Models With and Without Context

Data compression depends on us having a *model* of the source — which provides probabilities for each symbol.

So far, we've assumed that symbols are *independent*, and that their probabilities don't depend on their position in the file.

Very often, this isn't true: The probability of a symbol may depend on the *context* in which it occurs — eg, what symbol precedes it. Example: "U" is much more likely after "Q" (in English), than after another "U".

Probabilities may also depend on position in the file, though this is less common. Example: Executable program files may have machine instructions at the beginning, and symbols for debugging at the end.

Static and Adaptive Models

Models also differ in whether the probabilities (in a given context) are fixed — a *static* model — or whether the probabilities *adapt* based on what we learn by looking at previous symbols.

A static model is appropriate if we know a lot about the source even before we see any symbols. For example: We may know a lot about frequencies of letters in English.

An adaptive model is appropriate if we don't know a lot — maybe not even the language the docment is in — or if the source itself varies. Example: English documents vary depending on whether they are reports of football games, legal judgements, recipies, etc.

Coding Based on Symbol Frequencies

One common adaptive scheme: Encode symbols based on probablities obtained from the frequencies (counts) of how often the symbol occurred *earlier* in the document.

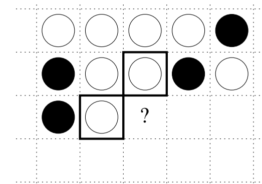
The decoder can know the same frequencies, since it will have already decoded the earlier symbols.

We need to avoid any *zero* frequencies — since they will make encoding that symbol impossible. One scheme: Set the frequencies to one at the beginning, then increment them as symbols are seen.

Example: Adaptively Compressing Black-and-White Images

Suppose we have images (of some fixed dimensions) in which pixels are either black or white. We may expect large regions of white pixels and large regions of black pixels, and wish to use that knowledge to compress them.

But how large will the regions be? We probably don't know, so we use an adaptive scheme, with contexts defined by pixels above and to the left:



The Encode Program

```
/* Initialize model. */

for (a = 0; a<2; a++) {
    for (l = 0; l<2; l++) {
        freq0[a][l] = 1;              /* Set frequencies of 0's */
        freq1[a][l] = 1;              /* and 1's to be equal.   */
    }
}

/* Encode image. */

for (i = 0; i<Height; i++) {
    for (j = 0; j<Width; j++) {
        a = i==0 ? 0 : image[i-1][j];  /* Find current context.  */
        l = j==0 ? 0 : image[i][j-1];
        encode_bit(image[i][j],        /* Encode pixel.          */
                   freq0[a][l],freq1[a][l]);
        if (image[i][j]) {             /* Update frequencies for */
            freq1[a][l] += 1;          /* this context.          */
        }
        else {
            freq0[a][l] += 1;
        }
        if (freq0[a][l]+freq1[a][l]>Freq_full) { /* Avoid huge   */
            freq0[a][l] = (freq0[a][l]+1) >> 1;  /* frequencies  */
            freq1[a][l] = (freq1[a][l]+1) >> 1;
        }
    }
}
```

The Decode Program

```
/* Initialize model. */

for (a = 0; a<2; a++) {
    for (l = 0; l<2; l++) {
        freq0[a][l] = 1;              /* Set frequencies of 0's */
        freq1[a][l] = 1;              /* and 1's to be equal.   */
    }
}

/* Decode and write image. */

for (i = 0; i<Height; i++) {
    for (j = 0; j<Width; j++) {
        a = i==0 ? 0 : image[i-1][j];  /* Find current context.  */
        l = j==0 ? 0 : image[i][j-1];
        image[i][j] =                  /* Decode pixel.          */
          decode_bit(freq0[a][l],freq1[a][l]);
        printf("%c%c",image[i][j] ? '#' : '.',
                      j==Width-1 ? '\n' : ' ');
        if (image[i][j]) {             /* Update frequencies for */
            freq1[a][l] += 1;          /* this context.          */
        }
        else {
            freq0[a][l] += 1;
        }
        if (freq0[a][l]+freq1[a][l]>Freq_full) { /* Avoid huge   */
            freq0[a][l] = (freq0[a][l]+1) >> 1;  /* frequencies  */
            freq1[a][l] = (freq1[a][l]+1) >> 1;
        }
    }
}
```

Example: Adaptive Text Compression

We can also adaptively compress text, using counts of how often letters (and other symbols) have occurred earlier.

With many symbols, we gain efficiency by reordering them by decreasing frequency, so that we usually don't have to search very far.

The main encode program:

```
frequencies f;  /* Structure holding character frequencies  */
int ch;         /* Character to encode                      */
int index;      /* Index of character to encode             */

initialize_frequencies(&f);           /* Set all frequencies to 1 */

for (;;) {                            /* Loop through characters. */

    ch = getc(stdin);                 /* Read the next character. */
    if (ch==EOF) break;               /* Exit loop on end-of-file.*/
    index = f.symbol_to_index[ch];    /* Translate to an index.   */
    encode_symbol(index,f.cum_freq);  /* Encode that symbol.      */
    update_frequencies(&f,index);     /* Update symbol frequencies*/
}

index = f.symbol_to_index[EOF_symbol];/* Encode the EOF symbol.   */
encode_symbol(index,f.cum_freq);
```

Procedure to Increment a Frequency

```
for (i = index; f->freq[i]==f->freq[i-1]; i--) ; /* Find new index.   */

if (i<index) {
    sym_i = f->index_to_symbol[i];         /* Update the translation  */
    sym_index = f->index_to_symbol[index]; /* tables if the symbol has*/
    f->index_to_symbol[i] = sym_index;     /* moved.                  */
    f->index_to_symbol[index] = sym_i;
    f->symbol_to_index[sym_i] = index;
    f->symbol_to_index[sym_index] = i;
}

f->freq[i] += 1;                           /* Increment the frequency */
while (i>0) {                              /* count for the symbol and*/
    i -= 1;                                /* update the cumulative   */
    f->cum_freq[i] += 1;                   /* frequencies.            */
}

if (f->cum_freq[0]>Freq_full) {            /* See if frequency counts */
    f->cum_freq[No_of_symbols] = 0;        /* are past their maximum. */
    for (i = No_of_symbols; i>0; i--) {    /* If so, halve all counts */
        f->freq[i] = (f->freq[i]+1) >> 1;  /* (keeping them non-zero).*/
        f->cum_freq[i-1] = f->cum_freq[i] + f->freq[i];
    }
}
```