Compression, Information and Entropy – Huffman’s coding

Lecture 25
December 1, 2015

Codes...

1. \( \Sigma \): alphabet.
2. **binary code**: assigns a string of 0s and 1s to each character in the alphabet.
3. each symbol in input = a codeword over some other alphabet.
4. Useful for transmitting messages over a wire: only 0/1.
5. receiver gets a binary stream of bits...
6. ... decode the message sent.
7. **prefix code**: reading a prefix of the input binary string uniquely match it to a code word.
8. ... continuing to decipher the rest of the stream.
9. binary/prefix code is **prefix-free** if no code is a prefix of any other.
10. ASCII and Unicode’s UTF-8 are both prefix-free binary codes.

Morse code is binary+prefix code but **not** prefix-free.

... code for \( S (\cdot \cdot \cdot) \) includes the code for \( E (\cdot) \) as a prefix.

Prefix codes are binary trees...

...characters in leafs, code word is path from root.

prefix tree: prefix tree or **code trees**.

Decoding/encoding is easy.
Frequency table for...
“A tale of two cities” by Dickens

<table>
<thead>
<tr>
<th>char</th>
<th>frequency</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>48165</td>
<td>1110</td>
</tr>
<tr>
<td>'B'</td>
<td>8414</td>
<td>10100</td>
</tr>
<tr>
<td>'C'</td>
<td>13896</td>
<td>00100</td>
</tr>
<tr>
<td>'D'</td>
<td>28041</td>
<td>01101</td>
</tr>
<tr>
<td>'E'</td>
<td>74809</td>
<td>01110</td>
</tr>
<tr>
<td>'F'</td>
<td>13559</td>
<td>11111</td>
</tr>
<tr>
<td>'G'</td>
<td>12530</td>
<td>11111</td>
</tr>
<tr>
<td>'H'</td>
<td>38961</td>
<td>11001</td>
</tr>
<tr>
<td>'I'</td>
<td>41005</td>
<td>10111</td>
</tr>
<tr>
<td>'J'</td>
<td>710</td>
<td></td>
</tr>
<tr>
<td>'K'</td>
<td>4782</td>
<td></td>
</tr>
<tr>
<td>'L'</td>
<td>22030</td>
<td>10101</td>
</tr>
<tr>
<td>'M'</td>
<td>15298</td>
<td>01000</td>
</tr>
<tr>
<td>'N'</td>
<td>42380</td>
<td></td>
</tr>
<tr>
<td>'O'</td>
<td>46499</td>
<td></td>
</tr>
<tr>
<td>'P'</td>
<td>9957</td>
<td></td>
</tr>
<tr>
<td>'Q'</td>
<td>37187</td>
<td></td>
</tr>
<tr>
<td>'R'</td>
<td>37187</td>
<td></td>
</tr>
<tr>
<td>'S'</td>
<td>37575</td>
<td>1000</td>
</tr>
<tr>
<td>'T'</td>
<td>54024</td>
<td></td>
</tr>
<tr>
<td>'U'</td>
<td>16726</td>
<td>01001</td>
</tr>
<tr>
<td>'V'</td>
<td>5199</td>
<td>11111</td>
</tr>
<tr>
<td>'W'</td>
<td>14113</td>
<td>00101</td>
</tr>
<tr>
<td>'X'</td>
<td>724</td>
<td>11111</td>
</tr>
<tr>
<td>'Y'</td>
<td>12177</td>
<td>11110</td>
</tr>
<tr>
<td>'Z'</td>
<td>215</td>
<td>11111</td>
</tr>
</tbody>
</table>

Computed prefix codes...

Mergeability of code trees

1. two trees for some disjoint parts of the alphabet...
2. Merge into larger tree by creating a new node and hanging the trees from this common node.
3. \[ M \cup U \Rightarrow M \cup U \]
4. ...put together two subtrees.
Building optimal prefix code trees

1. take two least frequent characters in frequency table...
2. ... merge them into a tree, and put the root of merged tree back into table.
3. ... instead of the two old trees.
4. Algorithm stops when there is a single tree.
5. Intuition: infrequent characters participate in a large number of merges. Long code words.
6. Algorithm is due to David Huffman (1952).
7. Resulting code is best one can do.
8. **Huffman coding**: building block used by numerous other compression algorithms.

---

Lemma: lowest leafs are siblings...

**Lemma**

1. $\mathcal{T}$: optimal code tree (prefix free!).
2. Then $\mathcal{T}$ is a full binary tree.
3. ... every node of $\mathcal{T}$ has either 0 or 2 children.
4. If height of $\mathcal{T}$ is $d$, then there are leafs nodes of height $d$ that are sibling.

---

Proof...

1. If $\exists$ internal node $v \in V(\mathcal{T})$ with single child... remove it.
2. New code tree is better compressor:
   \[ \text{cost}(\mathcal{T}) = \sum_{i=1}^{n} f[i] \times \text{len(code}(i)) \]
3. $u$: leaf $u$ with maximum depth $d$ in $\mathcal{T}$. Consider parent $v = \overline{p}(u)$.
4. $\implies v$: has two children, both leafs

---

Infrequent characters are stuck together...

**Lemma**

$x, y$: two least frequent characters (breaking ties arbitrarily).

$\exists$ optimal code tree in which $x$ and $y$ are siblings.
Proof...

1. Claim: \( \exists \) optimal code s.t. \( x \) and \( y \) are siblings + deepest.
2. \( \mathcal{T} \): optimal code tree with depth \( d \).
3. By lemma... \( \mathcal{T} \) has two leafs at depth \( d \) that are siblings,
4. If not \( x \) and \( y \), but some other characters \( \alpha \) and \( \beta \).
5. \( \mathcal{T}' \): swap \( x \) and \( \alpha \).
6. \( x \) depth inc by \( \Delta \), and depth of \( \alpha \) decreases by \( \Delta \).
7. \( \text{cost}(\mathcal{T}') = \text{cost}(\mathcal{T}) - (f[\alpha] - f[x])\Delta. \)
8. \( x \): one of the two least frequent characters.
...but \( \alpha \) is not.
9. \( \Rightarrow f[\alpha] \geq f[x]. \)
10. Swapping \( x \) and \( \alpha \) does not increase cost.
11. \( \mathcal{T} \): optimal code tree, swapping \( x \) and \( \alpha \) does not decrease cost.
12. \( \mathcal{T}' \) is also an optimal code tree
13. Must be that \( f[\alpha] = f[x] \).

Proof continued...

1. \( y \): second least frequent character.
2. \( \beta \): lowest leaf in tree. Sibling to \( x \).
3. Swapping \( y \) and \( \beta \) must give yet another optimal code tree.
4. Final opt code tree, \( x, y \) are max-depth siblings. \( \blacksquare \)

Huffman’s codes are optimal

Theorem

Huffman codes are optimal prefix-free binary codes.

Proof...

1. If message has 1 or 2 diff characters, then theorem easy.
2. \( f[1 \ldots n] \) be original input frequencies.
3. Assume \( f[1] \) and \( f[2] \) are the two smallest.
5. lemma \( \Rightarrow \exists \) opt. code tree \( \mathcal{T}_{\text{opt}} \) for \( f[1..n] \)
6. \( \mathcal{T}_{\text{opt}} \) has 1 and 2 as siblings.
7. Remove 1 and 2 from \( \mathcal{T}_{\text{opt}} \).
8. \( \mathcal{T}'_{\text{opt}} \): Remaining tree has 3, \ldots, \( n \) as leafs and “special” character \( n + 1 \) (i.e., parent 1, 2 in \( \mathcal{T}_{\text{opt}} \))
La proof continued...

1. character \( n+1 \): has frequency \( f[n+1] \).
   Now, \( f[n+1] = f[1] + f[2] \), we have
   \[
   \text{cost}(\mathcal{T}_\text{opt}) = \sum_{i=1}^{n} f[i]\text{depth}_{\mathcal{T}_\text{opt}}(i)
   \]
   \[
   = \sum_{i=3}^{n+1} f[i]\text{depth}_{\mathcal{T}_\text{opt}}(i) + f[1]\text{depth}_{\mathcal{T}_\text{opt}}(1)
   \]
   \[
   + f[2]\text{depth}_{\mathcal{T}_\text{opt}}(2) - f[n+1]\text{depth}_{\mathcal{T}_\text{opt}}(n+1)
   \]
   \[
   = \text{cost}(\mathcal{T}_\text{opt}') + (f[1] + f[2])\text{depth}(\mathcal{T}_\text{opt})
   \]
   \[
   - (f[1] + f[2])(\text{depth}(\mathcal{T}_\text{opt}) - 1)
   \]
   \[
   = \text{cost}(\mathcal{T}_\text{opt}') + f[1] + f[2].
   \]

What we get...

2. using above Huffman compression results in a compression to a file of size 439,688 bytes.
3. Ignoring space to store tree.
4. gzip: 301,295 bytes
   bzip2: 220,156 bytes!
5. Huffman encoder can be easily written in a few hours of work!
6. All later compressors use it as a black box...

La proof continued...

1. implies \( \min \) cost of \( \mathcal{T}_\text{opt} \equiv \min \) cost \( \mathcal{T}_\text{opt}' \).
2. \( \mathcal{T}_\text{opt}' \): must be optimal coding tree for \( f[3 \ldots n+1] \).
3. \( \mathcal{T}_H \): Huffman tree for \( f[3, \ldots, n+1] \)
   \( \mathcal{T}_H \): overall Huffman tree constructed for \( f[1, \ldots, n] \).
4. By construction:
   \( \mathcal{T}_H \) formed by removing leaves 1 and 2 from \( \mathcal{T}_H \).
5. By induction:
   Huffman tree generated for \( f[3, \ldots, n+1] \) is optimal.
6. \( \text{cost}(\mathcal{T}_\text{opt}') = \text{cost}(\mathcal{T}_H) \).
8. \( \implies \) Huffman tree has the same cost as the optimal tree.

Average size of code word

1. input is made out of \( n \) characters.
2. \( p_i \): fraction of input that is \( i \)th char (probability).
3. use probabilities to build Huffman tree.
4. Q: What is the length of the codewords assigned to characters as function of probabilities?
5. special case...
Average length of codewords...

Special case

Lemma
1, \ldots, n: symbols.
Assume, for i = 1, \ldots, n:
1. \( p_i = 1/2^i \): probability for the i\textsuperscript{th} symbol
2. \( l_i \geq 0 \): integer.

Then, in Huffman coding for this input, the code for i is of length \( l_i \).

Proof

1. induction of the Huffman algorithm.
2. \( n = 2 \): claim holds since there are only two characters with probability \( 1/2 \).
3. Let i and j be the two characters with lowest probability.
4. Must be \( p_i = p_j \) (otherwise, \( \sum_k p_k \neq 1 \)).
5. Huffman’s tree merges this two letters, into a single “character” that have probability \( 2p_i \).
6. New “character” has encoding of length \( l_i - 1 \), by induction (on remaining \( n - 1 \) symbols).
7. resulting tree encodes i and j by code words of length \( (l_i - 1) + 1 = l_i \).

Translating lemma...

1. \( p_i = 1/2^i \)
2. \( l_i = \lg 1/p_i \).
3. Average length of a code word is

\[
\sum_i p_i \lg \frac{1}{p_i}.
\]

4. \( X \) is a random variable that takes a value i with probability \( p_i \), then this formula is

\[
H(X) = \sum_i \Pr[X = i] \lg \frac{1}{\Pr[X = i]},
\]

which is the entropy of \( X \).