(s)$ "too big" (now averaged over both the source and noise distributions).
- We want the encoder to *add redundancy* to the input sequence.

Measuring how well we do 15

- In compression, we want our encoding to be as *small* as possible.
- In transmission, we want to maximize the *rate* at which we can send/receive over the noisy channel, while still achieving acceptably good recovery of the original data.
- In many cases (e.g. compressing computer files, reading/writing to memory chips) we require *lossless recovery*, which means that the recovered sequence of symbols is exactly identical to the original.
- In other cases (e.g. images, audio, video) we are willing to accept some *distortion*, meaning that the recovered sequence may be slightly different than the original. Of course, this often means we can compress/transmit much more aggressively.
- We may also require that the encoding or decoding process be fast, have small delay or use only a small amount of processor power or extra memory (e.g. streaming audio).

- The mathematical theory behind solving the problems of compression and communication, culminating in two famous theorems proved by Claude Shannon in 1948 that give us limits on how well *any* method can do.
- Practical algorithms (which took much longer to appear) for actually doing compression/communication that are almost optimal according to the theory. Examples: *arithmetic coding, linear codes, low-density parity check codes*.
- Modelling issues; i.e. how we can take a real world problem and cast in into the mathematical form necessary for us to analyze it with Shannon's theorems and apply known algorithms to it. Examples: source modeling using *dictionary methods* such as *PPM* and *Lempel-Ziff* (on which `gzip` is based); channel models such as *binary symmetric channels*.

- You'll need certain tools to understand the theory and algorithms we will cover this term:
  - *Probability and Statistics*
  - *Linear Algebra*
  - *Modular Arithmetic*
  - *Basic Programming*
- Possible Extra Topics (time permitting)
  - Shannon's *rate-distortion theory* for lossy data compression.
  - Practical algorithms for lossy data compression.
  - Information theory with continuous random variables (as opposed to discrete symbols).
  - Other applications of information theory in biology, physics, etc.