# Compression

CISC489/689-010, Lecture #5

Monday, February 23

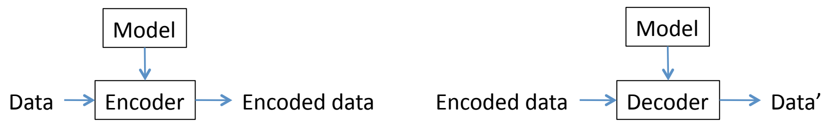
Ben Carterette

## Why Compress?

- Recall from last time: index files
  - Vocabulary file contains all terms with pointers to lists in an inverted file.
  - Inverted file contains lists of all documents the terms appear in.
  - Collection file contains all the document names.
- This can be a lot of information to store, access, and transfer!
  - Easily takes up several gigabytes in memory or on disk.
- Compression helps work with large files.

# What is Compression?

- Compression is a type of *encoding* of data.




- The goal is to make the data smaller.
- A very big topic in CS and engineering.
  - We have a full course on data compression.

# Types of Compression

- Lossless compression:
  - The encoding preserves all information about the original data.
  - The original data can be recovered completely.
- Lossy compression:
  - The encoding loses some information about the original data.
  - The original data can be recovered approximately.
- Signature file indexes are a type of lossy compression.

## Compression in IR

- Text compression:
  - Used to compress vocabulary, document names, original document text.
  - Based on assumptions about language.
- Data compression:
  - Used to compress inverted lists.
  - Not generally based on assumptions, but on observations about the data.

## Preliminaries

- "Text" means based on characters.
- What is a character? (Think C, C++)
  - A data type.
  - Generally stores 1 byte.
  - 1 byte = 8 bits.
  - Since each bit can be 0 or 1, one byte can store $2^8$ = 256 possible characters.

## ASCII Encoding

- ASCII is a common character encoding.
- Each character is represented with 8 bits.
  - A = ASCII 65 = 01000001
  - ¿ = ASCII 168 = 10101000
  - 256 possible characters.
- Decoding: table maps bytes to characters.
- Fish: 01000110 01101001 01110011 01101000
  - 32 bits = 4 bytes.

## Fixed Length Codes

- Short bytes: use the smallest number of bits needed to represent all characters.
  - English has 26 letters. How many bits needed?
  - 5 bits can represent $2^5$ = 32 letters.
  - 26 letters * 2 cases = 52 characters.
    - Requires 6 bits… or does it?
- Use numbers 1-30 (00001 – 11110) to represent two sets of characters.
  - Use 0 (00000) to toggle the first set (e.g. capital letters).
  - Use 31 (11111) to toggle the second set (e.g. small letters).
- Fish: $\underbrace{00110}_{F}\ \underbrace{11111}_{\downarrow}\ \underbrace{01001}_{i}\ \underbrace{10011}_{s}\ \underbrace{01000}_{h}$
  - 25 bits, slightly over 3 bytes.

## Fixed Length Codes

- Bigram codes: use 8 bits to encode either 1 or 2 characters.
  - *is* would be encoded in 8 bits.
- Use values 0-87 for space, 26 lower case, 26 upper case, 10 numbers, and 25 other characters.
- Use values 88-255 for character pairs.
  - Master (8): blank, A, E, I, O, N, T, U
  - Combining (21): blank, all other letters except JKQXYZ
  - 88 + 8*21 = 256 possibilities encoded
- Fish: 00100000 10101010 00001000
  F is h
  - 24 bits, 3 bytes.

## Fixed Length Codes

- *N*-gram codes: same as bigram, but encode character strings of length less than or equal to *n*.
- Select most common strings for 8-bit encoding in advance.
  - Goal: most commonly occurring *n*-grams require only one byte.
- Fish: 00100000 10111010
  F ish
  - 16 bits, 2 bytes.

## Fixed Length Summary

- Fixed length codes are generally simple, easy to use, and effective when assumptions are met.
- Limited alphabet size allowed.
- If data does not meet assumptions, compression will not be good.

## Restricted Variable Length Codes

- Idea: different characters can have encodings of different lengths.
- Similar to case-shifting in short byte codes:
  - First bit indicates case.
  - 8 most common characters encoded in 4 bits (0xxx)
  - 128 less common characters encoded in 8 bits (1xxxxxxx)
  - First bit tells you how many bits to read next.
- 8 most common English letters are e, t, a, i, n, o, r, s.
- Fish: 10000110 0011 0110 10000100
  - 24 bits, 3 bytes.

# Restricted Variable Length Codes

- 8 most common letters in English are 64% of characters in wiki000 subset.
- Expected code length = 0.64*4 bits + 0.36*8 bits = 5.44 bits per character.
- A little worse than short bytes, but can encode many more characters.
  - Can also generalize to more than 2 cases:
    - 0xxx for most common 8 characters.
    - 1xxx0xxx for next $2^6$ = 64 characters.
    - 1xxx1xxx0xxx for next $2^9$ = 512 characters, …

# Unicode

- Unicode is an encoding designed to handle many different alphabets and symbol sets.
- Unicode is a type of restricted variable length coding.
  - Uses 21 bits to encode 1,114,112 symbols.
  - First 5 bits encode “plane” (numbered 0-16).
  - Within each plane, 16 bits encode characters (numbered 0-65,536).

## UTF-n for Unicode

- UTF-n encodes Unicode using n-bit chunks.
  - Each value of n can encode all 1,114,112 symbols.
- Encodings designed to map between different values of n without losing information.
- UTF-32:
  - 32 bits can store more than 4 billion symbols.
  - Just assign each Unicode symbol a 32-bit string.
  - 11 bits never used.

## UTF-8

- "Chunk" is 8 bits (1 byte).
- Use 7 bits (0xxxxxxx) to store first 128 Unicode symbols (which are basic ASCII).
- Higher values stored in 2 or more bytes.
  - First byte encodes number of bytes in *unary*.
    - 110xxxxx means a 2-byte character.
    - 1110xxxx means a 3-byte character.
  - Remaining bytes in form 10xxxxxx.
  - Free bits (x's) used to encode symbols.

## UTF-8 Templates

- 0xxxxxxx (1 byte, 7 free bits):
  - Unicode symbols 0 to 127 (basic ASCII: A-Z, a-z, 0-9, etc.)
- 110xxxxx 10xxxxxx (2 bytes, 11 free bits):
  - Unicode symbols 128 to 2176 (Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, etc.)
- 1110xxxx 10xxxxxx 10xxxxxx (3 bytes, 16 free bits):
  - Unicode symbols 2177 to 67,714 (almost all other alphabets)
- 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx (4 bytes):
  - All remaining Unicode symbols.

## UTF-8 Examples

- Letter A is Unicode 65.
  - $0 \le 65 < 128$, so only needs 1 byte: 01000001
- Greek letter α is Unicode 945.
  - $128 \le 945 < 2176$, so needs 2 bytes.
  - Template is 110xxxxx 10xxxxxx.
  - 945 in 11 bits is 00111011001.
  - UTF-8 is 11000111 10011001.
- Korean character ᅡ is Unicode 4449.
  - $2177 \le 4449 < 67{,}714$, so needs 3 bytes.
  - Template is 1110xxxx 10xxxxxx 10xxxxxx.
  - 4449 in 16 bits is 00001000 10110001.
  - UTF-8 is 11100000 10100010 10110001.

## Restricted Variable Length Codes

- Encoding numbers:
  - Use 1 byte for numbers 0 through 127.
    - Template = 1xxxxxxx.
  - Use 2 bytes for numbers 128 through 16,512.
    - Template = 0xxxxxxx 1xxxxxxx.
  - Use 3 bytes for numbers 16,513 through 2,113,665.
    - Template = 0xxxxxxx 0xxxxxxx 1xxxxxxx.
  - Etc.
- This could be used to encode document numbers, term frequencies, term positions, etc…

## Variable Length Codes

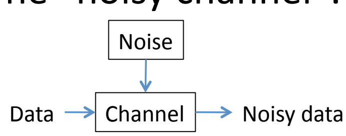
- Dictionary-based encoding: encode entire words.
  - Sort words in decreasing order of frequency.
  - Use the rank of the word to encode it.
    - the = 1, of = 2, a = 3, …, politician = 501, …, contractor = 15,304, …
  - Use numeric coding to encode the rank.
- Con: difficult to decode. Needs to have access to the sorted dictionary in order to decode.

# Variable Length Summary

- Restricted variable length codes are simple and effective.
- Assumptions about language are weaker (more likely to be met in general).
- Flexible enough to handle very large alphabets.
- Require a dictionary or other lookup table for decoding.

# Information Theory

- Encodings and compression have theoretical grounding in *information theory*.
- The "noisy channel":




- Shannon studied theoretical limits for compression and transmission rates.

Claude Shannon (1916-2001)

## Shannon Game

- The President of the United States is Barack …
  - Only one possible option. We don't even need to send the last word to transmit the information.
- The best web search engine is …
  - Many options, but one has high probability. Two others have lower but non-negligible probability. Many others have low probability.
  - We could guess the next word, but we could be wrong.
- Mary was …
  - Happy? angry? tall? Who knows…

## Information Content

- The *information content* of a message is a function of how predictable it is.
  - … Obama – very predictable → very low information content if you read U.S. news at all.
  - … Google – somewhat predictable → low (but non-zero) information content.
  - … Queen of England from 1553 to 1558 – unpredictable → high information content: you weren't expecting it.

# Encoding Information

- Let $p_i$ be the probability of message *i*.
  - For first example, $p_{Obama} = 1$.
  - For second, suppose $p_{Google} = 0.5$, $p_{Yahoo} = 0.3$, $p_{Microsoft} = 0.15$, $p_{Other} = 0.05$.
  - For third, many possibilities with low probability.
- The number of bits needed to encode *i* is $-\log_2 p_i$.
  - Obama: $-\log_2 1 = 0$ bits.
  - Google: $-\log_2 0.5 = 1$ bit; Yahoo: $-\log_2 0.3 = 1.74$ bits; Microsoft: $-\log_2 0.15 = 2.74$ bits; other = $-\log_2 0.05 = 4.32$ bits.
    - "not Google": $-\log_2 (1 – 0.5) = 1$ bit.

# Information Entropy

- The *entropy* of a message is the expected number of bits needed to encode it.
  - Expectation = sum over all possibilities, probability of possibility times value of possibility.
  - Entropy = $H(p) = -\sum_{i=1}^{n} p_i \log_2 p_i$
- First example: $H = -1*\log_2 1 = 0$.
- Second example: $H = -0.5*\log_2 0.5 – 0.3*\log_2 0.3 – 0.15*\log_2 0.15 – 0.05*\log_2 0.05 = 1.65$ bits.
  - Google vs. non-Google: H = -.5*log .5 - .5*log .5 = 1 bit.

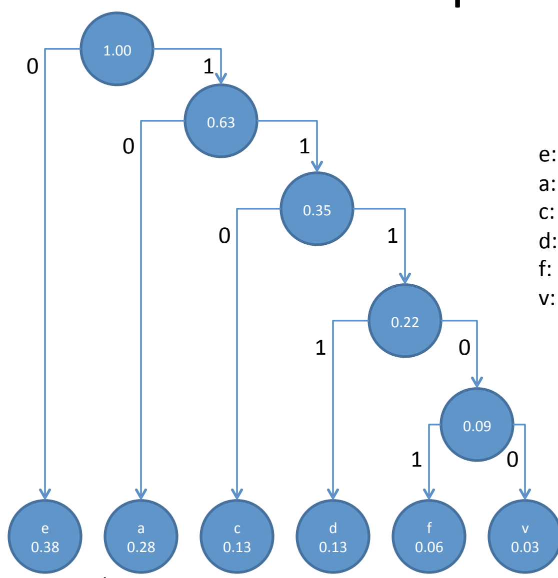
## Information Theory and Codes

- We have implicitly been using information theory to determine minimum code lengths.
  - Recall short byte codes: characters represented with 5 bits.
  - For alphabet size 26, each letter probability 1/26:
    - $-\log_2 1/26 = 4.7$ bits, so 5 bits necessary.
- Information theory allows us to find more compact representations.
  - Using frequencies of letter occurrences, we can reduce entropy to 3.56 bits or less.
  - Humans can guess the next letter in a sequence accurately; only need 1.3 bits.

## Huffman Encoding

- An information-theoretic variable-length code.
- Basic idea: create a tree
  - Calculate the probability of each symbol.
  - Make the two lowest-probability symbols or nodes inherit from a parent node.
    - P(parent) = P(child1) + P(child2)
  - Label lower-probability node 0, other node 1.
  - Iterate until all nodes connected in a tree.
- Path from root to leaf determines code of leaf.

# Huffman Example




e: 0
a: 10
c: 110
d: 1111
f: 11101
v: 11100

P(letter) = # occurrences/(total # letters)

# Huffman Codes

- Huffman codes are “prefix free”: no code is a prefix of another.
  – Uniquely decodable; lossless compression.
- They come very close to the limits of compressibility proved by Shannon.
- Decoding somewhat inefficient.
  – Must store entire tree in memory; process encoded data bit by bit.
- Works on text too.
  – Compose tree from word frequencies.

## Lempel-Ziv Compression

- A dictionary-based approach to variable length coding.
- Build a dictionary as text is encountered in the file.
  - If Zipf's law is obeyed, the dictionary will be good.
- Dictionary does not need to be stored, as both encoder and decoder know how to create it.
- Used in many modern compression programs:
  - gzip, Unix compress, zip.
  - And some compressed file formats like PNG.

## Original Algorithm (LZ77)

- Read data character-by-character.
- Greedy string-match to locate previously-compressed strings.
- Data is encoded as a sequence of tuples:
  - (number of characters to go back, length, next char)
- Example:
  - Data: abaababbbbbbbbbbba
  - Encoding: (0,0,a),(0,0,b),(2,1,a),(3,2,b),(1,10,a)
- Optimizations:
  - Use restricted variable length codes for back-pointers and lengths.
  - Store characters only when necessary.

## gzip Variant

- Use hash tables and linked lists to store compressed strings in memory.
- Improve compression using lookahead rather than simple greedy string match.
- Use Huffman codes for back-pointers, lengths, and characters.