ECE 573 – DATA STRUCT & ALGS

Data Compression using Arithmetic Coding

Rutgers University

Zhongze Tang (zt67)
4-3-2018

Data Compression using Arithmetic Coding

Zhongze Tang (zt67)

Abstract

The world is full of data, so the algorithms and data structures that can represent or store data effectively play an essential role in the computer world. Arithmetic Coding is a form of entropy encoding used in lossless data compression. In this paper, we will first show you an example of Arithmetic Coding, then we will introduce how to implement it, and we will compare its compression ratio and performance with Huffman Coding.

1 Introduction

Lossless compression is a kind of techniques to reduce the quantity of data but keep the quality of data. There are many kinds of lossless compression algorithms like Huffman Coding, LZW, and DEFLATE. Arithmetic Coding is one of these algorithms. The basic idea of it is to convert a message composed of symbols (nearly always 8-bit ASCII characters) to a real number which is not less than 0 and less than one.

Another thing about Arithmetic Coding is that a model is needed to depict the symbols' characterization. The model reflects the probability of a symbol is in the message. If you have an accurate model, your compression result will be very close to optimally.

2 Example of Arithmetic Coding

To describe this algorithm more clearly, let's start with a simple example. Suppose we are going to encode "WE LOVE RU" using Arithmetic Coding, then we will get a table like this:

Symbols	Probability
SPACE	2/10
E	2/10
L	1/10
O	1/10
R	1/10
U	1/10
V	1/10
W	1/10

As mentioned before, we can call the table above a "model" because it reflects the probability of each symbol. The probabilities of symbols are known now; then we need to assign ranges to symbols along a

probability line, which is nominally [0, 1) The order of symbols on the line doesn't matter, as long as the encoder and decoder are in the same manner. The table may look like this:

Symbol	Probability	Range
SPACE	2/10	[0, 0.2)
E	2/10	[0.2, 0.4)
L	1/10	[0.4, 0.5)
O	1/10	[0.5, 0.6)
R	1/10	[0.6, 0.7)
U	1/10	[0.7, 0.8)
V	1/10	[0.8, 0.9)
W	1/10	[0.9, 1.0)

Each symbol is assigned a part of the [0, 1) range, and the length of the range depends on the probability of it. The first symbol in the message is "W," which is the most important symbol because the output of the algorithm should be greater than or equal to 0.9 and less than 1.0 to decode the first symbol properly. To be more specific, to encode a symbol is to find which range does it fall in. So after encoding the first symbol, we find that the upper bound of the output is 1.0 and the lower bound of the output is 0.9. The next symbol, "E," owns the range of [0.2, 0.4). It means that it holds the range of [0.2, 0.4) in the output range we have already get. What we will get is [0.92, 0.94) after we encode the second symbol.

The algorithm to encode the message is shown as below:

```
low = 0.0
high = 1.0
WHILE input symbol != null DO
    read in an input symbol
    range = high - low
    high = low + range * get_high(symbol)
    low = low + range * get_low(symbol)
END
RETURN low
```

Keep tracing in this encoding method; we will get another table:

Symbols	low	high (not included)
W	0.9	1
E	0.92	0.94
SPACE	0.920	0.924
L	0.9216	0.922
O	0.9218	0.92184
V	0.921832	0.921836
E	0.9218328	0.9218336
SPACE	0.92183280	0.92183296
R	0.921832896	0.921832912
U	0.9218329072	0.9218329088

The final Lower Bound, 0.9218329072, is the ultimate output, which uniquely encodes the message “WE LOVE RU” using our current encoding strategy.

As we all know, when we talk about coding, we have to talk about encoding and decoding at the same time. Given the encoding method above, it is easy to find out how to decode the output. Here is the decoding algorithm:

```
read in the encoded_number
DO
    find the symbol whose range includes the encoded_number
    OUTPUT the symbol
    encoded_number = encoded_number - get_low(symbol)
    range = get_high(symbol) - get_low(symbol)
    encoded_number = encoded_number / range
UNTIL encoded_number == 0.0
```

Based on this algorithm, the decoding of 0.9218329072 looks like the table below:

encoded_number	output_symbol	low	high	Range
0.9218329072	W	0.9	1	0.1
0.218329072	E	0.2	0.4	0.2
0.09164536	SPACE	0	0.2	0.2
0.4582268	L	0.4	0.5	0.1
0.582268	O	0.5	0.6	0.1
0.82268	V	0.8	0.9	0.1

0.2268	E	0.2	0.4	0.2
0.134	SPACE	0	0.2	0.2
0.67	R	0.6	0.7	0.1
0.7	U	0.7	0.8	0.1
0				

3 Implementation

The example using Arithmetic Coding above is simple. However, if you think more deeply, you will find that the algorithm is entirely impractical on computers because the accuracy of double type in programming languages is limited, which means when low and high are close enough, the difference of them is too small to be represented by a double number. After that, the algorithm cannot continue to work at all. So, in this section, we will introduce how to implement a good Arithmetic Coding algorithm, using integer numbers instead of double numbers.

3.1 Encoder

First of all, we set low and high like this:

```
unsigned int high = 0xFFFFFFFFU;
unsigned int low = 0;
```

They are not real int numbers.Instead, they're decimals, which means high is 0.FFFFFFFF in hex or 0.1111…1111 in binary, and low is 0.0. However, in Section 2, high is equal to 1.0. So we have to consider high as 0.FFFF…, with infinite Fs behind it (or 1s, in binary). We only see eight of them due to the limitation of memory, and later the remaining Fs will be shifted into memory.

Second, there are some principles, which will be very helpful when understanding the algorithm below:

- Low is always smaller than high.
- Low never decreases while high never increases.

To deal with binary numbers with arbitrary length, our algorithm has to process the numbers bit by bit. Then we will get our new algorithm:

```
unsigned int high = 0xFFFFFFFFU;
unsigned int low = 0;
int pending_bits = 0;
char c;
while ( input >> c ) {
  int range = high - low + 1;

  prob p = model.getProbability(c);
  //structure prob is used to describe the probability.
```

//Take "W" (with range [0.9, 1)) above as an example, it will return {9, 10, 10}, which are lower, upper and denominator respectively.

```
  high = low + (range * p.upper)/p.denominator;
  low = low + (range * p.lower)/p.denominator;
  for ( ; ; ) {
    if ( high < 0x80000000U ) {
    //In this case, MSB of high is 0, so as the low because low is always smaller than high.
    //This means we no longer need the first bit of high, so we just output and discard it.
      output_bit_plus_pending( 0, pending_bits );
      low <<= 1;
      high << = 1;
      high |= 1; //shift in a 1 into the LSB of high, because we have infinite 1s waiting for
being shifted
    } else if ( low >= 0x80000000U ) {
      //In this case, MSB of low is 1, so as the high.
      //No longer need the first bit low, output and discard it.
      output_bit_plus_pending( 1, pending_bits );
      low <<= 1;
      high << = 1;
      high |= 1;   //shift in a 1 into the LSB of high, because we have infinite 1s waiting for
being shifted
    } else if ( low >= 0x40000000 && high < 0xC0000000U )
      //In this case, check the second MSB (for low, its first two MSBs are 01, and for high are 10) to
avoid the convergence situation that high and low become the same like 0x80000000 or 0x7FFFFFFF.
      //If converge, we will always get the same result; our algorithm cannot continue to work.
      //At the same time, when this near-convergence case happens, both of low and high will be
0x01111... Or 0x10000... The second MSB will be the opposite to the first MSB. So we do the count and
discard the second MSB and shift the last 30 bits left to avoid the convergence situation.
      pending_bits++;
      low << = 1;
      low &= 0x7FFFFFFF;
      high << = 1;
      high |= 0x80000001;
    } else
      break;
  }
}

void output_bit_plus_pending(bool bit, int &pending_bits)
{
  //When we really need to output a bit, we output the bit first and then output the opposite bit of it
%pending_bits% times.
  output_bit( bit );
  while ( pending_bits-- )
```

```
    output_bit( !bit );
}
```

3.2 Decoder

The decoder is similar to the encoder above. The variable value is another “infinite” variable, and it contains the 32 bits from the encoded messages that we are processing. We use a variable called count to find out where the char falls on the probability line, and use a function called getChar to find what symbol it is. We can see that the high and low we generate when calculating the encoded messages are useless in the decoding algorithm.

```
unsigned int high = 0xFFFFFFFFU;
unsigned int low = 0;
unsigned int value = 0;
for ( int i = 0 ; i < 32 ; i++ ) {
  value <<= 1;
  value += m_input.get_bit() ? 1 : 0;
}
for ( ; ; ) {
  unsigned int range = high - low + 1;
  unsigned int count =  ((value - low + 1) * m_model.getCount() - 1 ) /
range;
  int c;
  prob p = m_model.getChar( count, c );
  if ( c == 256 )
    break;
  m_output.putByte(c);
  high = low + (range*p.high)/p.count -1;
  low = low + (range*p.low)/p.count;
  for( ; ; ) {
    if ( low >= 0x80000000U || high < 0x80000000U ) {
      low <<= 1;
      high <<= 1;
      high |= 1;
      value <<= 1;
      value += m_input.get_bit() ? 1 : 0;
    } else if ( low >= 0x40000000 && high < 0xC0000000U ) {
      low << = 1;
      low &= 0x7FFFFFFF;
      high <<= 1;
      high |= 0x80000001;
      value <<= 1;
      value += m_input.get_bit() ? 1 : 0;
    } else
      break;
```

```
  }
}
```

3.3 Models

The probabilities in the model can have nothing to do with the actual appearance frequencies in the message to be encoded. The Arithmetic Coding algorithm will still work if we use an evenly distributed model. However, the compression ratio will be lower if the model fits the message very well.

There are two kinds of models: fixed models and adaptive models. In fixed models, the probabilities of symbols are fixed. The exact fixed model can be calculated and sent before starting encoding the message. And it is found by Cleary and Witten [4] that in general, its performance will not be better than adaptive models.

An adaptive model reflects the frequencies counted so far in the message. Firstly, all frequencies should be the same, and they will be updated both in encoding and decoding. It's an entirely expensive operation to update our models when encoding or decoding because we have to maintain cumulative totals.

4 Analysis and Comparison with Huffman Coding

The time complexity of this algorithm depends on the implementation, and the analysis of it is very complex. Simply put, if we use sequential search on sorted symbols, the order of it is O(N). And if we use an adaptive model with bisection tree, the order of it is O(lg(N)) [2].

The test Arithmetic Coding programme uses a simple adaptive model. All the symbols in the model have a frequency of one in the beginning, and their frequencies will be updated along with the process of encoding or decoding, which is helpful to reduce the output length. Both of the programmes are written in JAVA.

TextFiles/Algorithms	Arithmetic Coding [8]			Huffman Coding [9]		
	Compress Time (ms)	Compression Ratio	Decompress Time (ms)	Compress Time (ms)	Compression Ratio	Decompress Time (ms)
HuckleBerry.txt	53	57.88%	47	62	58.37%	59
PrideAndPrejudice.txt	56	55.99%	45	49	56.25%	45
test.jpeg	N/A	99.65%	N/A	N/A	101.34%	N/A

We can see that Arithmetic Coding performs better than Huffman Coding in compression ratio when compressing text files. As for the Compress Time and Decompress Time, the data tells us they do not have too much reference value because the time depends on both the implementation method of the

algorithm and what is going to be compressed. However, when these two algorithms try to compress a JPEG file, their performances are really bad, which indicates that they are not suitable for picture compression.

5 Conclusion

In this paper, we use an example to show what the Arithmetic Coding look like, and then introduce it's implementation, at last, we compare its performance to Huffman Coding. We can see that it's a good compression algorithm for text compression. However, it is not used that much in commercial products nowadays, and many companies prefer to use Huffman Coding or other compression algorithms. The main reason is that the IBM company owns some patents of it. Anyway, it's a great and interesting algorithm that is worth to learn.

6 Reference

1. Witten, Ian H., Neal, Radford M., and Cleary, John G. (1987) *Arithmetic Coding for Data Compression*, Communications of the ACM, June, pp 520-540.

2. Said, Amir, *Introduction to Arithmetic Coding - Theory and Practice*, Lossless Compression Handbook

3. Sedgewick, Robert, and Wayne, Kevin (2011) *Data Compression*, Algorithm 4th Edition, pp 810-845.

4. Cleary, J.C., and Witten, I.H. *A comparison of enumerative and adaptive codes.* IEEE Trans. Inf. Theory IT-30,2 (Mar. 1984). 306-315.

5. https://en.wikipedia.org/wiki/Arithmetic_coding

6. https://en.wikipedia.org/wiki/Huffman_coding

7. http://marknelson.us/2014/10/19/data-compression-with-arithmetic-coding/

8. https://www.nayuki.io/page/reference-arithmetic-coding

9. https://algs4.cs.princeton.edu/55compression/Huffman.java.html