CSC 310: Information Theory

University of Toronto, Fall 2011

Instructor: Radford M. Neal

Week 13

Good Codes and Minimum Distance

Recall that for a code to be guaranteed to correct up to t errors, its minimum distance must be at least $2t + 1$.

What's the minimum distance for the random codes used to prove the noisy coding theorem?

A random N-bit code is very likely to have minimum distance $d \leq N/2$ — if we pick two codewords randomly, about half their bits will differ. So these codes are likely *not guaranteed* to correct patterns of $N/4$ or more errors.

A BSC with error probability f will produce about Nf errors. So for $f > 1/4$, we expect to get more errors than the code is guaranteed to correct. Yet we know these codes are good!

Conclusion: A code may be able to correct *almost all* patterns of t errors even if it can't correct *all* such patterns.

What a Good Linear Code Looks Like

Minimum distance isn't the whole story, but nevertheless, it's not good for a linear code to have very low-weight code words (and hence very small minimum distance).

A consequence: The generator matrix for a good code should not be sparse — each row should have many 1s, so that encoding a message with only one 1 produces a codeword that has many 1s.

The decoder's perspective: To be confident of decoding correctly, getting even *one* bit wrong should produce a large change in the codeword, which will be noticeable (unless we're very unlucky).

Low Density Parity Check Codes

We should avoid sparse generator matrices. But can we use a sparse parity-check matrix?

Doing so isn't *quite* optimal, but such "Low Density Parity Check" (LDPC) codes can be very good.

The big advantage of LDPC codes: There is a *computationally feasible* way of decoding them that is good, though not optimal.

We can construct LDPC codes randomly, in various ways.

One way: Randomly generate columns of H with exactly three 1s in them.

For best results, equalize the number of 1s in each row (as much as possible) by randomly picking the position of the three 1s in the next column from among rows that don't already have $3N/(N-K)$ 1s.

Example: A $[50, 25]$ LDPC Code

Here's the parity-check matrix for a small LDPC code (three 1s in each column, six in each row).

```
0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0
0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0
0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0
```

A Generator Matrix for the Example

A systematic generator matrix obtained from the parity-check matrix (with columns re-ordered):

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1 1 0 1 1 1 0 1 0 0 1 0 1 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 1 0 0 1 1 0 1 1 1 0 1 1 1 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 0 0 1 0 0 1 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 0 1
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 0 1 0 0 0 1 0 1 1 0 1 1
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 0 1 1 1 1 1 1 0 1 0 0 0 1 0 1 1 0 1 1
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 0 0 1 0 0 1 1 0 1 1
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 1 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 1 1 1 1 0 1 0 0 0 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 1 1 0 0 1 1 0 1 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 0 1 0 0 1 1 0 1 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 1 0 1 1 1 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 0 1 0 0 0 1 1 1 0 1 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0 0 1 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 1 1 1 1 0 1 0 1 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 1 0 1 0 0 0 1 0 1 1 0 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 0 0 1 1 1 0 1 0 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 1 0 1 0 0 0 0 0 1 0 1 1 1 0 1 0 1 0 0 0
```

Decoding LDPC Codes

To encode a message with an LDPC, we just multiply it by the generator matrix. But how do we decode?

The optimal method (assuming a BSC, and equally-probable messages) is to pick the codeword nearest to what was received. But this is computationally infeasible when K and $N - K$ are large.

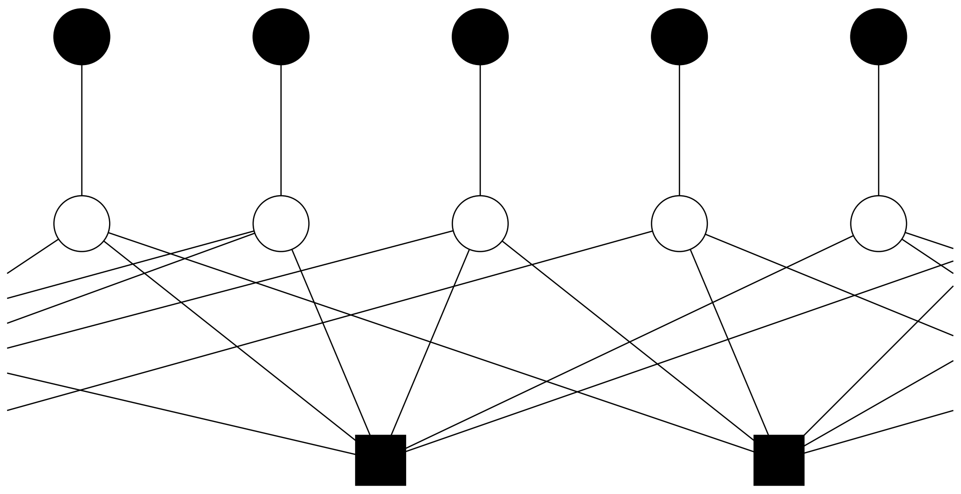
The reason LDPC codes are interesting is that the sparseness of their parity-check matrices allows for an *approximate* (good, but not optimal) decoding method that works by *propagating probabilities* through a graph.

Graphical Representation of a Code

We can represent a code by a graph:

- Empty circles represent bits of a codeword.
- Black circles represent received data bits.
- Black squares represent parity checks.

Here's a fragment of such a graph:



Notice that each codeword bit connects to three parity checks — corresponding to the three 1s in each column of H. Each parity check connects to six codeword bits.

Our task: Fill in the empty circles.

Decoding by Propagating Probabilities

We can't be absolutely sure of the codeword bits, but we can keep track of the *odds* in favour of 1 over 0 (the ratio of the probability of 1 over the probability of 0).

Each black node will send each codeword bit it connects to a message giving its idea of what the odds for 1 over 0 should be for that bit.

All the messages a codeword receives are multiplied to give the current idea of what the odds are for that bit — used to guess the codeword once these odds have stabilized.

But first, we iterate: Each codeword bit sends each parity check it connects to a message with its current odds, which the parity check node uses to update its messages to other codeword bits. Messages propagate until the odds have stabilized.

Details of the Messages

Received data bit to codeword bit: For a BSC, odds sent are $(1-f)/f$ if the received data is 1, $f/(1-f)$ if the received data is 0.

Parity check to codeword bit: Message is the probability of the parity check being satisfied if that bit is 1, divided by the probability if that bit is 0. These probabilities are calculated based on that parity check's idea of the odds for the *other* bits in the parity check being 1 versus 0.

Codeword bit to a parity check: Message is the odds of the bit being 1 versus 0, based on the received data, and on the messages from the *other* parity checks the codeword bit is involved in.

Avoiding Double-Counting Information

Messages sent between codeword bits and parity checks exclude information obtained from the node the message is being sent to. This avoids undesirable "double-counting" of information when a message comes back from that node.

But: This works perfectly only if the graph is a tree. If there are cycles in the graph, information can return to its source indirectly.

This is why probability propagation is only an *approximate* decoding method. It works well up to a point, but doesn't have as low an error rate as nearest-neighbor (maximum likelihood) decoding would achieve.

Demonstration of LDPC Codes

I tried rate 1/2 LDPC codes with three bits in each column of H, with varying codeword lengths, tested using a BSC with varying error probability, f, and hence capacity, $C = 1 - H_2(f)$.

Here are the block error rates for three such codes, estimated from 1000 simulated messages:

f	C	$[100, 50]$	$[1000, 500]$	$[10000, 5000]$
0.02	0.86	0.000	0.000	0.000
0.03	0.81	0.012	0.000	0.000
0.04	0.76	0.059	0.000	0.000
0.05	0.71	0.108	0.000	0.000
0.06	0.67	0.213	0.005	0.000
0.07	0.63	0.327	0.104	0.000
0.08	0.60	0.482	0.404	0.125

Tests were done with software available from my web page, at

`http://www.cs.utoronto.ca/~radford/`

History of LDPC and Related Codes

- Gallager, LDPC codes — 1961.

  True merits not realized? Computers too slow? Largely ignored and forgotten.

- Berrou, *et al*, TURBO codes — 1993.

  Surprisingly good codes, practically decodable, but not really understood.

- MacKay and Neal — 1995.

  Reinvent LDPC codes, slightly improved. Show they're almost as good as TURBO codes. Decoding algorithm related to other probabilistic inference methods.

- Many (Richardson, Frey, etc.) — ongoing.

  Further improvements in LDPC codes, relationship to TURBO codes, theory of why it all works.

Lossy Compression

Many kinds of data — such as images and audio signals — contain "noise" and other information that is not really of interest. Preserving such useless information seems wasteful.

A common approach: *Lossy compression*, for which decompressing a compressed file gives you something *close* to the original, but not necessarily exactly the original.

We should be able to compress to a smaller file size if we don't have to reproduce the original exactly.

What do We Mean by “Close”?

Any lossy compression scheme is based (at least implicitly) on some idea of what counts as “close to the original”.

This is a question that can only be answered by considering the *users* of the compression program, and what they want.

For images and audio signals, two fundamental issues are:

- What differences can humans perceive?
  It is thought, for example, that humans perceive only *frequencies* in audio but not the associated *phases* of sine waves.
- What differences do humans find annoying or distracting?
  For example, slight changes in colour might be regarded as less important than making a straight line be jagged.

Formalizing Distortion

Let the input to the compression program be the sequence a_1, a_2, $\ldots$, a_N. The decompression program outputs the sequence b_1, b_2, $\ldots$, b_N. The a_i and the b_i might come from the same or different alphabets.

We can measure how close the decompressed output is to the original by its average "distortion":

$$\bar{d} \;=\; \frac{1}{N}\sum_{i=1}^{N} d(a_i, b_i)$$

$d(a, b)$ is a non-negative *distortion function* measuring how bad it is for a decompressed symbol to be b if the original was a.

Note: In practice, the overall distortion might not be a sum of distortions for individual symbols, but I'll ignore that complication.

Simple Distortion Functions

Distortion functions that measure what we're really interested in are likely to be complicated. But we can consider some simple examples that are easier to handle.

Hamming distance: For a bilevel image (such as a fax), we might use a distortion function for which $d(0,0) = d(1,1) = 0$ and $d(0,1) = d(1,0) = 1$.

Squared error: For a gray-scale image with pixel intensity values in $\{0, \ldots, 255\}$, we might use $d(a,b) = (a-b)^2$.

Rate for a Given Distortion

If the entropy of our source is H, we expect to be able to losslessly compress N symbols into NH bits — ie, at rate H.

But what if decompression is allowed to produce any output that has average distortion less than some limit, D?

The *rate distortion function*, $R(D)$, tells us how well we can do — it is the *smallest* rate (average bits per input symbol) for *any* compression scheme that has average distortion no greater than D.

Note that $R(D)$ depends on both the source probabilities and on the distortion function chosen.

Example: Binary Data

Suppose our source alphabet is binary, with equal probabilities for 0 and 1 (independently from symbol to symbol).

Suppose we will decompress to the same alphabet, and that we measure distortion by Hamming distance.

If we insist on lossless compression, we can't compress at all, since the entropy is one.

How well can we compress if we allow an average distortion of up to 1/8 — ie, if we allow up to one in eight bits to be wrong?

Lossy Compression Using Hamming Codes

Here's a scheme that compresses a binary source to 4/7 of the original file size while altering only 1/8 of the bits, on average:

1. Grab the next 7 input bits from the source.
2. Pretend these bits are received data from a $[7, 4]$ Hamming code in systematic form.
3. "Decode" these 7 bits by the usual Hamming code procedure.
4. Output the 4 systematic bits from this "decoded" codeword.

To decompress, we take blocks of 4 bits and "encode" them in 7 bits the usual way.

Result: Perfect reconstruction of the 7 bits 1/8 of the time; one wrong bit 7/8 of the time.

The Rate Distortion Theorem

Consider all channels, $\mathcal{C}$, with input alphabet $\{a_i\}$ and output alphabet $\{b_j\}$. Given the input probabilities that our source has, we can find for each such channel

- Its mutual information, $I(\mathcal{A}, \mathcal{B})$.
- The average distortion between the channel input and the resulting output.

Shannon proved that the rate distortion function, $R(D)$, is equal to the minimum value for $I(\mathcal{A}, \mathcal{B})$ over all channels whose average distortion is no more than D.

For a binary source where 0 has probability $p_0 \leq 1/2$, and where distortion is measured by Hamming distance, it turns out that

$$R(D) = \begin{cases} H_2(p_0) - H_2(D) & \text{for } 0 \leq D \leq p_0 \\ 0 & \text{for } D > p_0 \end{cases}$$

How Can We Achieve This?

As for his noisy coding theorem, Shannon's rate distortion theorem can be proved using codes chosen at random.

Consider a channel $\mathcal{C}$ that minimizes $I(\mathcal{A}, \mathcal{B})$ subject to the distortion between input and output being less than D. We find the output probabilities for such a channel, and then pick codewords at random with symbol probabilities equal to these output probabilities.

Encoding procedure: Find the codeword closest (as measured by distortion) to the actual message; then send an index of that codeword. If we chose 2^K codewords, sending this index will take K bits, for a rate of K/N.

Decoding procedure: Output the codeword corresponding to the received index.

Lossy Data Compression in Practice

Shannon's elegant theory currently plays little role in practical lossy data compression (or the similar task of "vector quantization").

Instead, various *ad hoc* methods are used.

Two reasons for this:

1. Formalizing a suitable distortion function taking account of human perceptual abilities and tolerances is difficult.

2. The step from the impractical random codes used to prove the rate distortion theorem to a practical method of optimal compression hasn't been achieved.

Overcoming these issues is a current challenge for research.