Chapter 3

Arithmetic coding

This chapter describes arithmetic coding, an algorithm that produces, given a sequence of input symbols and associated probability distributions, a near-optimally compressed output sequence whose length is within 2 bits of the input sequence's Shannon information content. The appeal of such a method is that the tasks of data modelling and code generation can be cleanly separated, allowing compression methods to be constructed through data modelling.

This chapter proposes an architecture for a modular compression library based on arithmetic coding, which allows compression algorithms to be designed in a coherent, modular and compositional manner.

For concreteness, this chapter includes a complete implementation of an arithmetic coder as JAVA source code, and describes how to interface it to various fundamental probability distributions that serve as building blocks for the compression methods in later chapters.

3.1 Introduction

3.1.1 Information content and information entropy

Data compression and probability theory are fundamentally linked by the relationship of the probability distribution P, and the expected amount of information required to convey a random P-distributed object. Given P, an object x has an *information content* of $h_P(x)$ bits, where:

$$h_P(x) \;=\; \log_2 \frac{1}{P(x)}. \tag{3.1}$$

Averaging over all possible messages $x \in \mathcal{X}$ yields the *expected* information content of the probability distribution P, called its *information entropy*:

$$\mathrm{H}(P) \;=\; \sum_{x \in \mathcal{X}} P(x) \cdot h_P(x) \;=\; \sum_{x \in \mathcal{X}} P(x) \cdot \log_2 \frac{1}{P(x)} \tag{3.2}$$

The information entropy $\mathrm{H}(P)$ of any finite discrete distribution P is highest when P is uniform, i.e. when all elements have equal probability mass. Both information content and information entropy are measured in *bits*, where one bit is the amount of information needed to convey a choice between two equiprobable options. N bits can convey a choice among 2^N equiprobable options, and a fair choice among K equiprobable options has an information content of $\log_2 K$ bits.

The information entropy of a distribution P marks the theoretical limit down to which P-distributed messages can, on average, be losslessly compressed. No code can expect to communicate N messages in fewer than $N \cdot \mathrm{H}(P)$ bits of information, if the messages are random, independent, and distributed according to P.

3.1.2 The relationship of codes and distributions

As mentioned in section 1.2, every compression code necessarily assumes, at least implicitly, a distribution over input objects. Consider a uniquely decodable code C that maps inputs $x \in \mathcal{X}$ to code words $C(x) \in \mathcal{A}^*$, where $\mathcal{A}$ is some finite alphabet. The compression effectiveness of code C (given $|\mathcal{A}|$, the size of the alphabet) is completely determined by the *lengths* of the code words, i.e. by the set of mappings from x to $|C(x)|$.

These code lengths can be interpreted as a probability distribution P_C over input objects:

$$P_C(x) = \frac{1}{Z} \cdot |\mathcal{A}|^{-|C(x)|} \tag{3.3}$$

where Z normalises the probabilities to unity. Code C works best on objects that are distributed according to P_C. However, C might not be the *best* possible code for P_C: it might be a wasteful code whose code words are longer than necessary.

A lossless compression code is *optimal* for a given probability distribution if the expected code length for a random message is minimal. The most effective code word lengths, for a given distribution P, would be:

$$|C_{\mathrm{OPT}}(x)| \;=\; \log_{|\mathcal{A}|} \frac{1}{P(x)} \;=\; \frac{1}{\log_2 |\mathcal{A}|} \cdot h_P(x). \tag{3.4}$$

As these quantities are non-integer, it is not generally possible to find code words that match these optimal lengths exactly. However, it is possible to construct code words whose integer lengths are "close enough" to the optimal lengths to guarantee that the *expected code word length* is within one symbol of the theoretical optimum:

$$\underbrace{\frac{1}{\log_2 |\mathcal{A}|} \cdot \mathrm{H}(P)}_{\text{optimal length}} \;\leq\; \underbrace{\sum_{x \in \mathcal{X}} P(x) \cdot |C(x)|}_{\text{expected length}} \;<\; \underbrace{\frac{1}{\log_2 |\mathcal{A}|} \cdot \mathrm{H}(P) + 1}_{\text{optimal length+1}}. \tag{3.5}$$

For details and proof, see e.g. Cover and Thomas (2006), chapter 5.

3.1.3 Algorithms for building optimal compression codes

One method for constructing an optimal set of code words that satisfies the bound given in (3.5) is the Huffman algorithm (as described in section 2.1.1). A Huffman code for P has an expected code word length that's within one bit of $\mathrm{H}(P)$, the Shannon entropy of P. However, generating a Huffman code requires enumerating all possible messages in advance, which makes its use impractical for large or unbounded sets of messages, such as files of human text.[1]

Fortunately, there are techniques that can compress single messages efficiently without considering all possible messages beforehand. An *arithmetic coding* algorithm (or "arithmetic coder", for short) is an example of such a method. Arithmetic coders progressively construct, for any message x and associated probability distribution P, a single code word whose length is within 2 bits of $h_P(x)$, the Shannon information content of the input. The expected output length of an arithmetic coder is therefore at most 2 bits worse than that of a Huffman code.

An arithmetic coder can be used whenever the following requirements are fulfilled:

- the input message can be broken up into a series of discrete decisions or symbols,
- the probability distribution over messages can be factorised into a product of univariate conditional distributions (one for each symbol or decision),
- the *cumulative distribution* of each of these univariate distributions can be computed.

Section 3.2 describes the principal idea behind arithmetic coding, and includes a concrete implementation of the algorithm.

Huffman coding versus arithmetic coding. An advantage of arithmetic coding is its ability to compute the code word for a message without having to consider all other possible messages or code words. Huffman coding, by comparison, requires comparing the probability mass of all possible messages. Disadvantages of arithmetic coding include that the generated output can be longer than that of a Huffman code. Consider, for example, encoding exactly one of two possible messages $\{\mathtt{A}, \mathtt{B}\}$, where $P(\mathtt{A}) = 0.999$ and $P(\mathtt{B}) = 0.001$. The information contents of the two messages are $h_P(\mathtt{A}) \approx 0.001$ bits and $h_P(\mathtt{B}) \approx 9.966$ bits. The Huffman algorithm allocates code words of length 1 to each message, which is optimal if only one

[1] Note that a Huffman code over *source symbols* is optimal only for encoding a single symbol. Encoding a sequence of symbols by concatenating those Huffman code words is not generally optimal (except when the symbols in the sequence are independent and identically distributed with known and strictly dyadic symbol probabilities).

message is to be transmitted. An arithmetic coder requires 1 bit of output to communicate message `A`, and ca. 10 bits for message `B`.

Assuming that only a single message is being transmitted, and that all messages and their probabilities $P(x)$ can be enumerated, the Huffman algorithm is guaranteed to produce an optimal code for P-distributed messages. If the number of possible messages is very small, and if the probability distribution over messages doesn't change, constructing a Huffman code is most likely the best thing to do; in most other situations (such as compressing files of human text), arithmetic coding is the tool of choice.

Arithmetic coding can be viewed as a natural generalisation of Huffman coding: when used on dyadic input distributions, the output produced by an arithmetic coder is identical to that of a Huffman coder. For a concrete demonstration of the similarity of both algorithms see e.g. the article by Bloom (2010).

A note on the term 'entropy coding'. In the compression literature, there seems to be wide-spread and inconsistent use of the term *entropy coding*. For lack of a precise definition, use of this term is best avoided entirely; more information is given on page 41.

3.2 How an arithmetic coder operates

Arithmetic coding is a general coding method that compresses a sequence of input symbols $x_1 \ldots x_N$, given associated probability distributions $P_1 \ldots P_N$, to a sequence of binary digits whose length is within 2 bits of the sequence's information content. The probability distributions $P_1 \ldots P_N$ must be available to both the sender and the receiver.

In a nutshell, arithmetic coding successively maps each input symbol x_n to an interval $R_n \subseteq R_{n-1}$ in proportion to its probability mass $P_n(x_n)$, starting from $R_0 = [0, 1)$. The encoded file is the binary representation of the shortest number $r \in \mathbb{R}$ inside the final interval R_N.

3.2.1 Mapping a sequence of symbols to nested regions

Consider a sequence of symbols $x_1 \ldots x_N$ with associated probability distributions $P_1 \ldots P_N$. The joint probability mass of the sequence is:

$$\Pr(x_1 \ldots x_N \mid P_1 \ldots P_N) \;=\; \prod_{n=1}^{N} P_n(x_n) \tag{3.6}$$

An arithmetic coder can map this sequence to a series of nested regions of the unit interval, one symbol at a time. The process starts out with the interval $R_0 = [0, 1)$. For some arbitrary ordering of the symbols in the alphabet $\mathcal{X}$, this interval is partitioned into non-overlapping

Reasons for avoiding the term 'entropy coding'

The first occurrence of the term entropy coding I could find appears in a paper by Goblick and Holsinger (1967) on the digitization of analog signals. The authors give a definition with local scope, in the context of a particular encoding method.
An alternative source of the term might be a paper by O'Neal (1967) on the same topic, which contains the following statement:

> [...] *Therefore, the technique of entropy coding* [5] *(also called "Shannon–Fano coding" or "Huffman coding") can be used either to increase the S/N ratio for a given bit rate or to decrease the bit rate for a given S/N ratio.* [...] where [5] refers to a book by Fano (1961).

In 1971, the same author published an article with "entropy coding" in the paper title. The text contains the plural: *"entropy coding techniques (Huffman or Shannon–Fano coding)"*. No reference or definition is given.
A paper by Berger et al. (1972) makes heavy use of the term "entropy coding", but gives no explicit reference to its definition or origin. However, the authors were aware of the paper by Goblick and Holsinger (1967), as it is referenced in their bibliography. From 1972 onwards, many papers contain the term "entropy coding", possibly copying O'Neal's use of the term.
Where definitions do show up, they are rather vague. For example, ITU recommendation H.82 (ITU-T, 1993) defines an "entropy coder" to be *any lossless method for compressing or decompressing data*. Similarly, the book by Wiegand and Schwarz (2011) defines entropy coding to be a synonym for "lossless coding" and "noiseless coding". No explanations or references are given.

What the term "entropy coding" could mean:

- *"any coding method whose expected rate of compression equals the entropy of the input distribution."* But a better criterion than the expected rate of compression is the actual compression: a coding method whose output length is equal to the Shannon information content of the input. Also, it's possible to construct optimal codes for sources that do not even have a finite entropy.

- *"a coding method whose distribution over* output symbols *has high entropy."* Such a definition would not even imply compression; typical encryption methods, for example, also produce output symbols with uniform probability.

In summary, "entropy coding" doesn't seem a clearly defined or distinguished concept. Leading text books, e.g. by Cover and Thomas (1991, 2006) or MacKay (2003), do not contain the term "entropy coding", and discuss optimal codes instead. Following their example, the term entropy coding is not used further in this thesis.

regions, one for each symbol $x \in \mathcal{X}$, in proportion to the mass $P_1(x)$ of its symbol. Once the arithmetic coder knows the first symbol x_1, it chooses the region R_1 matching that symbol, and discards all information about the other regions.

Similarly, the process continues for the remaining symbols. At step n, the previously computed region R_{n-1} is divided into subintervals according to P_n, and the next region R_n is selected by the next symbol x_n.

The final region R_N contains the sequence of choices made for each symbol, such that the size of R_N matches the joint probability mass of all symbols in the sequence, as given in equation (3.6). The key observation is that knowing the final region R_N and the probability distributions P_1 ... P_N is sufficient for reconstructing the original sequence x_1 ... x_N.

3.2.2 Mapping a region to a sequence of symbols

Given the final region R_N and the symbol distributions P_1 ... P_N, the original sequence x_1 ... x_N can be recovered as follows. Starting from $n = 1$ and the unit interval $[0, 1)$, each symbol x_n can be recovered by partitioning the current interval into regions according to P_n, and selecting the region that contains R_N as a subregion. The label of the chosen region is the original symbol x_n. The process continues recursively from the previously chosen region, until all N symbols are recovered.

For the recovery process to work, only a single point (or subregion) inside the final region R_N needs to be known, rather than the exact region boundaries of R_N. In this case, the first N symbols can still be reconstructed perfectly, but N itself cannot. Unless the sequence length N is known in advance, it must be encoded separately, e.g. by using a special `EOF` symbol to mark the termination of the sequence. (Methods of encoding sequence termination are discussed in section 5.7.2.)

Adaptive distributions. Arithmetic coding allows a different distribution to be used for each symbol in the sequence. Furthermore, the probability distribution P_n at timestep n is allowed to depend on the previous symbols x_1 ... x_{n-1}, because P_n is only needed after x_{n-1} has been decoded. This property makes it possible to use adaptive distributions, e.g. distributions that gradually adjust by learning from previous symbols in the sequence. Adaptive compression techniques are described in more detail in chapter 4, and find use in state-of-the-art sequence compressors (such as those investigated in chapter 6).

3.2.3 Encoding a region as a sequence of binary digits

Having shown how to represent a sequence of symbols with a region R_N, and how to recover the sequence again from this region, it remains to be shown how to *encode* R_N to an output

sequence of binary digits. The method is beautiful and simple: we can apply the procedure from section 3.2.2, but instead of using the input alphabet $\mathcal{X}$ and distributions P_1 ... P_N, we use the binary alphabet $\{0, 1\}$ and uniform probabilities $U(0) = U(1) = \frac{1}{2}$ for each subregion.

At every step of this procedure, the current region (again, starting from $[0, 1)$) is subdivided into two equal halves, one for digit 0, and one for digit 1. If the known target region R_N lies completely within the 0-region, a 0 is written, and the 0-region is used as the enclosing region for the next coding step. Similarly, when R_N lies within the 1-region, a 1 is written. The procedure repeats until the current region lies completely inside the target region R_N, resulting in M binary output digits s_1 ... s_M. It is then possible to reconstruct a subregion of R_N from s_1 ... s_M (using U_1 ... U_M), and from there also the original sequence x_1 ... x_N (using P_1 ... P_N).

More generally, arithmetic coding can be used to compactly translate a sequence of values from a *source distribution* to a sequence of values distributed according to a *target distribution*. (This slightly more general functionality is used e.g. in section 8.1 to compress messages to output alphabets with unequal symbol lengths.)

Practical arithmetic coding algorithms can perform this translation gradually, keeping only as much state information as is required to produce the next output symbol.

3.3 Concrete implementation of arithmetic coding

In practice, arithmetic coders are typically implemented using fixed width integer arithmetic. A concrete implementation of such a fixed-precision coder was published by Witten, Neal and Cleary (1987), and developed further by Moffat, Neal and Witten (1998). The arithmetic coder proposed in this chapter is based on their work; its full source code is shown on pages 47–48.

Background. In a fixed-precision arithmetic coder, the interval $[0, 1)$ is represented by a range of integers from 0 to $2^b - 1$, where b is the bitwidth of available integer variables. The coder for this chapter was written in JAVA and uses variables of type `long` (64 bits wide).

The internal state of an arithmetic coder is the current region R_n, which in a fixed-precision implementation is represented with two integers, e.g. L (the low pointer) and R (the region width). Fixed-width coders typically also store an additional quantity named `bits_waiting`, which is explained in detail by Moffat et al. (1998). These internal quantities are modified whenever a new source symbol is read from the input, and whenever a compressed symbol is written to the output.

Innovations of the proposed coder. When compressing a sequence with an arithmetic coder, every input symbol x_n must be accompanied by a distribution P_n; this distribution is

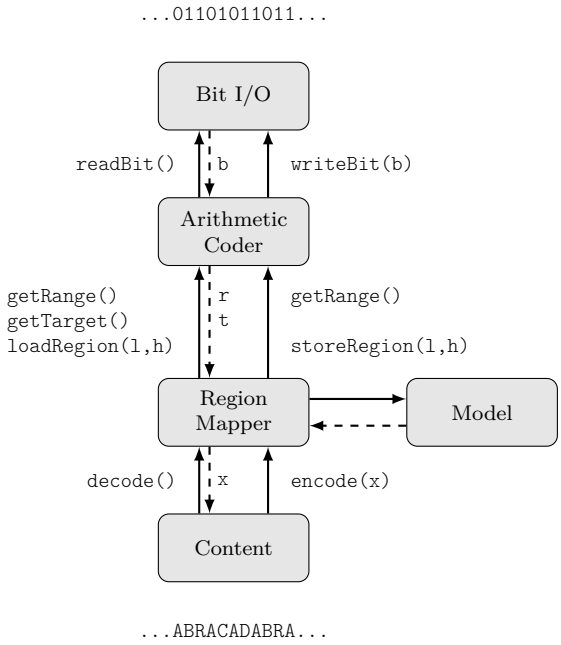

Figure 3.1: Proposed architecture of compression algorithms based on probabilistic models and arithmetic coding. The *region mapper* uses the model's predictive distribution to translate encoding and decoding requests into calls to the arithmetic coder. The components may be separate or combined entities.

used to partition the current region into subregions. The arithmetic coder proposed in this chapter never handles symbols or probabilities explicitly, and instead relies on the implementation of the probabilistic source model to translate symbols x_n and symbol distributions P_n to discrete regions.

This design choice allows probability distributions to offer a simple function interface for compressing and decompressing values, without the need to expose the inner workings of the arithmetic coder. For example, encoding a value `x` with a distribution P can be expressed as P`.encode(x,ec)`, where `ec` is an instance of an arithmetic *encoder*. Similarly, the corresponding inverse operation is given by `x =` P`.decode(dc)` where `dc` is an instance of an arithmetic *decoder*. P need not be a distribution over symbols, it could be a distribution over numbers, vectors, strings, sets or any type of object. Figure 3.1 shows a graphical representation of the proposed interface.

The implementation of the probability distribution P must provide the `encode` and `decode` functions: these two functions are responsible for mapping any element x to a discrete region whose size is approximately proportional to $P(x)$.[2] The region boundaries are communicated to an arithmetic coder instance using the functions shown in Table 3.1.

[2]At least implicitly, the `encode` and `decode` functions necessarily compute a discrete form of P_Σ (the cumulative distribution of P). The implementation must ensure that every input x with positive probability mass $P(x) > 0$ is mapped to a region of non-zero width, as those inputs would otherwise become unencodable.

Use	Operation	Description
ENC+DEC	`long getRange()`	Returns `R`, the current range of the coder.
ENC	`void storeRegion(l,h)`	Zooms into the discrete region `(l,h)`, and writes compressed output bits if possible. The region is specified with integers `l`, the lower (inclusive) point of the region, and `h`, the upper (exclusive) point.
DEC	`long getTarget()`	Returns an integer between 0 (inclusive) and `R - L` (exclusive) that allows the decoder to identify the next region.
DEC	`void loadRegion(l,h)`	Zooms into the discrete region `(l,h)`, and reads compressed bits from the input if necessary.

Table 3.1: Components of the arithmetic encoding and decoding interface. Probabilistic model implementations use these functions to communicate discrete regions for encoding and decoding. Code listing 3.5 shows how a model's `encode` and `decode` methods might be structured.

The design of suitable `encode` and `decode` functions is not always trivial; for example, multivariate distributions and distributions with infinite support may require special treatment. A selection of useful techniques can be found in section 3.4. Implementations for many common probability distributions are included in the compression library written for this thesis.

A historically prominent application of arithmetic coding is the encoding of symbols whose probability mass is computed from their empirical occurrence counts, for example as follows:

$$P(x_N = x \mid x_1 \ldots x_{N-1}) = \frac{\#[x_n = x]_{n=1}^{N-1} + 1}{N + |\mathcal{X}|}. \tag{3.7}$$

where $|\mathcal{X}|$ is the size of the symbol alphabet. This application presumably motivated the design of the procedure `narrow_region(l,h,t)` in the Witten–Neal–Cleary coder, which computes the next region by scaling cumulative counts `l` and `h` relative to a total count `t`.

Of course, adaptive symbol models are not limited to the form shown in (3.7), and symbol probabilities are not generally integer fractions. For these reasons, I advocate the more general interface shown in Table 3.1, as it allows models greater control over the computation of regions. For completeness and convenience, traditional versions of `storeRegion`, `loadRegion` and `getTarget` that include built-in scaling are given in code listing 3.4.[3]

[3]The original form of the PPM algorithm by Cleary and Witten (1984a) can be implemented entirely using these traditional versions. An in-depth treatment of PPM is given in chapter 6.

Arithmetic Coding: example application

```
public class ABC {

  /** Local state */
  int na = 1;
  int nb = 1;
  int nc = 1;
  int all = 3;

  /** Records a symbol observation. */
  public void learn(char c) {
    switch(c) {
      case 'A': na++; break;
      case 'B': nb++; break;
      case 'C': nc++; break;
      default : throw new IllegalArgumentException();
    }
    all++;
  }

  /** Encodes a symbol. */
  public void encode(char c, Encoder ec) {
    switch(c) {
      case 'A': ec.storeRegion(0, na, all); break;
      case 'B': ec.storeRegion(na, na+nb, all); break;
      case 'C': ec.storeRegion(na+nb, all, all); break;
      default : throw new IllegalArgumentException();
    }
  }

  /** Decodes a symbol. */
  public char decode(Decoder dc) {
    long t = dc.getTarget(all);
    if (t >= 0) {
      if (t < na) {
        dc.loadRegion(0, na, all);
        return 'A';
      } else
      if (t < na+nb) {
        dc.loadRegion(na, na+nb, all);
        return 'B';
      } else
      if (t < all) {
        dc.loadRegion(na+nb, all, all);
        return 'C';
      }
    }
    throw new IllegalStateException();
  }

} // end of class ABC
```

```
public class ABCTest {

    public static void main(String[] args) throws Exception {
      /* Compressing a sequence of ternary values */
      char[] data = new char[] { 'B','B','B','B','A','B','B','B','C' };
      ABC abc = new ABC();
      Arith ac = new Arith();
      BitWriter bw = IOTools.getBitWriter("output.bin");
      ac.start_encode(bw);
      for (char x : data) {
        abc.encode(x,ac);
        abc.learn(x);
      }
      ac.finish_encode(); // writes "0111 1001 0100 1"
      bw.close();         // appends "000" to fill the last byte

      /* Decompressing the sequence */
      abc = new ABC();  ac = new Arith();
      BitReader br = IOTools.getBitReader("output.bin");
      ac.start_decode(br);
      String s="";  // we append decompressed symbols here
      char x;
      do {
        x = abc.decode(ac);
        abc.learn(x);
        s += x;  // append symbol x
      } while (x != 'C');
      ac.finish_decode();
      br.close();
      System.out.println("Decoded: "+s); // prints "BBBBABBBC"
    }

} // end of class ABCTest
```

Code listing 3.1: An example of arithmetic coding used to compress a sequence of ternary values $x_n \in \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$. The class `ABC` implements three methods: `encode`, `decode`, and `learn`. These methods are called in the class `ABCTest` for adaptively compressing a fixed sequence to an external file `output.bin`, and then decompressing this file to recover the original sequence.

Arithmetic Coding

CONSTANTS

```
/** Number of bits available. */
final long b = Long.SIZE - 2;
/** Index of lower quarter. */
final long lb = (long) 1 << (b-2);
/** Index of midpoint. */
final long hb = (long) 1 << (b-1);
/** Index of top point. */
final long tb = ((long) 1 << b) - 1;
/** Mask of b 1-bits. */
final long mask = tb;
```

LOCAL STATE

```
/** Current range of coding interval. */
long R;
/** Low index of coding interval. */
long L;
/** Target location in coding interval. */
long D;             // Decoder only
/** Number of opposite-valued bits queued. */
long bits_waiting; // Encoder only

BitWriter output = null;
BitReader  input = null;
```

MAIN PROCEDURES

```
/** Outputs encoder's processed bits. */
void output_bits() {
  while (R <= lb) {
    if (L+R <= hb) {
      output_all((byte) 0);
    } else
    if (L >= hb) {
      output_all((byte) 1);
      L = L - hb;
    } else {
      bits_waiting++;
      L = L - lb;
    }
    L <<= 1;  R <<= 1;  // zoom in
  }
}

/** Writes a bit, followed by bits_waiting
  * bits of opposite value. */
void output_all(byte bit) {
  output.writeBit(bit);
  while (bits_waiting > 0) {
    output.writeBit((byte) (1 - bit));
    bits_waiting--;
  }
}

/** Sets a region. */
void narrow_region(long l, long h) {
  L = L + l;   // CAUTION: l, not 1
  R = h - l;
}
```

```
/** Discards decoder's processed bits. */
void discard_bits() {
  while (R <= lb) {
    if (L >= hb) {
      L -= hb; D -= hb;
    } else
    if (L+R <= hb) {
      // in lower half: nothing to do
    } else {
      L -= lb; D -= lb;
    }
    L <<= 1; R <<= 1;  // zoom in
    D <<= 1; D &= mask; D += input.readBit();
  }
}
```

```
/** Loads a region. */
public void loadRegion(long l, long h) {
  narrow_region(l,h);
  discard_bits();
}
```

```
/** Returns a target pointer. */
public long getTarget() { return D-L; }
```

```
/** Returns the coding range. */
public long getRange() { return R; }
```

```
/** Encodes a region. */
public void storeRegion(long l, long h) {
  narrow_region(l,h);
  output_bits();
}
```

Code listing 3.2: JAVA source code of the main operational procedures in an arithmetic coder. The shaded methods are intended for external use.

Arithmetic Coding: starting and stopping

STARTING

```
/** Starts an encoding process. */
void start_encode(BitWriter output) {
  this.output = output;
  L = 0;   // lowest possible point
  R = tb;  // full range
  bits_waiting = 0;
}
```

```
/** Starts a decoding process. */
void start_decode(BitReader input) {
  this.input = input;
  D = 0;
  // fill data pointer with bits
  for (int k=0; k<b; k++) {
    D <<= 1;
    D += input.readBit();
  }
  L = 0;
  R = tb; // WNC use "tb", MNW use "hb"
}
```

STOPPING

```
/** Finishes an encoding process. */
void finish_encode() {
  while (true) {
    if (L + (R>>1) >= hb) {
      output_all((byte) 1);
      if (L < hb) {
        R -= hb - L;   L  = 0;
      } else {
        L -= hb;
      }
    } else {
      output_all((byte) 0);
      if (L+R > hb) { R = hb - L; }
    }
    if (R == hb) { break; }
    L <<= 1;  R <<= 1;
  }
}
```

```
/** Finishes a decoding process. */
void finish_decode() {
  // no action required
}
```

Code listing 3.3: JAVA source code of the start and stop procedures in an arithmetic coder.

Arithmetic Coding: default scaling

```
void narrow_region(long l, long h, long t) {
  long r = R / t;
  L = L + r*l;
  R = h < t ? r * (h-l) : R - r*l;
} // Moffat-Neal-Witten (1998)
```

```
void storeRegion(long l, long h, long t) {
  narrow_region(l,h,t);
  output_bits();
}
```

```
void loadRegion(long l, long h, long t) {
  narrow_region(l,h,t);
  discard_bits();
}
```

```
long getTarget(long t) {
  long r = R / t;
  long dr = (D-L) / r;
  return (t-1 < dr) ? t-1 : dr;
} // Moffat-Neal-Witten (1998)
```

```
void narrow_region2(long l, long h, long t) {
  long T = (R*l) / t;
  L = L + T;
  R = (R*h) / t - T;
} // Witten-Neal-Cleary (1987)

long getTarget2(long t) {
  return (((D-L+1)*t)-1) / R;
} // Witten-Neal-Cleary (1987)
```

Code listing 3.4: JAVA source code for the "default scaling" variants of the main interface methods `storeRegion`, `loadRegion` and `getTarget`, using the scaling code by Moffat, Neal and Witten (1998). The scaling method by Witten, Neal and Cleary (1987) is included for comparison.

Template methods for arithmetic encoding and decoding

COMPRESSION

```
void encode(X x, Encoder ec) {
  long r = ec.getRange();
  // find (l,h) for x, given r
  ec.storeRegion(l,h);
}
```

DECOMPRESSION

```
X decode(Decoder dc) {
  long r = dc.getRange();
  long t = dc.getTarget();
  // find (l,h,x) from (r,t)
  dc.loadRegion(l,h);
  return x;
}
```

Code listing 3.5: A sketch of low-level `encode` and `decode` methods implementing compression and decompression using the arithmetic coding interface. The implementing class would typically be a probability distribution over objects `x` of a class `X`, and include means of computing discrete regions `(l,h)` for any object `x` (given the currently available coding range `r`).

3.3.1 Historical notes

Arithmetic codes were first made practical through the development of coding operations in finite precision arithmetic by Rissanen (1976) and Pasco (1976). These ideas were advanced further by Rissanen and Langdon (1979, 1981); Langdon (1984) and others, but it is fair to add that many people worked on arithmetic codes, with the original idea of the method even tracing back to Shannon's paper on information theory (1948). Details on the history of arithmetic coding can be found in the PhD thesis of Sayir (1999).

The arithmetic coding algorithm of this chapter is based on the designs by Witten, Neal and Cleary (1987) and Moffat, Neal and Witten (1998). A tutorial on arithmetic coding can be found e.g. in a technical report by Howard and Vitter (1992).

Aside. Arithmetic coding can be understood as a generalised change of number base. As an illustration, consider feeding e.g. a few hundred digits of the binary expansion of $\frac{\pi}{10}$ =.0101000 00110110010111101110110100111001 1... into the arithmetic decoder described earlier. The decoder, decompressing this input sequence to uniformly distributed decimal digits {0...9}, produces the familiar looking output sequence 3141592653589793 2414227... The shaded region indicates where the decoded digits begin to differ from the true decimal expansion of π: this deviation is a consequence of using finite-precision arithmetic coding.

3.3.2 Other generic compression algorithms

There are other generic algorithms that produce near-optimal compression codes for arbitrary distributions. The *Q-coder* by Pennebaker et al. (1988), for example, is a carefully optimised implementation of arithmetic coding for binary alphabets. It is always possible, given a

discrete probability distribution and an outcome x, to encode x as a sequence of biased binary choices; the Q-coder can therefore be used as universally as an arithmetic coder can.

Similarly, the *Z-coder* by Bottou et al. (1998) produces an optimal compression code for any sequence of binary input symbols and associated biases. The operation of the Z-coder is different from the Q-coder and the arithmetic coding algorithm, and is based on a generalisation of Golomb codes (Golomb, 1966).

3.4 Arithmetic coding for various distributions

In this section, I'll discuss how arithmetic coding can be used in practice to encode data from various basic probability distributions. These basic distributions provide important building blocks from which more complex models can be built. Examples of such models can be found in later chapters of this thesis. The techniques presented here are therefore of important practical use, and also serve as concrete examples of how arithmetic coding can be applied in practice.

3.4.1 Bernoulli code

The perhaps simplest use of an arithmetic coder is the compression of Bernoulli variables with known bias, i.e. a sequence of binary outcomes from coin flips with a known bias φ. Of course, such a sequence could be encoded naïvely by simply writing the outcomes with binary symbols $\in \{0, 1\}$ unmodified, and in the case that the coin is fair (i.e. $\varphi = \frac{1}{2}$), this encoding procedure is in fact optimal. But for biases different from $\frac{1}{2}$, or output alphabets other than binary, an arithmetic coder will be more effective.

The Bernoulli distribution over values $\{0, 1\}$ is defined as:

$$\text{Bernoulli}(x \mid \varphi) \;=\; \varphi^{1-x}\,(1-\varphi)^{x} \tag{3.8}$$

where the bias φ is the probability of obtaining a 0. To encode a Bernoulli-distributed value x with an arithmetic coder, each possible value (0 or 1) must be mapped to a region whose size is roughly proportional to its probability. Using the interface from Table 3.1, this mapping can be implemented by checking the coder's current range R with `getRange()`, and finding the integer M that is closest to $R \cdot \varphi$, and not equal to 0 or R itself. The tuples $(0, M)$ and (M, R) then identify the regions. Concrete procedures for encoding and decoding are shown in code listing 3.6.

Arithmetic coding for Bernoulli distributions

ENCODING

`void encode(int x, Encoder ec) {`

1. Get $R \leftarrow$ `ec.getRange()`.
2. Compute $M \leftarrow$ `round`$(R \cdot \varphi)$.
3. Ensure $0 < M < R$.
   (If necessary, adjust $M \leftarrow M \pm 1$.)
4. If $x = 0$:
   `ec.storeRegion`$(0, M)$.
   Otherwise:
   `ec.storeRegion`(M, R).

`}`

DECODING

`int decode(Decoder dc) {`

1. Get $R \leftarrow$ `dc.getRange()`.
2. Compute $M \leftarrow$ `round`$(R \cdot \varphi)$.
3. Ensure $0 < M < R$.
   (If necessary, adjust $M \leftarrow M \pm 1$.)
4. Set $T \leftarrow$ `dc.getTarget()`.
5. If $T \leq M$:
   `dc.loadRegion`$(0, M)$. Return 0.
   Otherwise:
   `dc.loadRegion`(M, R). Return 1.

`}`

Code listing 3.6: Fixed-precision arithmetic encoding and decoding procedures for a Bernoulli distribution with bias φ, where φ is the probability of outcome 0. The arguments `ec` and `dc` are pointers to instances of an arithmetic encoder and decoder.

3.4.2 Discrete uniform code

Generalising a fair choice between two options, a discrete uniform distribution defines a choice among N equiprobable options (for any finite N), such that each value $n \in \{1, ..., N\}$ has probability $\frac{1}{N}$. Encoding and decoding procedures for such a distribution could use the scaling versions of `storeRegion`, `loadRegion`, and `getTarget` from code listing 3.4. Encoding an integer n is as simple as invoking `ec.storeRegion`$(n-1,\ n,\ N)$. Decoding works by obtaining $k \leftarrow$ `dc.getTarget`(N), calling `dc.loadRegion`$(k,\ k+1,\ N)$, and returning $n = k+1$.

3.4.3 Finite discrete distributions

The approach of the previous two sections can be generalised to distributions P over N possible values, where N is finite. Each of the N possible values $\{1, ..., N\}$ must be mapped to a discrete region in the coder's current range R.

Any method that achieves the above necessarily computes P's *cumulative distribution* in some form. Intuitively, this is because each element is mapped to a line-segment of a length proportional to the element's probability mass, and the location of the segment is given by the sum of all preceding segment sizes. The order of the segments doesn't matter, and can be chosen for computational convenience, as long as the encoder and decoder agree.

P's cumulative distribution can be defined as follows:

$$P_\Sigma(n) \stackrel{\text{def}}{=} \sum_{k=1}^{n-1} P(k) \tag{3.9}$$

$$\text{and} \quad P^+_\Sigma(n) \stackrel{\text{def}}{=} P_\Sigma(n) + P(n). \tag{3.10}$$

The cumulative distribution can be used to map each $n \in \{1, ..., N\}$ to a region $[P_\Sigma(n), P^+_\Sigma(n))$ of length $P(n)$ inside the unit interval. To use these regions with our fixed-precision arithmetic coder, the boundaries must be scaled to integers between 0 and R, taking care that there are no gaps and that no region has a width of zero.

A brute-force way of computing discrete regions with the above constraints is shown in code listing 3.7. To improve efficiency, it may help to choose an ordering in which high probability elements (i.e. large regions) come first, so that the expected number of summations and evaluations of P is kept low.

The computational costs can be lowered in certain circumstances. For example, when the distribution P is used frequently and changes rarely, one might cash a precomputed copy of the cumulative distribution P_Σ and scale it on demand. Even cheaper alternatives may exist when P_Σ can computed in closed form, avoiding the need to iterate over the elements.

3.4.4 Binomial code

The binomial distribution describes the number of successes versus failures in a set of N independent Bernoulli trials. It is parametrised by natural number N and success probability θ, and ranges over positive integers $n \in \{0 \dots N\}$. A binomial random variable has the following probability mass function:

$$\text{Binomial}(n \mid N, \theta) = \binom{N}{n} \cdot \theta^n \ (1-\theta)^{N-n} \tag{3.11}$$

Encoding a binomial random variable with an arithmetic coder requires computing the cumulative distribution function of the binomial distribution. A method for doing this efficiently is to make use of the following recurrence relation:

$$\text{Binomial}(n+1 \mid N, \theta) = \frac{N-n}{n+1} \cdot \frac{\theta}{1-\theta} \cdot \text{Binomial}(n \mid N, \theta) \tag{3.12}$$

The cumulative binomial distribution can then be computed as follows. Initialise:

$$B_\Sigma := 0 \tag{3.13}$$

$$B := (1-\theta)^N \tag{3.14}$$

Budget allocation algorithm

```
public static <X> HashMap<X,Long> getDiscreteMass(Mass<X> mass,
                                                  Iterable<X> set, long budget) {
  HashMap<X,Long> map = new HashMap<X,Long>();
  int n = 0; // number of elements
  long sum = 0L; // total allocated mass
  for (X x : set) {
    final double m = mass.mass(x);
    long mb = (long) (m * budget);
    if (mb <= 0L) { mb = 1L; }
    map.put(x, mb);
    sum += mb; n++;
  }
  // deal with left-over or overspent budget:
  long diff = budget - sum;
  if (diff > 0L) {
    // underspent budget
    int delta =
          (diff > n) ?  (int) (diff/n) : 1;
    for (Map.Entry<X,Long> e : map.entrySet()) {
      e.setValue(e.getValue()+delta);
      diff-=delta;
      if (diff==0L) { break; }
      else
      if (diff <= n) { delta = 1; }
    }
  } else
  if (diff < 0L) {
    // overspent budget
    int delta =
          (diff < -n) ? (int) - (diff/n) : 1;
    for (Map.Entry<X,Long> e : map.entrySet()) {
      Long v = e.getValue();
      if (v > 1L) {
        e.setValue(v-delta);
        diff+=delta;
      }
      if (diff==0L) { break; }
      else
      if (diff >= -n) { delta = 1; }
    }
  }
  return map;
}
```

```
public static <X> void encode(X x,
                    Mass<X> mass,
                    Iterable<X> order,
                    Encoder ec) {
  long sum = 0L;
  Map<X,Long> map = getDiscreteMass(
     mass, set, order, ec.getRange());
  for (X y : order) {
    long m = map.get(y);
    if (x.equals(y)) {
      ec.storeRegion(sum,sum+m);
      return;
    }
    sum += m;
  }
  // unsupported element
  throw new RuntimeException();
}
```

```
public static <X> X decode(
                      Mass<X> mass,
                      Iterable<X> order,
                      Decoder dc) {
  long sum = 0L;
  Map<X,Long> map = getDiscreteMass(
     mass, set, order, dc.getRange());
  long r = dc.getTarget();
  for (X x : order) {
    long m = map.get(x);
    if (r >= sum && r < sum+m) {
      dc.loadRegion(sum,sum+m);
      return x;
    }
    sum += m;
  }
  // unused coding range
  throw new RuntimeException();
}
```

Code listing 3.7: A JAVA method that discretizes a probability mass function `mass` (over any iterable space `X`) to a given integer budget. The method returns a map from elements in `X` to integers that sum to the given budget. Summing the resulting integers returns region boundaries that can be used with the arithmetic coding interface: suitable implementations of `encode` and `decode` are shown in the shaded boxes.
Note: These three methods are presented in this form mainly for clarity of illustration, they can (and should) be optimised.

To encode a binomially distributed value n, repeat for each k from $1 \ldots n$:

$$B_\Sigma := B_\Sigma + B \tag{3.15}$$

$$B := \frac{N-k}{k+1} \cdot \frac{\theta}{1-\theta} \cdot B \tag{3.16}$$

The interval $[B_\Sigma, B_\Sigma + B)$ is then a representation of n that can be scaled for use with a finite-precision arithmetic coder.

3.4.5 Beta-binomial code

The Beta-binomial compound distribution results from integrating out the success parameter θ of a binomial distribution, assuming θ is Beta distributed. The Beta distribution is a continuous distribution over the unit interval $[0, 1]$, and is defined as follows:

$$\mathrm{Beta}(\theta \mid \alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\,\Gamma(\beta)} \cdot \theta^{\alpha-1}\,(1-\theta)^{\beta-1}\,, \tag{3.17}$$

where $\Gamma(\cdot)$ denotes the Gamma function.[4] The Beta-binomial distribution expresses the probability of getting exactly n heads from N coin flips, when the coin has an unknown, Beta-distributed bias θ.

The Beta-binomial distribution is parametrised by the number of trials N and the parameters α and β of the Beta prior:

$$\mathrm{BetaBin}(n \mid N, \alpha, \beta) = \int \mathrm{Binomial}(n \mid N, \theta) \cdot \mathrm{Beta}(\theta \mid \alpha, \beta)\, d\theta \tag{3.18}$$

$$= \binom{N}{n} \cdot \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\,\Gamma(\beta)} \cdot \frac{\Gamma(\alpha+n)\,\Gamma(\beta+N-n)}{\Gamma(\alpha+\beta+N)} \tag{3.19}$$

Just like for the binomial distribution, there is a recurrence relation that can be used to compute the probabilities for successive values of n:

$$\mathrm{BetaBin}(n+1 \mid N, \alpha, \beta) = \frac{N-n}{n+1} \cdot \frac{\alpha+n}{\beta+N-n-1} \cdot \mathrm{BetaBin}(n \mid N, \alpha, \beta) \tag{3.20}$$

Summing these probabilities in ascending order of n yields the cumulative distribution. The method described in section 3.4.4 can be modified accordingly, yielding a Beta-binomial coding scheme.

Binomial and Beta-binomial codes are used e.g. in chapter 7 as part of a compressor for multisets of binary sequences.

[4] The Gamma function can be regarded a continuous generalisation of the factorial function, with $k! = \Gamma(k+1) = k \cdot \Gamma(k)$. The Beta distribution gets its name from the Beta function $\mathrm{B}(\alpha, \beta) = \Gamma(\alpha)\Gamma(\beta)/\Gamma(\alpha+\beta)$, as $\mathrm{B}(\alpha, \beta)^{-1}$ is the normalising constant of the Beta distribution.

3.4.6 Multinomial code

The K-dimensional *multinomial distribution* is a generalisation of the binomial distribution to trials with K possible outcomes. It is parametrized by the number of trials N, and by a discrete probability distribution Q over the K possible values. A draw from a multinomial distribution is a K-dimensional vector $\boldsymbol{n} = (n_1 \ldots n_K)$ over positive integers, whose components sum to N. The probability mass function of the multinomial distribution is:

$$\mathrm{Mult}(n_1 \ldots n_K \mid N,\ Q) = N! \prod_{k=1}^{K} \frac{Q(k)^{n_k}}{n_k!} \tag{3.21}$$

Interfacing a multinomial distribution to an arithmetic coder is not as straightforward as with distributions over integers, because the coding domain is vector-valued and may have a large state space. Each possible vector must map to a coding region, which requires fixing some total ordering over vectors. The approach recommended here is to encode $\boldsymbol{n}$ by decomposing it into a series of simpler coding steps, preserving optimality.

The technique described here exploits a decomposition property of the Multinomial distribution:

$$N! \prod_{k=1}^{K} \frac{Q(k)^{n_k}}{n_k!} = \left((N-n_1)! \cdot \prod_{k=2}^{K} \frac{1}{n_k!} \left(\frac{Q(k)}{1-Q(1)} \right)^{n_k} \right) \cdot \frac{N!}{n_1!\ (N-n_1)!}\, Q(1)^{n_1} (1-Q(1))^{N-n_1} \tag{3.22}$$

As a consequence of this property, the Multinomial distribution can be factorised as follows:

$$\begin{aligned} &\mathrm{Mult}(n_1 \ldots n_K \mid N,\ Q) \\ &\quad = \mathrm{Binomial}(n_1 \mid N, Q(1)) \\ &\qquad \cdot \mathrm{Mult}\left(n_2 \ldots n_K \,\middle|\, N-n_1,\ \frac{Q(2)}{1-Q(1)} \ldots \frac{Q(K)}{1-Q(1)} \right) \end{aligned} \tag{3.23}$$

This equation can be applied recursively to decompose the multinomial distribution over $\boldsymbol{n}$ into a component-wise product of binomial distributions:

$$\mathrm{Mult}(n_1 \ldots n_K \mid N,\ Q) = \prod_{k=1}^{K} \mathrm{Binomial}\left(n_k \,\middle|\, N_k,\ \frac{Q(k)}{1-\sum_{j<k} Q(j)} \right) \tag{3.24}$$

where $N_k = \left(N - \sum_{j=1}^{k-1} n_j\right) = \left(\sum_{j=k}^{K} n_j\right)$ are the remaining trials. This factorisation may also be expressed using domain restriction (defined in section 3.5.2):

$$\mathrm{Mult}(n_1 \ldots n_K \mid N,\ Q) = \prod_{k=1}^{K} \mathrm{Binomial}\left(n_k \,\middle|\, N_k,\ Q_{\setminus\{<k\}}(k) \right) \tag{3.25}$$

Based on this insight, a basic multinomial coding method can process $\boldsymbol{n}$ sequentially and encode each n_k using a binomial code, such as the one described in section 3.4.4.

Infinite limit

It is worth pointing out that the multinomial coding method described here works even when $K = \infty$. This "infinomial distribution" is defined exactly like the multinomial distribution in (3.21), except that it has a countably infinite number of components. It is still parametrised by a finite number of trials $N \in \mathbb{N}$, but the number of possible outcomes is now infinite, i.e. the distribution Q has support over a countably infinite set.

To encode an infinomial vector $\boldsymbol{n}$, the same algorithm can be used. Equation (3.24) applies just like in the finite case, and once N_k reaches zero the computation may safely stop since all remaining factors are then equal to 1.

3.4.7 Dirichlet-multinomial code

Just like the multinomial distribution generalises the binomial distribution to trials with K possible outcomes, the *Dirichlet-multinomial* compound distribution generalises the Beta-binomial distribution. The distribution arises from integrating out Q from a multinomial distribution, assuming that Q itself is Dirichlet distributed.

The K-dimensional Dirichlet distribution defines a probability for discrete distributions over K possible values, and is parametrized by a K-dimensional concentration vector $\boldsymbol{\alpha} = (\alpha_1 \ \ldots \ \alpha_K)$. It is therefore a distribution over distributions, defined as follows:

$$\mathrm{Dir}(Q \mid \boldsymbol{\alpha}) \ = \ \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \cdot \prod_k Q(k)^{\alpha_k - 1} \tag{3.26}$$

The Dirichlet distribution is a multivariate generalisation of the Beta distribution, as can be seen from equations (3.17) and (3.26).

The probability mass function of the Dirichlet-multinomial can be obtained as follows:

$$\begin{aligned} \mathrm{DirMult}(\boldsymbol{n} \mid N, \ \boldsymbol{\alpha}) \ &= \ \int \mathrm{Mult}(\boldsymbol{n} \mid N, Q) \cdot \mathrm{Dir}(Q \mid \boldsymbol{\alpha}) \, dQ && (3.27) \\ &= \frac{N!}{\prod_k n_k!} \cdot \frac{\Gamma(A)}{\prod_k \Gamma(\alpha_k)} \cdot \int \prod_k Q(k)^{n_k + \alpha_k - 1} \, dQ && (3.28) \\ &= \frac{N! \ \Gamma(A)}{\Gamma(N + A)} \cdot \prod_k \frac{\Gamma(n_k + \alpha_k)}{n_k! \ \Gamma(\alpha_k)}, && (3.29) \end{aligned}$$

where $A = (\sum_k \alpha_k)$, and the components of $\boldsymbol{n}$ are non-negative integers that sum to N.

Just like with the multinomial distribution, a decomposition property can be used to split

the Dirichlet-multinomial distribution into a product:

$$\begin{aligned}&\mathrm{DirMult}(n_1 \,...\, n_K \mid (\alpha_1 \,...\, \alpha_K)\,, N)\\ &\quad= \mathrm{BetaBin}(n_1 \mid N,\ \alpha_1,\ A{-}\alpha_1)\\ &\quad\cdot\ \mathrm{DirMult}(n_2 \,...\, n_K \mid \alpha_2 \,...\, \alpha_K,\ N{-}n_1)\end{aligned} \tag{3.30}$$

This equation can be applied recursively to decompose the Dirichlet-multinomial distribution over $\boldsymbol{n}$ into a component-wise product of Beta-binomial distributions:

$$\mathrm{DirMult}(n_1 \,...\, n_K \mid \alpha_1 \,...\, \alpha_K, N) = \prod_{k=1}^{K} \mathrm{BetaBin}\left(n_k \,\middle|\, \left(\alpha_k, \sum_{j=k+1}^{K} \alpha_j\right), N - \sum_{j=1}^{k} n_j\right) \tag{3.31}$$

One can therefore compress $\boldsymbol{n}$ by sequentially encoding each component n_k using the Beta-binomial code from section 3.4.5.

Multinomial and Dirichlet-multinomial codes are used in chapter 5 for compressing unordered structures, such as multisets.

3.4.8 Codes for infinite discrete distributions

This section outlines approaches for using arithmetic coding with distributions over countably infinite sets. Such distributions include e.g. the Poisson, geometric, and negative binomial distributions. The primary difficulty of interfacing an infinite discrete distribution to a finite-precision coder is the handling of the infinite set of elements.

Without loss of generality, suppose we want to encode natural numbers $n \in \mathbb{N}$ with some known distribution P. Using the natural ordering, each element n could conceptually be mapped to $[P_\Sigma(n), P^+_\Sigma(n))$ in the unit interval. But these regions cannot be discretized into any finite range of integers, as would be required for use with a finite-precision arithmetic coder.

One solution is to factorise P in such a way that each factor involves a choice between a finite number of regions: in such a case the number of factors is generally unbounded and depends on n.

There are several ways in which such a factorisation can be made. For example, each n can be represented as a sequence of binary decisions of the form 'Is it k?' for all k from 0 to n:

$$P(n) = \prod_{k=0}^{n} \mathrm{Bernoulli}(\mathbb{1}[k{=}n] \mid \theta_k), \quad \text{where} \quad \theta_k = \frac{P(k)}{P(\{0, ..., k{-}1\})}. \tag{3.32}$$

This scheme could be viewed as a generalisation of the unary code for integers from section 2.2.1: instead of using Bernoulli choices of bias $\frac{1}{2}$, it uses Bernoulli choices of biases θ_k as shown in equation (3.32).

This method encodes each natural number n with an expected number of $\log_2 P(n)^{-1}$ bits, the Shannon information content of n under P. However, the process may be computationally inefficient for many choices of P, as the number of coding steps grows linearly with n. If the distribution P has heavy tails, or distant high mass elements, this could introduce long waits. This inefficient behaviour can be reduced by arranging the elements in descending order of probability mass, decreasing the average number of coding steps per element; but ultimately, the fact that the sequence of coding steps has a unary structure can be a computationally limiting factor for many choices of P.

Of course, many other factorisations are possible whose suitability depends on P. It should be noted that a fixed-precision arithmetic coder cannot faithfully encode choices below a probability mass of 2^{-R}, where R is the coder's current coding range. The factors must therefore be chosen such that its probabilities lie sufficiently above the limiting threshold.

Another approach to this problem might be to ignore elements whose probability is below a certain threshold; and treat the occurrence of such low-probability events as an error condition.

3.5 Combining distributions

Probabilistic models can be constructed by combining simpler distributions. This section describes coding techniques for some common ways in which probability distributions can be combined.

3.5.1 Sequential coding

Recall from section 3.2 that the sequential arithmetic encoding of symbols $x_1 \ldots x_N$ with associated probability distributions $P_1 \ldots P_N$ produces a final region whose implied probability mass corresponds to the product of the individual symbol probabilities:

$$P(x_1 \ldots x_N) \;=\; \prod_{n=1}^{N} P_n(x_n). \tag{3.33}$$

If we want to arithmetically encode a multivariate vector $(x_1 \ldots x_N)$ for which only the joint distribution is known, we may have to factorise the distribution into marginal and conditional distributions first. For example, a distribution over two variables can be factorised in two ways:

$$\Pr(x, y) \;=\; \Pr(x) \cdot \Pr(y \mid x) \;=\; \Pr(y) \cdot \Pr(x \mid y) \tag{3.34}$$

To encode values x and y according to their joint distribution, one could first encode x with the marginal distribution $\Pr(x)$ and then encode y with the conditional distribution $\Pr(y \mid x)$.

This construction preserves optimality and always works, but requires a coding method for the marginal and conditional distributions.

It can help to choose the marginal: sometimes $\Pr(x \mid y)$ is easier to encode than $\Pr(y \mid x)$. If conditional distributions are difficult to compute, but the marginals are not, it might be tempting to use *only* the marginals: i.e. coding x with $\Pr(x)$, then y with $\Pr(y)$. While using such an encoding still allows (x, y) to be reconstructed, the wrongly assumed independence of x and y may incur a bandwidth cost.[5]

More generally, factors in a probabilistic model turn into sequential coding steps. The order of the steps must be such that each step depends only on information that has already been encoded, so that the decoder can reconstruct each conditional distribution in the same order. Which marginals or conditionals are chosen does not matter, and can be chosen for computational convenience.

3.5.2 Exclusion coding

One particular form of conditional distribution is a *value exclusion* or *domain restriction*, where a subset $\mathcal{R}$ of values is excluded from the choice:

$$P(x \mid x \notin \mathcal{R}) \;=\; \frac{P(x \notin \mathcal{R} \mid x) \cdot P(x)}{P(x \notin \mathcal{R})} \;=\; \frac{\mathbb{1}[x \notin \mathcal{R}] \cdot P(x)}{1 - P(\mathcal{R})} \tag{3.35}$$

This conditional distribution is well defined as long as $P(\mathcal{R}) < 1$. We'll abbreviate this conditional distribution $P_{\setminus \mathcal{R}}$:

$$P_{\setminus \mathcal{R}}(x) \;=\; \begin{cases} 0 & x \in \mathcal{R} \\ \dfrac{P(x)}{1 - P(\mathcal{R})} & \text{otherwise} \end{cases} \tag{3.36}$$

Sampling from $P_{\setminus \mathcal{R}}$ is the same as sampling from P and rejecting values $x \in \mathcal{R}$. (Of course there may be more efficient ways to sample from $P_{\setminus \mathcal{R}}$ than rejection sampling.)

Encoding or decoding values x under a conditional distribution of the above kind is called *exclusion coding*. For data compression algorithms, exclusion coding can be a useful tool for several reasons:

1. It provides a simple way to incorporate knowledge into an existing model that certain values cannot occur, saving otherwise wasted information.

2. It preserves the relative proportions of probability mass of non-excluded elements, and

[5] For example, consider the distribution $\Pr(x, y) = \mathbb{1}[x = y]$ where x and y take values in $\{0, 1\}$. The marginal distributions are $\Pr(x) = \Pr(y) = \frac{1}{2}$. Encoding a tuple (x, y) with $\Pr(x) \cdot \Pr(y \mid x)$ costs 1 bit, whereas coding x and y independently with $\Pr(x)$ and $\Pr(y)$ costs 2 bits.

can be used as a mathematical tool to encode components of multivariate distributions. Such use occurred e.g. in section 3.4.6, equation (3.25).

3. Domain restrictions can be used to replace a mixture model with a computationally favourable alternative, by enforcing the domains of the mixture components to be disjoint. Separating the mixture components in this way lowers the cost of encoding and decoding.

Value exclusions are used implicitly in some data compression algorithms, for example in the PPM algorithm by Cleary and Witten (1984a) and many of its variants. PPM algorithms and their corresponding probabilistic models are covered in detail in chapter 6.

3.5.3 Mixture models

A mixture model combines a finite or countable number of distributions according to a mixing distribution. For example, consider a family of distributions D_1 ... D_K over some space $\mathcal{X}$, and a discrete distribution Θ over the index set $\{1 \dots K\}$, to define a *mixture distribution* M:

$$M(x) \;=\; \Pr(x \mid \Theta, D_1 \;...\; D_K) \;=\; \sum_{k=1}^{K} \Theta(k) \cdot D_k(x) \tag{3.37}$$

The distribution Θ is called the *mixing distribution*. Sampling from M is easy: first draw an index k from Θ, then draw a value x from the kth component distribution D_k (and throw away k). To obtain the probability mass $M(x)$ of an outcome x, the contributions of each component distribution D_k must be summed (unless the sum has a closed form).

To arithmetically encode a value x with a mixture distribution M, discrete region boundaries must be computed, for example by using the mixture's cumulative probability $M_\Sigma(x)$.

If the mixing distribution Θ changes more frequently than the component distributions D_k, it may help to compute $M_\Sigma(x)$ as follows:

$$M_\Sigma(x) \;=\; \sum_{k=1}^{K} D_{k\Sigma}(x) \tag{3.38}$$

where the cumulative component distributions $D_{k\Sigma}$ can be cached to speed up the computation.