Data Compression Techniques

Part 2: Text Compression

Lecture 5: Context-Based Compression

Juha Kärkkäinen

14.11.2017
Text Compression

We will now look at techniques for text compression.

These techniques are particularly intended for compressing natural language text and other data with a similar sequential structure such as program source code. However, these techniques can achieve some compression on almost any kind of (uncompressed) data.

These are the techniques used by general purpose compressors such as zip, gzip, bzip2, 7zip, etc..
Compression Models

Entropy coding techniques such as Huffman, arithmetic and ANS coding need a probability distribution as an input. The method for determining the probabilities is called a model. Finding the best possible model is the real art of data compression.

There are three types of models:

- static
- semiadaptive or semistatic
- adaptive.

A static model is a fixed model that is known by both the encoder and the decoder and does not depend on the specific data that is being compressed. For example, the frequencies of symbols in English language computed from a large corpus of English texts could be used as the model.
A semiadaptive or semistatic model is a fixed model that is constructed from the data to be compressed. For example, the symbol frequencies computed from the text to be compressed can be used as the model. The model has to be included as a part of the compressed data.

- A semiadaptive model adapts to the data to be compressed. For example, a static model for English language would be inoptimal for other languages or something completely different such as DNA sequencies. A semiadaptive model can always be the optimal one.
- The drawback of a semiadaptive model is the need to include the model as a part of the compressed data. For short data, this could completely negate any compression achieved. On the other hand, for very long data, the space needed for the model is negligible. Sometimes it may possible to adapt the complexity of the model to the size of the data in order to minimize the total size. (According to the Minimum Description Length (MDL) principle, such a model is the best model in the machine learning sense too.)
- Every semiadaptive model can be seen as a static model with adaptable parameters. An extreme case would be to have two static models and a one bit parameter to choose between them.
An adaptive model changes during the compression. At a given point in compression, the model is a function of the previously compressed part of the data. Since that part of the data is available to the decoder at the corresponding point in decompression, there is no need to store the model. For example, we could start compressing using a uniform distribution of symbols but then adjust that distribution towards the symbol frequencies in the already processed part of the text.

- Not having to store a model saves space, but this saving can be lost to a poor compression rate in the beginning of the compression. There is no clear advantage either way. As the compression progresses, the adaptive model improves and approaches optimal. In this way, the model automatically adapts to the size of the data.

- The data is not always uniform. An optimal model for one part may be a poor model for another part. An adaptive model can adapt to such local differences by forgetting the far past.

- The disadvantage of adaptive computation is the time needed to maintain the model. Decoding, in particular, can be slow compared to semiadaptive compression.
Zeroth Order Text Compression

Let \( T = t_0 t_2 \ldots t_{n-1} \) be a text of length \( n \) over an alphabet \( \Sigma \) of size \( \sigma \). For any symbol \( s \in \Sigma \), let \( n_s \) be the number of occurrences of \( s \) in \( T \). Let \( f_s = n_s / n \) denote the frequency of \( s \) in \( T \).

**Definition**

The **zeroth order empirical entropy** of the text \( T \) is

\[
H_0(T) = - \sum_{s \in \Sigma} f_s \log f_s.
\]

The quantity \( nH_0(T) \) represents a type of lower bound for the compressed size of the text \( T \).

- If we encode the text with arithmetic coding using some probability distribution \( P \) on \( \Sigma \), the length of the encoding is about

  \[
  - \sum_{s \in \Sigma} n_s \log P(s) = -n \sum_{s \in \Sigma} f_s \log P(s).
  \]

- The distribution \( P \) minimizing the encoding is \( P(s) = f_s \) for all \( s \in \Sigma \). Then the size of the encoding is about \( nH_0(T) \approx nH(P) \) bits.
A simple **semiadaptive** encoding of $T$ consists of:

- the symbol counts $n_s$ encoded with $\gamma$-coding or some other suitable way (see Exercises 2.4 and 3.5), and
- the text symbols encoded with entropy coding using the symbol frequencies $f_s$ as probabilities.

The size of the second part is close to $nH_0(T)$ bits.

The size of the first part can be reduced slightly by encoding codeword lengths (Huffman) or rounded frequencies instead of the symbol counts.

An optimization for the second part is to encode $t_i$ using the frequencies in $T[i..n]$ (or $T[0..i]$ for ANS since it decodes in reverse order), which can be computed by the decoder using the frequencies in the already decoded part $T[0..i)$. This generally improves compression. For example, the last symbol $t_{n-1}$ has a frequency of one and does not need to be encoded at all. However, the saving is usually too small to make it worth while.
The last optimization can reduce the size of the second part below \( n\mathcal{H}_0(T) \) bits, which is thus not a strict lower bound.

The number of strings with the length \( n \) and the symbol counts \( n_1, \ldots, n_\sigma \) is given by the multinomial

\[
\binom{n}{n_1 \ldots n_\sigma}
\]

Thus the average code length for such strings has to be at least

\[
\log \left( \binom{n}{n_1 \ldots n_\sigma} \right) \text{ bits.}
\]

It can be shown that

\[
n\mathcal{H}_0(T) = \log \left( \binom{n}{n_1 \ldots n_\sigma} \right) + \mathcal{O}(\sigma \log n).
\]

Thus \( n\mathcal{H}_0(T) \) is close to the lower bound. In fact, the overhead \( \mathcal{O}(\sigma \log n) \) matches what is required for encoding the counts.
Adaptive Encoding

When decoding the symbol $t_i$, the decoder already knows the frequencies in $T[0..i]$ (or in $T[i+1..n]$ with ANS). If we use those frequencies to encode $t_i$, there is no need to store the symbol counts separately. This is the idea of adaptive coding.

However, we have to deal with the zero frequency problem. If $t_i = s$ is the first occurrence of the symbol $s$ in $T$, its frequency in $T[0..i]$ is zero. There are two ways to solve this problem:

- Use pseudocounts, i.e., add a positive constant $\alpha$ to each count. Then a first occurrence $t_i$ will have a frequency $\alpha / (i + \alpha \sigma)$. Usually $\alpha = 1$.
- Add a special escape symbol before each first occurrence. After an escape, the new symbol is encoded using a uniform probability over the symbols that have not occurred (or simply a fixed length code over the whole alphabet). The escape symbol itself has an artificial count, often a constant throughout the encoding.
Example

Let $T = \text{badada}$ and $\Sigma = \{a, b, c, d\}$. The frequencies assigned to the symbols are:

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>a</th>
<th>d</th>
<th>a</th>
<th>d</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>semi-adaptive</td>
<td>1/6</td>
<td>3/6</td>
<td>2/6</td>
<td>3/6</td>
<td>2/6</td>
<td>3/6</td>
</tr>
<tr>
<td>optimized</td>
<td>1/6</td>
<td>3/5</td>
<td>2/4</td>
<td>2/3</td>
<td>1/2</td>
<td>1/1</td>
</tr>
<tr>
<td>adaptive ($\alpha = 1$)</td>
<td>1/4</td>
<td>1/5</td>
<td>1/6</td>
<td>2/7</td>
<td>2/8</td>
<td>3/9</td>
</tr>
<tr>
<td>adaptive ($n_{\text{escape}} = 1$)</td>
<td>1/1 \cdot 1/4</td>
<td>1/2 \cdot 1/3</td>
<td>1/3 \cdot 1/2</td>
<td>1/4</td>
<td>1/5</td>
<td>2/6</td>
</tr>
</tbody>
</table>

When the symbol counts grow large, they can be renormalized by dividing by some constant. This has the effect of partially forgetting the past, since the symbols encountered after the renormalization have a bigger effect on the counts than those preceding the renormalization. This can make the model adapt better to local differences.
The constantly changing frequencies of adaptive models require some modifications to the entropy coders. Assume that the model maintains symbol counts $n_s$, $s \in \Sigma$, in the part of text known to the decoder as well as their sum $n = \sum_s n_s$.

For arithmetic coding and rANS, the changing probabilities are not a problem as such, but there are a couple of details to take care of:

- We need cumulative probabilities. Since a single increment of a count can change many cumulative counts, maintaining the cumulative counts can become expensive for a large alphabet. A simple optimization is to update the counts only every few steps.

- A probability is represented as an integer $p$ encoding the value $p/n$. In particular, the divider is now not a constant and not usually a power of two.

For a binary alphabet, cumulative counts are not a problem and there are special techniques for updating the probabilities using a fixed divider representation.
With Huffman and tANS coding, changing probabilities are more problematic and they are not usually used as adaptive encoders. However, there is an adaptive Huffman coding variant based on the fact that a small change to the frequencies is likely to have only a small effect on the code.

The idea is to maintain the Huffman tree:

- The leaves store the symbol counts.
- Each internal node stores the sum of its children’s counts.

All nodes are kept on a list in the order of their counts. Ties are broken so that siblings (children of the same parent) are next to each other. Whenever an increment causes a node \( u \) to be in a wrong place on this list, it is swapped with another node \( v \):

- \( v \) is chosen so that after swapping \( u \) and \( v \) on the list the list is in the correct order both before and after the increment.
- The subtrees rooted at \( u \) and \( v \) are simultaneously swapped in the tree.

The increments start at a leaf and are propagated upwards (to the new parent after a possible swap).
Example

Let $T = \text{abb} \ldots$ and $\Sigma = \{a, b, c, d\}$.

Starting with all counts being one and a balanced Huffman tree, processing the first two symbols $\text{ab}$ updates the counts but does not change the tree.

```
R  U  V  a  b  c  d
6  4  2  2  2  1  1
```

The third symbol $b$ causes the swap of the nodes $b$ and $V$.

```
R  U  b  a  V  c  d
7  4  3  2  2  1  1
```

It can be shown that this algorithm keeps the tree a Huffman tree. The swaps can be implemented in constant time. The total update time is proportional to the codeword length.
Higher Order Models

In natural language texts, there is often a strong correlation between adjacent letters. Higher order models take advantage of this and can achieve much better compression than zeroth order models.

Let $T = t_0 t_2 \ldots t_{n-1}$ be a text of length $n$ over an alphabet $\Sigma$ of size $\sigma$. Consider $T$ to be circular, i.e., $t_i = t_{i \mod n}$ for any integer $i$. For any string $w \in \Sigma^*$, let $n_w$ the number of occurrences of $w$ in $T$.

**Example**

If $T = \text{badada}$, then $n_{\text{da}} = 2$ and $n_{\text{ab}} = 1$.

**Definition**

The *k*th order empirical entropy of the text $T$ is

$$H_k(T) = - \sum_{w \in \Sigma^k} \frac{n_w}{n} \sum_{s \in \sigma} \frac{n_{ws}}{n_w} \log \frac{n_{ws}}{n_w}.$$ 

The value $n_{ws}/n_w$ is the frequency of $s$ in the *k*th order context $w$ and $n_w/n$ is the frequency of that context.
The circularity of the text makes the definition cleaner and simpler as we avoid complexities involving the ends of the text. For example, in a non-circular case the precise frequency of a $k$th order context $w$ should be $n_w/(n-k)$ or even $(n_w-1)/(n-k)$ when $w = T[n-k\ldots n-1]$.

The clean definition allows alternative formulations such as (exercise):

$$H_k(T) = \frac{1}{n} \left( \sum_{w \in \Sigma^k} n_w \log n_w - \sum_{w \in \Sigma^{k+1}} n_w \log n_w \right)$$

The difference between circular and non-circular version becomes clearer if we add a unique end-of-string symbol (EOS) to the end of the string.

- If $w$ does not contain EOS, $n_w$ is the same in circular and non-circular string.
- If $w$ contains EOS, either $n_w = 0$ or $n_w = 1$ and $\log n_w = 0$. In either case, the terms involving $n_w$ become 0. (In the equations, $0/0 = 0 \log 0 = 0$.)
- The remaining difference is the coefficient $1/n$ vs. $1/(n-k)$. 
A simple semiadaptive $k$th order encoding of $T$ consists of:

- the symbol counts $n_w$ for all $w \in \Sigma^{k+1}$ encoded in some way,
- the first $k$ symbols of $T$ using some simple encoding, and
- the rest of text symbols encoded with an entropy coder using as probabilities the symbol frequencies in their $k$th order contexts.

The size of the last part of the encoding is about $nH_k(T)$.

Notice that we do not need to store $n_w$ for $w \in \Sigma^k$ separately since $n_w = \sum_{s \in \Sigma} n_{ws}$.

A simple adaptive $k$th order model maintains $\sigma^k$ separate zeroth order models, one for each $w \in \Sigma^k$. Each symbol $t_i$ is encoded using the model for the context $T[i − k..i − 1]$ and then the model is updated just as in zeroth order compression.
Example

Let $T = \text{badadabada}$ and $\Sigma = \{a, b, c, d\}$. The frequencies of symbols in first order contexts are:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0/5</td>
<td>2/5</td>
<td>0/5</td>
<td>3/5</td>
</tr>
<tr>
<td>b</td>
<td>2/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
</tr>
<tr>
<td>c</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>d</td>
<td>3/3</td>
<td>0/3</td>
<td>0/3</td>
<td>0/3</td>
</tr>
</tbody>
</table>

The first order empirical entropy is

$$H_1(T) = -\frac{5}{10} \left( \frac{2}{5} \log \frac{2}{5} + \frac{3}{5} \log \frac{3}{5} \right) \approx 0.485$$

while $H_0(T) \approx 1.49$.

(When computing entropies, terms involving probabilities 0 and 1 can be ignored since $0 \log 0 = 1 \log 1 = 0$.)

The semiadaptive and adaptive (pseudocount $\alpha = 1$) models might assign the following frequencies to the symbols:

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>a</th>
<th>d</th>
<th>a</th>
<th>d</th>
<th>a</th>
<th>d</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>d</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>semiadaptive</td>
<td>1/4</td>
<td>2/2</td>
<td>3/5</td>
<td>3/3</td>
<td>3/5</td>
<td>3/3</td>
<td>2/5</td>
<td>2/2</td>
<td>3/5</td>
<td>3/3</td>
<td>3/3</td>
<td></td>
</tr>
<tr>
<td>adaptive</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>2/5</td>
<td>2/5</td>
<td>1/6</td>
<td>2/5</td>
<td>3/7</td>
<td>3/6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A problem with $k$th order models is the choice of the context length $k$. If $k$ is too small, the compression rate is not as good as it could be. If $k$ is too large, the overhead components start to dominate:

- the storage for the counts in semiadaptive compression
- the poor compression in the beginning for adaptive compression.

Furthermore, there may not exist a single best context length and a different $k$ should be used at different places.

One approach to address this problem is an adaptive method called prediction by partial matching (PPM):

- Use multiple context lengths $k \in \{0, \ldots, k_{\text{max}}\}$. To encode a symbol $t_i$, find the longest context $T[i - k..i - 1]$ that has occurred before.
- If $t_i$ has not occurred in that context, encode the escape symbol similarly to what we saw with zeroth order compression. Then reduce the context length by one, and try again. Keep reducing the context length until $t_i$ can be encoded in that context. Each reduction encodes a new escape symbol.

There are further details and variations.
There are other complex modelling techniques such as **dynamic Markov compression** that dynamically builds an automaton, where each state represents a certain set of contexts.

The most advanced technique is **context mixing**. The idea is to use multiple models and combine their predictions. The variations are endless with respect to what models are used and how the predictions are combined. Context mixing compressors achieve the best compression ratio in many compression benchmarks but are usually very slow.

Many of the most complex techniques operate on a binary source alphabet, which simplifies both modelling and entropy coding. Texts with a larger alphabet are often transformed into binary using a fixed-length encoding or Huffman coding. Then the probability of each bit is computed using a complex model and encoded, usually with arithmetic coding.