An Open Interface for Probabilistic Models of Text

John G. Cleary and W. J. Teahan

{*j.cleary, b.teahan*}*@cs.waikato.ac.nz*

February 7, 2000

Abstract: An Application Program Interface (API) for modelling sequential text is described. The API is intended to shield the user from details of the modelling and probability estimation process. This should enable different implementations of models to be replaced transparently in application programs. The motivation for this API is work on the use of textual models for applications in addition to strict data compression, e.g. determination of the source of text, spelling correction or segmentation of text by inserting spaces. The API is probabilistic: that is, it supplies the probability of the next symbol in the sequence. It is general enough to deal accurately with models that include escapes for probabilities. The concepts abstracted by the API are explained together with details of the API calls.

1 Introduction

In the past decade, many algorithms to compress text have been proposed and many different implementations of these algorithms produced. All recent successful algorithms are adaptive, that is, the encoding of the compressed text is adapted to the text in the message. Often this is done incrementally, the algorithm dynamically changing its behavior as each symbol is encoded. These algorithms have been refined and performance increased with successively more complex and larger models which can be split into two major classes. There are dictionary techniques many of which are variants of the basic Ziv-Lempel algorithms (Bell, Cleary & Witten, 1990). These tend to have high execution speeds and to be relatively simple to implement. Some variants have achieved compression performance close to the very best achieved by any algorithms (Fiala & Green, 1989). The second major class of algorithms are those that estimate the probability of the next symbol in the text sequence. These split the compression process into two steps: first a *modelling* step which estimates the probability and then a coding step that uses the probability to optimally encode the next symbol. The encoding is usually done with some variant of arithmetic coding (Moffat, Neal, & Witten, 1995).

A large number of different estimation and modelling algorithms have been proposed. As shown by Bunton (1996; 1997), most of these can be fitted into the framework of *FSMX* models. In such models, the predictions of the next symbol are conditioned on the basis of a preceding finite sequence of text. In some algorithms such as PPM (Cleary & Witten, 1984; Moffat, 1990), there is a fixed upper bound to the lengths of these contexts, in others (Cleary & Teahan, 1997; Bunton, 1996; Horspool & Cormack, 1986) there is no such prior bound and the contexts can be of unbounded length. Even algorithms such as the Burroughs-Wheeler compressor (Burrows & Wheeler, 1994) which superficially appear very different have been shown to fit into this framework (Cleary & Teahan, 1997). While these algorithms are all conceptually similar, in that they generate probability estimates for each symbol based in the immediately prior context, they also differ considerably in the details of their implementation and the interfaces to these implementations.

Such predictive models can be used for a number of applications other than compression. For example, the problem of ascribing a particular language or style (such as American or British English) to a piece of text can be solved as follows (Teahan, 1998). Construct separate models of American and British text and then compute how well the test text is compressed using these models. The text is ascribed to the same genre as the model which compresses it better. This technique can also be used for authorship ascription and other related problems. Another application is to help with cracking ciphers. Thus if a good model is available of the text which has been encrypted, then the search space for a brute force attack on a code can be significantly reduced (Irvine, 1997).

Another class of related problems is correcting text which has been corrupted in some way. The idea here is to use two models. One a model of "good" text and the other a model of the types of errors that can occur in the original text, sometimes referred to as a *confusion* model. The basic idea is to find the text with maximum probability given the confused text. Such problems include the insertion of spaces back into text where they have been elided (Teahan *et al.*, 1998) and the correction of spelling errors. An even more complex problem is the extraction of text from other data such as OCR data (Teahan *et al.*, 1998) or speech. The principle is the same—two models are used. One model is of the confusion between the original glyphs or phonemes and the text, together with a model of good text.

As has been shown in the references cited above, adaptive probabilistic models can provide better compression (in some cases much better) than more traditional non-adaptive algorithms. This performance depends in turn on the quality of the models themselves. This depends both on the implementation of the models and on the amount of training text used to build the models. For example, to achieve high levels of performance on English text it seems to be necessary to use hundreds of mega-bytes of training text and serious proposals have been made to construct models based on tens of gigabytes of text (roughly the amount of text available on the World Wide Web) (Teahan & Cleary, 1997). Such huge models are very difficult to construct and it would be a great boon to many users if they could be constructed once and then shared with others.

Users of the models do not want to be concerned about the details either of the implementation of the models or how they were trained and the sources of the training text. The problem considered in this paper is how to permit code for different models and actual trained models themselves to be interchanged easily between users. The fundamental idea is that it should be possible to write application programs independent of the details of particular modelling code, that it should be possible to implement different modelling code independent of the various applications, and that it should be possible to easily exchange different pre-trained models between users. It is hoped that this independence will foster the exchange and use of high performance modelling code; the construction of sophisticated adaptive systems based on the best available models; and the proliferation and provision of high quality models of standard text types such as English or other natural languages; and easy comparison of different modelling techniques.

The solution proposed here is that of a standard Application Programming Interface (API) for data compression programs. The API isolates applications and modelling code from each other by providing a standard way to access the needed information.

To meet these high level aims the API should support:

1. isolation of the application from details of the model implementation;

2. isolation of the model code from details of the various applications;

3. as wide a possible range of modelling algorithms;
4. simple implementation of as wide a possible range of applications;
5. trade-offs in the modelling code between space consumption, execution time and accuracy of predictions;
6. models which achieve performance at the highest levels possible;
7. efficient implementation both of models and applications;
8. the ability to load and restore models to and from disk;
9. multiple models in use simultaneously.

The next section introduces some of the issues raised by these requirements and presages the solutions to them in the API. Section 3 introduces the API step by step by considering the various data-structures and features supported and the calls that manipulate them. Section 4 discusses an example implementation of the interface and pragmatic issues raised by the implementation. The paper concludes with a discussion of the major weaknesses of the API and where future standardization work might proceed.

2 Background

2.1 Static and Dynamic Models

One distinction that is made in the API is between static and dynamic models. A static model cannot have its internal model updated while a dynamic model is updated as each symbol of the text is scanned. The main reason for this distinction is a matter of efficiency. It is possible to implement static models significantly more efficiently than dynamic models. There are two reasons for this. One is that the code can be simpler and less information needs to be carried in temporary data structures to access the model. However, the most important reason is that it is possible to make large savings in space. This can be done by compressing the data structure (by removing some pointers, and by using smaller counts, pointers, and symbol sizes) and by *pruning* the model (Bunton, 1996). That is, removing conditioning contexts that contribute little to the overall prediction can dramatically reduce the size of the model while not reducing the predictive performance very much (in fact, in some cases the predictive performance has been reported to improve).

The API allows empty dynamic models to be created. The transition to a static model occurs at the point when the model is written to disk. This can be done in two ways—either as a dynamic model or as a static model. Thus pruned static models can be used both to save memory during execution and to save memory on disk. The latter use is particularly important if models are to be broadcast to the world.

2.2 Probability Calculations

The issue of how to return the probabilistic predictions made by a model is unexpectedly vexed. A naive approach would be to return a floating point value between 0 and 1.0. Unfortunately, this is insufficient to allow precise operation of arithmetic coding. There are two reasons for this. The simpler is that arithmetic coding requires that the interval from 0 to 1 be partitioned into disjoint regions one for each predicted symbol. So both a lower and an upper bound are required for each symbol.

The second problem is that arithmetic coding requires very precise operation to be able to unambiguously decode symbols. Usually, this is done by using rational values for the probabilities. Thus the API uses two integers for a lower and upper probability bound together with a single integer numerator. Unfortunately, this is still insufficient as it is necessary to accommodate the *escape* mechanisms of the PPM and related algorithms (Bell, Cleary & Witten, 1990). The escape mechanism is used to combine predictions from sub-models which use different lengths of prior context to condition the next prediction. This length is referred to as the *order* of the model. Prediction then proceeds from the highest order to the order zero context (no conditioning context is used) or the "order -1" context where all symbols are equally likely. Typically the higher order contexts will only predict a small subset of the symbols with a non-zero probability. But for correct operation of arithmetic coding, all symbols must be given a non-zero probability. This problem is solved by setting aside an *escape probability* which is the probability that the model will need to escape to a shorter context and use that for the remaining predictions. The probabilities of such symbols is obtained by further sub-dividing the escape probability (including possibly further escape probabilities). Now one way to accommodate this would be to form the rational product of the escape probabilities. Experience shows that such products quickly overflow even 32-bit integer numbers. The solution chosen has been for the API to return a sequence of rational probabilities so that each escape can be encoded or decoded separately. This requires a relatively complex set of routines to access these sequences.

2.3 Alphabet Size

Symbols are specified by unsigned integer values which allows a huge range of possible symbols. However, it is often possible to optimise models (especially their use of space) by knowing beforehand what the actual range of symbols is. Thus when a model is created, it is obligatory to specify the number of symbols in the alphabet of the model (the symbols range from 0 up to one less than the number specified). It is strongly recommended that this size be given accurately as some implementations will break when the alphabet is too large. The most typical alphabet sizes will be 128 or 256 corresponding to 7-bit or 8-bit ASCII. However a wide range of possibilities occur from 2 for bit-wise applications, 4 for the nucleic acids in DNA, to many thousands for systems which encode natural language text on a word by word basis. Different implementations may well be suited to different alphabet sizes.

3 The Interface

This section explains the abstract data structures used by the API and the procedures that manipulate them. The procedures are introduced in the order of the complexity of the concepts used by them. An alphabetical listing of all the procedures is given in the appendix.

3.1 Model

The fundamental idea in the API is a model. Each model is self-contained and provides predictions for a sequence of symbols. All models in the API are referred to indirectly via either an integer *model number* or a context record (see Section 3.3).

New model numbers are generated by `SMI_create_model()` which creates a new empty dynamic model and by `SMI_load_model()` which loads a (static or dynamic) model from a

file. `SMI_create_model()` includes parameters for the alphabet size, a title and a parameter string. The title is intended to be a short string that is human readable and describes the provenance of the model. If a model is later re-loaded from disk, then the title may be used to remind the user of its origins. The parameters string is intended to be used for parameters that are implementation dependent. For example, the implementation described below uses this to specify the maximum order of the model. `SMI_get_model()` can be used to access all this information as well as whether the model is currently static or dynamic.

`SMI_write_model()` writes a dynamic model to a file. In the process, it can be left as dynamic or be converted to a static model. (This latter process may be computationally expensive).

`SMI_dump_model()` produces a human readable version of the internal structure of the model and is intended mainly for debugging.

`SMI_release_model()` releases both the memory associated with the model and its model number. It is good form to call this before exiting a program but it can also be used in situations where memory is tight and it is wanted to release one model before loading another.

3.2 Model Numbers

The API manages the model numbers internally. It allocates them when a model is loaded or created and de-allocates them when a model is released. It is often necessary to scan all the models, for example, to release all their memory or to dump them all for debugging. To relieve the user of the burden of keeping track of which model numbers are currently valid, API routines are provided for this. This is done via an implicit counter which is scanned in order through all the models. `SMI_reset_modelno()` sets this counter to the first valid model and returns it. `SMI_next_modelno()` steps to the next model number and returns it or `NIL` if no valid model numbers remain. `SMI_numberof_models()` returns the number of currently valid models. It can be written in terms of the previous two calls but is provided as a convenience.

3.3 Context Record

The most common way of gaining access to a model is via a *context record*. Almost all manipulation and usage of a model is via one of these records. There can be many context records all referring to (different parts of) a model at the same time. The context record provides a reference to a conditioning context within a model and is "stepped along" as new symbols are scanned from a symbol sequence. The context record maintains and provides access to the following information within a model:

- *current position*- a reference to a conditioning context within the model;
- *current symbol* - a reference to a symbol that is predicted by the current position;
- *current probability* - a reference to a probability within the list of probabilities associated with the current symbol;

`SMI_create_context()` creates a new context record and initializes the current position to the null string. `SMI_copy_context()` creates a new context record that is a copy of the specified context record. `SMI_release_context()` releases the memory associated with a context record.

The current symbol forms part of a sequence of symbols predicted by the current position. This sequence is ordered (although the ordering is implementation dependent and will vary depending on the current position and even between different times for the same current position). Procedures are provided for stepping through this sequence so that all the predicted symbols can be accessed. `SMI_reset_symbol()` sets the current symbol to the first predicted symbol. `SMI_next_symbol()` steps the current symbol to the next predicted symbol and returns `NIL` if the end of the sequence has been reached. While for some applications (such as decompression) it is necessary to step sequentially through the symbols, for many others it is more useful to go directly to some known predicted symbol. `SMI_find_symbol()` moves the current symbol directly to a specified symbol. In principle it could be implemented in terms of the two earlier routines. However, it is important that this operation be done quickly and many implementations will be able to provide it much more efficiently than doing a sequential scan.

Once the correct predicted symbol has been located, it is necessary to update the current position by stepping along to the next symbol in the sequence. The procedure `SMI_update_context()` does this by moving the current position to include the current symbol. The current symbol is then reset to be the first symbol of the new current position. For example, in a model where the maximum order is 4, the current position refers to the sequence of symbols "ABCD" and if the current symbol is "Z" then after the update, the current position will be "BCDZ".

As explained earlier, it is necessary to associate a list of probabilities with each symbol. Again procedures are provided for resetting to the first probability in the sequence, `SMI_reset_prob()` and to step to the next probability in the list, `SMI_next_prob()`. The same conventions as before apply here—the procedure returns a `NIL` when the end of the probability list is reached.

In some applications such as decoding, the bulk of the time will be spent repeatedly scanning the probability lists after stepping to a new symbol. To make this process more efficient, a procedure `SMI_nextnew_prob()` is provided. This makes use of the fact that in implementations that use escape mechanisms, many probability lists will share common heads. Thus, it is not necessary to rescan the entire probability list from the beginning—it is sufficient to remember it from the previous symbol and just restart scanning from the first point where the lists differ. The procedure sets the current probability to the correct position and returns the integer position in the list.

Many applications use models to compute the cost of encoding some known text. In this case it is not necessary to scan the different symbols nor to be concerned with the details of the probability lists. So as a convenience, the procedure `SMI_codelength_symbol()` is provided. It searches for the specified symbol and returns the cost of encoding it (i.e. the negative log of the product of the probabilities on its probability list in units of bits).

Another utility procedure that could be written in terms of the other procedures is `SMI_codelength_prob()`. This returns the cost of encoding the current symbol.

3.4 Identifying Current Position

While the intent of the API is to hide as much as possible of the internal structure of the models, there is one application where it seems as if this rule might have to be broken. When using Viterbi style algorithms (Viterbi, 1967) to compute minimal entropy sequences, it is necessary to know when two different paths have converged to the same conditioning context (so that thereafter the two paths will never diverge and only the lower entropy sequence need be considered, the other can be discarded). To do this, there needs to be

some way of deducing when two current positions in different context records are identical. The user could do this themselves for models with known orders, but this is complex, duplicating work done by the models themselves and is in any case impossible for more complex implementations that use unbounded orders. One way to do this would be to allow the user to see a memory pointer to the internal data structures which represents the different conditioning contexts. This is fraught with dangers and difficulties and breaks many of the isolation requirements for the API. `SMI_context_position()` returns an integer which is in a unique one-to-one correspondence to the internal conditioning contexts in the model. While the model may return a pointer to memory here, the API is deliberately agnostic about what is in fact returned. The user may make no presumptions about the value other than that if two current positions have the same integer value, then they are identical. For safety, the implementation might hash the value or divide by some convenient factor like the word length.

4 An Example Implementation

An implementation of the API has been built based on the PPM text compression scheme (Cleary & Witten, 1984), using order 5 models, escape method D (Howard, 1993) and deterministic scaling (Teahan, 1998).[1] Order 5 PPMD models have been found to have good predictive performance with manageable memory requirements for a wide range of data, and especially for natural language text (Teahan, 1998).

Any implementation will necessarily require a delicate balancing act between space consumption, execution speed and accuracy of predictions. The internal data structure used in the present implementation is based on the trie with suffix pointers described in Cleary & Teahan (1997). The use of suffix pointers speeds up access to the context records, but at the expense of model space (they typically increase the size of the models by 25%). Suffix pointers also reduce the size of data that needs to be stored with each context record. This is an important consideration for Viterbi style algorithms, where many thousands of contexts may be active at any one time. The current implementation does not optimise this, and as a result, some applications may require substantial memory space.

The size of the models typically occupy about four times the size of the training text. This varies of course with the training text—for example, the size of English text models typically grows with the square root of the training text length (Teahan, 1998). Later implementations of the API have been able to substantially reduce model size (to about two thirds of the training text length) by the removal of suffix pointers and the use of variable length pointers and symbols.

The implementation allows for arbitrary size alphabets. However, performance will degrade substantially for contexts with many symbols (as occurs with word-based alphabets, for example). This is because of the nature of the PPMD algorithm—with the escape mechanism, exclusions require all higher predictions be eliminated from lower order ones. This particular implementation has been targeted mainly for character-based applications (alphabet sizes ≤ 256) where exclusions produce significant gains in predictive performance (5–10%). A possible future enhancement might be to remove exclusions for models with word-sized alphabets, since in these cases predictive performance is typically improved by only 1% (Teahan, 1998). The removal of exclusions also allows for a faster implementation of `SMI_find_symbol()` (by using a binary search of a context's list of symbols, for example), whereas the current method is a linear search.

[1] The source is available on the Web at `http://www.cs.waikato.ac.nz/~wjt`.

5 Future Work

The decision to use "C" as the language for the API was a largely pragmatic one. Most implementations and work in this area seems to have been done in "C". However, "C" has significant deficiencies when attempting to produce a clean interface. For example, the use of a string to specify the implementation dependent parameters for a model is not very elegant. Another deficiency is that the typeing of the system cannot be very strong. Thus model numbers, symbols, and positions are all treated as integers, with essentially no type checking on them at all (errors such as permuting the parameters to a call would not be detected). Another problem with "C" is that it would be difficult to have models with different implementations co-exist within the same program because of name clashes.

A more suitable implementation framework would be Java. There the API could be specified as an *interface*. This would allow different implementations of the interface to include constructors with parameter sets which differed from implementation to implementation and would allow different implementations to co-exist. Also model numbers, context records, symbols etc. could be declared as different object classes permitting strong type checking. The style of access used in the API where the *reset* and *next* routines allow sequential scanning through the different data-structures is a common style in Java (see for example the iterator object class).

Another defect is that the API is silent about error handling. In the absence of any strong or well accepted model for error handling in "C," there seemed no alternative to this. Again, Java provides its standard exception handling mechanism which would allow the API to deal with errors in a standard and well understood way.

Another issue that has not been dealt with is the provision of some standard file format that would allow checking that the model being loaded matches the implementation. This would require agreement on a common labeling scheme that would allow different implementors to uniquely label their files. One scheme that might be possible here is to use the unique http web address of the organisation doing the implementation as a prefix of the unique label followed possibly by version or date information.

This paper is an alpha version of the API. At this point it has been presented for discussion and has only been implemented once within our group. Clearly it needs to be forged more strongly in the fire of different applications and model algorithms. Of particular interest here is whether it is sufficiently abstract to encompass algorithms, such as DMC, which are adaptive and probabilistic but whose underlying structure and implementation is very different.

6 Acknowledgments

We would like to thank Ian Witten, Stuart Inglis, Malika Mahoui and Zane Bray for discussions about this work.

7 References

Bell, T.C., Cleary, J.G. & Witten, I.H. 1990. *Text compression.* Prentice Hall, New Jersey.

Bunton, S. 1996. *On-line stochastic processes in data compression.* Ph.D. thesis, University of Washington.

Bunton, S. 1997. "Semantically motivated improvements for PPM variants." *Computer Journal*, **40**(2/3): 76–93.

Burrows, M. & Wheeler, D.J. 1994. "A block-sorting lossless data compression algorithm." Technical report, Digital Equipment Corporation, Palo Alto, California.

Cleary, J.G. and Witten, I.H. 1984. "Data compression using adaptive coding and partial string matching." *IEEE Transactions on Communications*, **32**(4), 396–402.

Cleary, J.G. & Teahan, W.J. 1997. "Unbounded length contexts for PPM." *Computer Journal*, **40**(2/3): 67–75.

Fiala, E.R. & Green, D.H. 1989. "Data compression with finite windows." *Communications of the ACM*, **32**(4): 490–505.

Horspool, R.N. & Cormack, G.V. 1986. "Dynamic Markov modelling—a prediction technique" in *Proceedings International Conference on the System Services*, Honolulu, Hawaii, IEEE Computer Society Press.

Howard, P.G. 1993. "The design and analysis of efficient lossless data compression systems." Ph.D. thesis, Brown University, Providence, Rhode Island.

Irvine, S.A. 1997. *Compression and Cryptology.* D.Phil. thesis, Univ. of Waikato, N.Z.

Moffat, A., Neal, R. & Witten, I.H. 1995. "Arithmetic coding revisited" in *Proceedings DCC'95*, edited by Storer, J.A. & Cohn, M., pages 202–211. IEEE Computer Society Press.

Moffat, A. 1990. "Implementing the PPM data compression scheme." *IEEE Transactions on Communications*, **38**(11): 1917–1921.

Teahan, W.J. 1998. *Modelling English text.* D.Phil. thesis, Univ. of Waikato, N.Z.

Teahan, W.J. & Cleary, J.G. 1997. "Models of English text" in *Proceedings DCC'97*, edited by Storer, J.A. & Cohn, M., IEEE Computer Society Press.

Teahan, W.J., Inglis, S., Cleary, J.G. & Holmes, G. 1998. "Correcting English text using PPM models" in *Proceedings DCC'98*, edited by Storer, J.A. & Cohn, M., IEEE Computer Society Press.

Viterbi, A.J. 1967. "Error bounds for convolutional codes and an asymptotically optimal decoding algorithm," *IEEE Transactions on Information Theory*, **13**, 260–269.

8 Appendix—all API Calls

```
#define NIL 0
#define SMI_static 0
#define SMI_dynamic 1

typedef unsigned char boolean;

unsigned int SMI_copy_context (unsigned int context);
```

Creates a new context record, copies the contents of the specified context into it, and returns an integer reference to it.

```
unsigned int SMI_create_context (unsigned int model);
```

Creates and returns an unsigned integer which provides a reference to a context record associated with the model's context. The current position is set to the null string. The current symbol is set to the first predicted symbol and the current probability to the first probability.

```
unsigned int SMI_create_model (unsigned int alphabet_size,
             char *title, char *parameters);
```

Creates a new empty dynamic model. Returns the new model number allocated to it if the model was created successfully, `NIL` if not. `alphabet_size` specifies the number of symbols permitted in the alphabet (all symbols for this model must have values from 0 to one less than `alphabet_size`). `title` is intended to be a short human readable text description of the origins and content of the model. `parameters` is intended to hold model information which differs between implementations. For example, the implementation described in Section 4 uses it to specify the maximum order of the PPM model.

```
void SMI_dump_model (FILE *fp, unsigned int model);
```

Prints a human readable version of the model (intended mainly for debugging).

```
void SMI_dump_prob (FILE * fp, unsigned int context);
```

Prints the probability list for the current symbol in a human readable form. It does this without altering the current position in the probability list as determined by the functions `SMI_reset_prob`, `SMI_next_prob` or `SMI_nextnew_prob`.

```
void SMI_find_symbol (unsigned int context, unsigned int symbol);
```

Updates the context record so that the current symbol becomes symbol. The current probability becomes the first probability value for this symbol.

```
void SMI_forcenull_context (unsigned int context);
```

The current context is updated so that the current position is forced to the null string. If the model is dynamic then any model statistics associated with the previous current position and current symbol are also updated. This is useful during training if there are statistics that differ markedly at the start of some text from in the middle of it (for example, individual names, and titles within a long list).

```
unsigned int SMI_getcontext_modelno (unsigned int context);
```

Returns the model number associated with the context.

```
unsigned int SMI_getcontext_position (unsigned int context);
```

Returns an integer which uniquely identifies the the current position associated with the context. (One implementation is to return a memory location corresponding to the current position. This routine is useful if you need to check whether different contexts have encoded the same prior symbols as when checking whether the context pathways converge in the Viterbi or trellis-based algorithms.)

```
boolean SMI_get_model (unsigned int model, unsigned int *model_type,
        unsigned int *alphabet_size, char **title, char **parameters);
```

Returns information describing the model. Returns `NIL` if the model does not exist (and leaves the other parameters unmodified in this case), non-zero otherwise. `alphabet_size`, `title` and `parameters` are the values used to create the model in `SMI_create_model()`. `modeltype` is set to `SMI_dynamic` or `SMI_static` depending on whether the model is static or dynamic.

```
unsigned int SMI_load_model (FILE *fp);
```

Loads a model which has been previously saved to the file into memory and allocates it a new model number which is returned.

```
unsigned int SMI_next_modelno ();
```

Returns the model number of the next valid model. Returns `NIL` if there isn't any.

```
boolean SMI_next_prob (unsigned int context, unsigned int *lbnd,
        unsigned int *hbnd, unsigned int *totl);
```

Places the current probability in `lbnd`, `hbnd` and `totl` (see text for more complete description). Update the context record so that the current probability becomes the next probability in the list of probabilities associated with the current symbol. If there are no more probabilities in the list then return `NIL` otherwise some non-`NIL` value.

```
void SMI_nextnew_prob (unsigned int context, unsigned int pos);
```

Compares the current probability list with the probability list associated with the prior predicted symbol. The current probability becomes the first probability (on the current symbols probability list) which differs between the two lists. The integer pos records this position on the list (starting with 0 for the first position). `SMI_next_prob` can be used to return the probability. In the special case of the first predicted symbol the current probability becomes the first on the current probability list.

```
boolean SMI_next_symbol (unsigned int context, unsigned int *symbol);
```

Updates the context record so that a new predicted symbol becomes the current symbol. If a sequence of these calls are made, every symbol in the alphabet will be visited exactly once although the order in which they are visited is undefined being implementation and data dependent. The first probability value for this symbol becomes the current probability.

```
unsigned int
SMI_numberof_models ();
```

Returns the number of currently valid models.

```
void SMI_release_context (unsigned int context);
```

Releases the memory allocated to the context and the context number (which may be reused in later `SMI_create_context` or `SMI_copy_context` calls).

```
void SMI_release_model (unsigned int model);
```

Releases the memory allocated to the model and the model number (which may be reused in later `SMI_create_model` or `SMI_load_model` calls).

```
void SMI_reset_modelno ();
```

Resets the current model number so that the next call to `SMI_next_modelno` will return the first valid model number (or `NIL` if there are none).

```
void SMI_reset_prob (unsigned int context);
```

Resets the position in the list of probabilities associated with the current symbol. The next call to `SMI_next_prob` will return the first probability on the list.

```
void SMI_reset_symbol (unsigned int context);
```

Resets the context record to point at the first predicted symbol of the current position.

```
void SMI_update_context (unsigned int context);
```

The referenced context record is updated so that the current symbol referred to by the context record becomes the last symbol of the current context position. If the model is dynamic then the internal statistics in the model are also updated. (Note that only one active context is permitted for each dynamic model, whereas there is no limit on the number of active contexts associated with a static model.)

```
void SMI_write_model (FILE *fp, unsigned int model, unsigned int model_type);
```

Writes out the model to file FP (which can then be loaded by other applications later). `model_type` must have the value `SMI_static` or `SMI_dynamic` and determines whether the model is static or dynamic when it is later reloaded using `SMI_load_model`.

8.1 Utility Procedures

The following procedures can all be written in terms of other routines in the API but are included either for convenience or because an implementation may be able to execute them faster with a specialised version.

```
void SMI_dump_models (FILE *fp);
```

Writes a human readable version of all the currently valid models to the file.

```
float SMI_codelength_prob (unsigned int context);
```

Returns the code length of the current probability in bits. It does this without altering the current symbol or the current probability.

```
float SMI_codelength_symbol (unsigned int context, unsigned int symbol);
```

Returns the code length of the current symbol in bits. It does this without altering the current symbol or the current probability.

```
void SMI_release_models ();
```

Releases the memory used by all the models.