Context modelling

Consider the familiar problem that the encoder/decoder wishes to encode/decode the next symbol of the sequence, say X_{j+1}, given that it has encoded X_1 to X_j. Let's now suppose the encode wishes to use a k^{th} order Markov model to encode the next element X_{j+1}. If the conditional probabilities are known then the encoder can use a technique called *arithmetic coding* which I will introduce a few lectures from now. Before I get to that, let's address a few basic issues. The first is, how can we represent probabilities? The second is, where do the probabilities come from?

Tree-based representation of probabilities

Consider an $N-$ary tree of depth $k+1$. Each node in the tree has N children, namely one child for each symbol A_i in our alphabet.

At each node at level l, we store the probability that $l+1$ consecutive variables $X_j, X_{j+1}, \ldots, X_{j+l}$ have particular values $i_j, i_{j+1}, \ldots, i_{j+l}$, namely

$$p(X_j = i_j, X_{j+1} = i_{j+1}, \ldots, X_{j+l} = i_{j+l}).$$

We also have an array of N pointers to the children.

Level l of the tree has N^l nodes. Note that all the nodes together at level l define the joint probability function, $p(X_j, X_{j+1}, \ldots, X_{j+l})$. Also note that the probability at any non-leaf node is equal to the sum of probabilities of the N children of that node.

[ASIDE: I have not specified whether the probability function is stationary or not. If it is stationary, then the same tree would hold for every j.]

Given this joint probability tree, we can define a conditional probability tree, namely for each node at level l we have:

$$p(X_{j+l} = i_{j+l} | X_j = i_j, \ldots, X_{j+l-1} = i_{j+l-1}) = \frac{p(X_j = i_j, \ldots, X_{j+l} = i_{j+l})}{p(X_j = i_j, \ldots, X_{j+l-1} = i_{j+l-1})}$$

At each node in the conditional probability tree, we would store the ratio of the probabilities from the joint probability tree, namely the ratio of probability at a node to the probability of the parent. (The root node has probability 1.)

As an example, consider a sequence generated by coin flips ($N = 2$). Each node in the joint probability binary tree at level l would represent a sequence of l flips, and the probability would be $(\frac{1}{2})^l$. For the conditional probability tree, at each (non-root) node we would have probability $\frac{1}{2}$.

Later in this lecture, we will return to these trees. For now, we turn to the second question mentioned above.

Where do the probabilities come from?

If the encoder/decoder wishes to use a k^{th} order Markov model to encode/decode the next symbol of the sequence, it needs to know the probabilities,

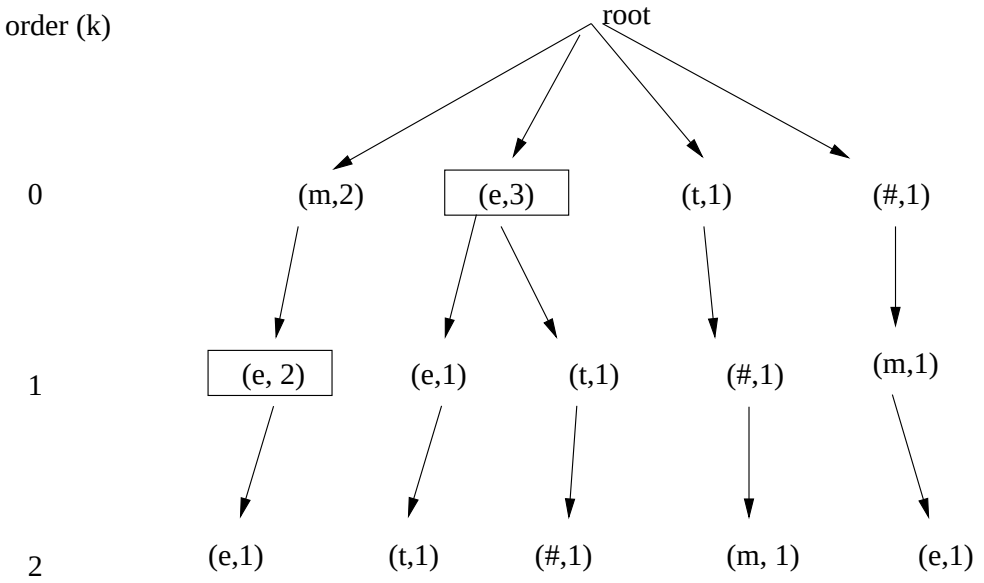
$$p(\text{next symbol} \mid \text{previous } k \text{ symbols }).$$

Where does the encoder/decoder get these probabilities from? There are several possibilities.

- Use some standard set of probabilities which is agreed upon in advance (for example, by a standards committee). These probabilities might be in the form of a table or a math formula.
- The encoder could scan through the sequence it wants to encode, and estimate a (stationary) probability function based on the global frequency of occurences of symbols in the various contexts, that is, how often does A_i occur given what the previous k symbols were (called the *context*). The advantage of pre-scanning is that the encoder can get a good model for the probabilities in the sequence, and so can choose an appropriate code for that sequence. As discussed in lecture 9, however, the disadvantage is that the encoder needs to tell the decoder what these probabilites are. Encoding the code costs bits, i.e. a header file must be sent.
- The encoder adaptively estimates probabilities as it is encoding the sequence, based on the symbols $X_1, \ldots, X_j$ that have been seen and encoded. The advantage of this method is that the decoder has access to these previously-seen symbols (once it decodes them), and so it can estimate the probabilities using the same scheme as the encoder is using. The disadvantage is that the encoder/decoder needs to build up the probability model from nothing. Symbols early in the sequence will not be compressed very well.

Let's discuss this last possibility in more detail. Intuitively, this method would calculate the probabilities of the next symbol based on the number of times in the past that the next symbol has been seen in the current context. For this, the encoder and decoder need to maintain counts, which we call ct.

Consider the example of a string `meet#me` where `#` is a blank. The figure below shows the nodes of a count tree for which the counts are non-zero. The box around two of the nodes shows the current context for a $k = 1$ and $k = 2$ model. Notice that for these two nodes, the counts of the children are one less than the current count. The reason is that the current context has occured (and is counted), but we don't yet know what the next symbol will be (and so one count has not yet been added to a child).



Given such a *ct* tree, we could estimate the probability that the next symbol is A_i given the current context, i.e. given the previous k symbols:

$$p(\text{next symbol is } A_i | \text{ previous } k \text{ symbols}) \equiv \frac{ct(A_i \mid \text{previous } k \text{ symbols})}{\sum_{j=1}^{N} ct(A_j \mid \text{previous } k \text{ symbols})}$$

However, this definition only makes sense when the numerator is non-zero for all i. Otherwise, we would be assuming that if a symbol has never occured before in the current context then the probability that it occurs *next* is zero, which makes no sense (since nothing could ever happen for the first time). Some other formula is therefore need to estimate conditional probabilities from counts.

Zero frequency problem

How do we assign a probability to something that has never happened before? This situation is known as the *zero frequency* problem. Various "solutions" to this problem have been debated. One simple solution is to assume a uniform probability to all events (symbols) that have never happened before in the given context. But note that this solution still does not quite solve the problem. How small should this uniform probability be (relative to events that have occured before in the present context). That is, how much weight should be given to events that *have* happened before, relative to events that have *not* happened before?

One solution is to change the above formula to

$$p(\text{next symbol is } A_i | \text{ previous } k \text{ symbols}) \equiv \frac{\epsilon + ct(A_i \mid \text{previous } k \text{ symbols})}{\sum_{j=1}^{N} \{\epsilon + ct(A_j \mid \text{previous } k \text{ symbols})\}}$$

Notice that this *is* a probability function, in that the sum over A_i's is 1.

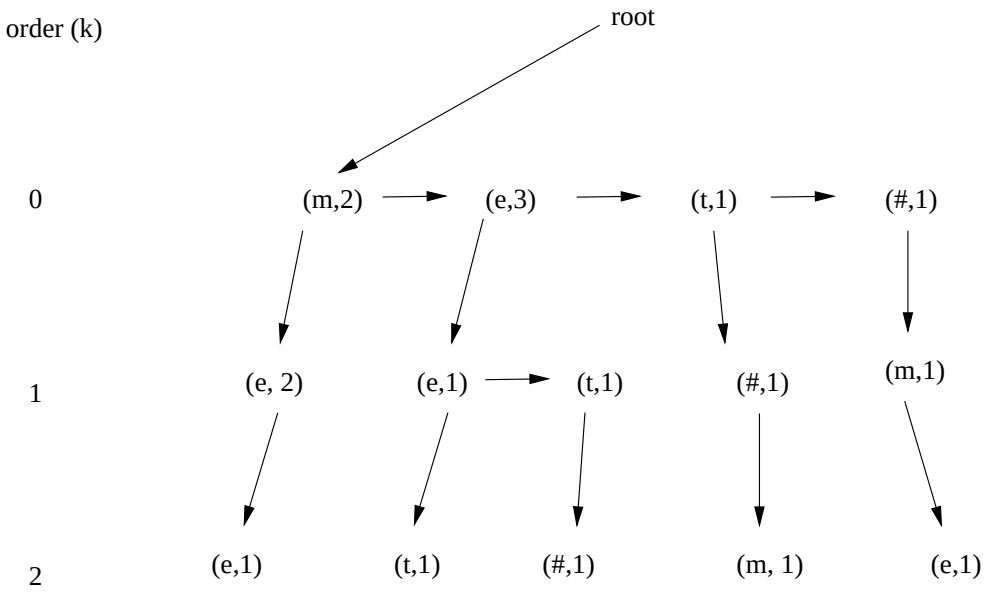
There is single no "correct" value for ϵ. Choosing ϵ to be larger would gives more weight to the prior assumption that all outcomes are equally likely.

Efficient representation

For large alphabets (large $N = 128$) and non-zero order Markov models (say $k = 3$), it is not practical to represent the counts of all combinations of next symbol given previous k symbols since there are N^{k+1} such counts (k for the context, 1 for the next symbol). Most of these counts would be 0!

Instead of having N child pointers at each node, we have a single child pointer for each node. Each node also has a sibling pointer. (If a node has no sibling, then the sibling would be the null pointer.) Thus, each node is part of a linked list. The list defines the set of all symbols that have occured in a given context. The same example from earlier in the lecture is shown below with the new data structure.

I neglected to mention in class that such trees are vaguely reminiscent of Lempel-Ziv (version 3), in that they represent a dictionary of strings which grow down branches. You can search for a string by traversing the tree from root down. But there are important differences. First, we maintain counts at each node. These counts are used to estimate probabilities. Second, we do not grow the tree to arbitrary depths. Rather, we grow to depth $k + 1$.



Finally, such a tree where we store a value at a node is also known as a *tries*. The term *trie* is taken from the word *retrieval*, such tries are often used to store dictionaries and to retrieve words from dictionaries.