An Improved Interface for Probabilistic Models of Text

W. J. Teahan
Research Fellow, Information Retrieval Group
School of Computing and Mathematical Sciences,
Robert Gordon University, Aberdeen, Scotland*
wjt@scms.rgu.ac.uk

June 30, 2006

Abstract: Recently, an Application Program Interface (API) has been defined for modelling sequential text (Cleary & Teahan, 1999). The intention of the API was to shield the user from details of the modelling and probability estimation process, with the primary motivation being to simplify the design of applications where the use of textual models is needed, such as data compression, spelling correction or segmentation of text by inserting spaces.

This paper describes modifications to the API that simplifies the interface both from a user's perspective and from an implementation point of view. Some further routines have also been added where this has been felt necessary. The API has also been extended to include a new class of models, called *markup* models. The extended interface substantially simplifies the implementation of a wide range of correction-based applications such as OCR spelling correction and word segmentation.

1 Introduction

Recently, an Application Program Interface (API) has been defined for modelling sequential text (Cleary and Teahan, 1999). The intention of the API was to shield the user from details of the modelling and probability estimation process, with the primary motivation being to simplify the design of applications where the use of textual models is needed, such as data compression, spelling correction or segmentation of text by inserting spaces.

Predictive models can be used for a number of applications. 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).

The problem considered by Cleary & Teahan (1999) was how to permit code for different models and the actual trained models themselves to be interchanged easily between users. 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 author would like to acknowledge the generous support of the Information Technology Department at Lund University, Sweden where much of the work described in this paper was undertaken from January to September, 1999.

The fundamental ideas they put forward 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 was hoped that this independence would 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 they proposed was 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, they noted that it should support the following goals:

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.

Cleary & Teahan built an implementation of the API based on the PPM text compression scheme (Cleary & Witten, 1984; Moffat, 1990; Cleary & Teahan, 1997) using order 5 models.

This paper describes modifications to the original API specification that simplifies the interface both from a user's perspective and from an implementation point of view. Some further routines have also been added where this has been felt necessary. The API has also been extended to include a new class of models, called *markup* models. The extended interface substantially simplifies the implementation of a wide range of correction-based applications such as OCR spelling correction and word segmentation.

This paper is organized as follows. The next section gives a brief overview of the abstract data structures used by the API. Section **??** describes the changes that have been made to the API, and lists all the routines that have either been added, removed or modified. The new specification can be found in Appendix A. Section **??** then describes the two main extensions to the API—the first consists of routines for creating, comparing and modifying text records (i.e. strings of symbols), and the second describes routines for creating and using the new class of markup models. The specification of these routines can be found in Appendices B and C. The paper concludes with some sample C source code for six applications, and a brief look at where future work on the API might proceed.

2 An overview of the API

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 below). 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.

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.

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;

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 either for stepping through this sequence so that all the predicted symbols can be accessed, for going directly to some known predicted symbol, and for updating the current position by stepping along to the next symbol in the sequence once the correct predicted symbol has been located. Since many applications use models to compute the cost of encoding some known text, procedures are also provided that compute the cost of encoding the symbols (i.e. the negative log of the product of the probabilities in units of bits).

3 What's different

This section details the modifications that have been made to the API. The first four sections describes the four major changes that have been made—firstly, the addition of a new object, called a *coder* for specifying the arithmetic coder; secondly, the use of "concurrent" updating; thirdly, the extension to allow for alphabets of unbounded size; and lastly, the creation of a special "sentinel" symbol.

The last three sub-sections list the modifications that have been made to the API specification itself, detailing the routines that have been added, removed or modified.

3.1 Coders

In the specification of the old API, it was noted that "the issue of how to return the probabilistic predictions made by a model is unexpectedly vexed." The solution put forward was to return the predictions as a probability list—a list of three integers specifying the arithmetic coding range as a lower and upper probability bound together with a single integer numerator total.

This solution was less than satisfactory, for the following reasons:

1. Substantial storage overhead was required to keep track of the cumulative counts for the probability lists for the correct operation of routines such as `SMI_next_symbol`.

2. Encoding and decoding operations were slower because they had the extra overhead of maintaining the probability lists.

3. There was also no way during decoding to take advantage of faster search methods (such as binary search trees) during operation of the search routine `SMI_find_symbol`.

4. From a user point of view, the interface was less than satisfactory, being unintuitive and needlessly complicated.

The solution proposed in this specification is to remove probability lists altogether from the API. This is achieved by the use of a new object, called a *coder*, which allows the user to explicitly specify the arithmetic coding routines to be used for encoding and decoding. A coder is defined using the routine `TLM_create_coder` which returns a coder number as a pointer to a coder record. The API manages the coder numbers internally. It allocates them when a coder is created and de-allocates them when a coder is released.

The routine `TLM_create_coder` takes four arguments: the maximum frequency allowed; and pointers to three routines that are required for encoding and decoding—`arithmetic_encode` and `arithmetic_decode` both take three unsigned integers as arguments that specify the current arithmetic coding range (low, high and total), whereas `arithmetic_decode_target` takes just a single unsigned integer as an argument, the total of the current coding range. (This coding interface was first proposed by Moffat, Neal, and Witten (1995) for their arithmetic coding implementation).

Two new routines, `TLM_encode_symbol` and `TLM_decode_symbol` have been added to the API to allow the user to explicitly perform the encoding and decoding operations for a given context and coder. Since the arithmetic coding functions are specified in the coder record passed to these routines, the user no longer needs to know what probabilities are actually being coded. The coding operations are done internally; thus, the user is completely shielded from the implementation details of probability counts, scaling and the like.

Many applications do, however, require access to the probability predictions for a given context. This is achieved in the new API by modifying the operation of the existing routines `TLM_next_symbol` and `TLM_find_symbol`, and providing an additional routine `TLM_update_codelength`. These routines return a *codelength*, a real value (float) which is set to the cost of encoding the current context symbol. One slight drawback with this approach is that a double search cost is incurred for certain applications which require a scan through the probability distribution prior to encoding and decoding. For example, `TLM_next_symbol` might be used to search through the probability distribution, and then `TLM_encode_symbol` subsequently called once a particular symbol has been located (such as the most probable symbol, say). However, in practice, such applications are rare, and the extra search cost is more than offset by the gains in speed that result from the combination of the API changes described in this and the following sections.

3.2 Concurrent updating

The new API supports a fundamental change, called *concurrent* updating, in how the model updating mechanism is performed. In the old API, the updating mechanism, performed by the routine `SMI_update_context`, was separate from the search functions `SMI_next_symbol` and `SMI_find_symbol`. Although this offered a functionality that was both elegant and concise, the actual implementation was much less so. An implementation of the old API by necessity had to maintain substantial local storage to keep track of the current context in order to allow for separate updates. The reason for this was that an implementation could not know in advance when executing a search function whether an update would ultimately be performed. In a PPM implementation based on tries, for example, a search function may need to escape to other nodes in the trie, as well as maintain its current position in the current node's list of symbols. Keeping track of all this information is needlessly complicated as well as expensive both in terms of on-line memory and execution speed since it needs to be done for every symbol in the sequence compounded by every node in the trie that needs to be visited for each symbol.

Concurrent updating avoids all of this. Since all nodes on the current active context list need to be updated (and not just the node which contains the target symbol), it is simplest to do this concurrently as the search for the target symbol is being performed. Hence, the routine `SMI_update_context` can be discarded altogether, with its functionality replaced by routines that perform both searching and updating simultaneously.

In the new API, the following four "update" routines are provided:

- `TLM_encode_symbol`
- `TLM_decode_symbol`
- `TLM_update_symbol`
- `TLM_update_codelength`.

The target symbol must be specified for each of the routines. `TLM_update_symbol` updates the context for the target symbol, while the other three also update the context concurrently while performing additional work such as encoding or decoding the target symbol, or returning is codelength.

In contrast, the following two "search" routines do not do any updating, and instead return only the codelength for the target or next symbol.

- `TLM_find_symbol`
- `TLM_next_symbol`.

3.3 Unbounded alphabets

In the old API implementation, there was no elegant way of dealing with word-based alphabets—alphabets where the size is essentially unbounded. In order to accommodate this, the new API provides support for a special model class, one where the alphabet is *incrementally expanding*. For this class of model, whenever a previously unseen symbol is encountered, the maximum symbol number in the alphabet is incremented by 1.

Setting the argument `alphabet_size` to 0 for the routine `TLM_create_model` specifies that the alphabet for the new dynamic model is unbounded. In this case, allowable symbol numbers range from 0 up to a special `expand_alphabet` symbol which is equal to the current maximum symbol number (this is one more than the highest previously seen symbol number). If the current symbol becomes the `expand_alphabet` symbol, then the current maximum symbol number is incremented by 1, thus effectively expanding the size of the alphabet by 1. The current maximum symbol number may be obtained by calling the routine `TLM_get_model`.

3.4 The sentinel symbol

It is a well-known fact that statistics differ markedly both between and within different genres of text (Witten *et al.*, 1999). For example, statistics for a person's name differ markedly from dates; similarly, a capital letter S is much more likely to occur at the start of a name than in the middle or end of it.

Included in the old API was an unusual routine `SMI_forcenull_context` whose purpose was rather obscure. The effect of this routine was to update the current context so that the current position was forced to the null string. It's purpose was to improve the prediction when multiple models were being used to predict different genres of text. Prediction was found to significantly improve whenever the text genre changed if the prior context was discarded, thus "forcing" the model to use a different set of statistics, namely only for those symbols that occurred at the start of the new text genre, rather than for those that occurred anywhere within it.

The solution to this problem within the new API is to introduce a special symbol, called the *sentinel* symbol. The sentinel symbol is a common device used to simplify the handling of sequential data in many string-based algorithms such as those based on suffix trees (Larsson, 1999). The sentinel symbol is useful where there is a break required in the updating of the context, such as when the end of string has been reached or when more than one model is being used to encode different parts of a string, or when during training there are statistics that differ markedly at the start of some text than in the middle of it (for example, individual names, as explained above, or titles within a long list).

A further routine has been added to the new API, called `TLM_sentinel_symbol`, which returns an unsigned integer that uniquely identifies the sentinel symbol. The counterpart routine `SMI_forcenull_context` from the original API has been removed in the new implementation since it is no longer needed. The effect of encoding the sentinel symbol is that the prior context is forced to the null string i.e. the subsequent context will contain just the sentinel symbol itself. This routine is usually used with the update and search routines (see section **??**). For example, the following source code

will update the context record so that the current symbol becomes the sentinel symbol: `TLM_update_symbol (context, TLM_sentinel_symbol ())`.

The importance of this change should not be underestimated—not only does it provide extended functionality, it also simplifies the API from a user's perspective. The major benefits, however, are that it greatly simplifies certain implementation details, and provides a means for overcoming some difficult coding problems while at the same time suggesting new methods. To illustrate, the sentinel symbol can be used to improve the prediction for a word based encoder. Encoding an extra sentinel symbol at the end of each sentence improves compression by one per cent—this is not an inconsequential amount considering the amount of effort that has already been expended in achieving the encoder's prior performance.

3.5 Routines that have been added

The following routines have been added to the API:

- `TLM_encode_symbol`
- `TLM_decode_symbol`
- `TLM_update_symbol`
- `TLM_sizeof_model`
- `TLM_minlength_model`.

The later two routines are useful utility routines for determining the current on-line memory requirements for a model, and it's offline minimum size (the later routine can be used in calculating minimum description lengths, for example).

3.6 Routines that have been removed

All routines needed to handle probability lists have been removed. These include:

- `SMI_dump_probs`
- `SMI_next_prob`
- `SMI_nextnew_prob`
- `SMI_reset_prob`
- `SMI_entropy_probs`.

The following routines have also been removed:

- `SMI_update_context`
- `SMI_forcenull_context`.

3.7 A list of modifications

The following is a list of the remaining modifications made to the API:

- All routines now have the `TLM_` prefix instead of `SMI_`.
- The routines `TLM_next_symbol` and `TLM_find_symbol` have an extra argument which is set to the cost in bits of encoding the symbol given the current context.
- The routine `TLM_update_codelength` replaces `SMI_codelength_symbol`. It has the same functionality as the new routine `TLM_update_symbol`, but additionally calculates the cost of encoding the symbol as it updates the context to point to the specified symbol.
- The result of a subsequent call to `TLM_next_symbol` will no longer be affected by a call to `TLM_find_symbol`. Similarly, a call to `TLM_find_symbol` and other routines will have no effect on the behavior of `TLM_next_symbol`.
- `SMI_entropy_symbol` has been removed since technically the value being returned was the length of encoding the symbols in bits, and not strictly the entropy. This value is now returned by the following three routines: `TLM_update_codelength`, `TLM_find_symbol` and `TLM_next_symbol`.
- The argument `new_title` has been added to the `TLM_load_model` function.
- An extra argument `dump_symbol_function` has been added to the `TLM_dump_model` and `TLM_dump_models` routines. This argument is a pointer to a user-defined function for printing symbols.

4 Extending the API

The original conception of the API was to provide a powerful, though restricted, interface for the probabilistic modelling of sequential text. During the evolutionary process accompanying its development, however, it became apparent that broadening the scope of the API to encompass a much wider range of text operations would be beneficial. For example, routines were needed to support the manipulation of sequences of symbols (similar to the string functions in the C programming language). As well, it was found that many of the API-based programs devised for a diverse class of correction-type applications such as spelling correction, name location and word segmentation had remarkable similarities in the underlying source code—despite the programs being long and complex, only a few changes were needed to define the key differences between the applications. It was felt that these similarities could be encapsulated in some manner to create a much more powerful specification.

In light of this, a number of extensions have been made to the API, the full description of which can be found in Appendices A, B and C. To distinguish the new and modified routines with the ones in the previous API specification, the routines have been grouped into three sub-libraries, one for the language modelling routines, one for text manipulation and the third to help with the correction-based algorithms. Each has been assigned a unique prefix to identify them—`TLM_`, `TXT_` and `TMM_` respectively. An overview of the latter two sub-libraries are described in the next two sub-sections.

4.1 Text routines

A new object, called a *text record*, has been added to the API to support the manipulation of sequences of symbols. A text record is created using the routine `TXT_create_text` which returns a text number as a pointer to a text record. The API manages the text numbers internally. It allocates them when a text record is created and de-allocates them when a text record is released.

The API supports a sub-library of routines for comparing and copying text records and for loading and dumping the text to and from a file. The sub-library also supports routines for manipulating symbols contained in the text records—symbols can be inserted into and extracted from the text, for example. There are also routines for determining the symbol type such as `TXT_is_alphabetic`, `TXT_is_nemeric` and `TXT_is_punct`. These routines reproduce the functionality of the string routines from the C programming language. Many of the routines, in fact, have an exact counterpart, for example the boolean functions (such as `TXT_is_alphabetic` and `TXT_is_punct`).

4.2 Markup models

The extended API also introduces a new class of model, called a *markup* model, that can be used for certain types of applications which involve correcting or "marking up" text in some manner. The essential idea here is that the output text can be considered to be a corruption of the source text—this is based on a common framework for the statistical modelling of many natural language applications using a theory developed by Shannon (1948) to model a noisy communication channel (Teahan *et al.*, 1998). The basic idea is to find the text with maximum probability given the observed output text. This is performed using a dynamic programming search algorithm such as the Viterbi (1967) algorithm.

With this theory, the idea 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. Applications include the insertion of spaces back into text where they have been elided and the correction of spelling errors. An even more complex problem is the extraction of text from other data such as OCR data 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 the good text.

Witten *et al.* (1999) apply the same approach to text mining—they show how a broad range of patterns such as names, dates, email addresses and URLs can be located in text. The key idea here is that the training data for the model of the good text contains text that is already "marked up" in some manner. The correction process is then performed to recover the most probable source text (with special markup symbols inserted) from the unmarked-up observed text.

These ideas provide us with an extremely powerful method for transforming text. The essential differences between quite diverse applications can be characterized by a few transformations as illustrated by Table **??**. Listed in the left hand column of the table are a number of applications; a terse representation of the transformations that uniquely characterize each application is given in the right hand column. These are the transformations that are performed when the most probable source text is being "recovered" from the observed text. The form *observed text* $\mapsto$ *corrected text* is used to denote a transformation from the observed to the corrected text; for example, $El \mapsto H$ denotes that that bigraph El is corrected to the letter H. The symbol $\star$ is used to denote the "wildcard" symbol—this will match any symbol in the observed text; the α symbol will match just alphanumeric

Application	**Markup transformations**
Chinese word segmentation	$\star \mapsto \star$; $\star \mapsto \sqcup\star$.
English word segmentation	$\star \mapsto \star$; $\alpha \mapsto \sqcup\alpha$.
Alphabetic input from a mobile telephone	$2 \mapsto$ [abcABC]; $3 \mapsto$ [defDEF]; $4 \mapsto$ [ghiGHI]; $5 \mapsto$ [jklJKL]; $6 \mapsto$ [mnoMNO]; $7 \mapsto$ [pqrsPQRS]; $8 \mapsto$ [tuvTUV]; $9 \mapsto$ [wxyzWXYZ].
Spelling correction	$\star \mapsto \star$; $\alpha \mapsto$[a...zA...Z0...9]; $\alpha \mapsto \alpha$[a...zA...Z0...9]; ...
OCR spelling correction	$\star \mapsto \star$; c $\mapsto$ e; e $\mapsto$ c; m $\mapsto$ n; m $\mapsto$ ni; n $\mapsto$ m; ni $\mapsto$ m; El $\mapsto$ El; El $\mapsto$ H; H $\mapsto$ El; ...
Language identification	$\star \mapsto \star$; $\star \mapsto$ {*English, German, Spanish ...*}$\star$.

Table 1: Some applications and their markup transformations

symbols. Ranges of symbols are denoted by the symbols between square brackets ([...], as used for Unix regular expressions). Special *model symbols* are denoted between braces ({...})—these symbols signal to the transformation process to insert a sentinel symbol into the corrected sequence (to terminate the coding of the prior context) after which all subsequent symbols will be coded using the new model number (up until the next sentinel symbol is encountered).

As a further explanation of the table, the Chinese word segmentation application can be characterized by just two markup transformations—one which keeps the input symbol unchanged, and another which inserts an extra space before it (denoted by the $\sqcup$ symbol). For English word segmentation, the space insertion only occurs for alphanumeric characters. For the alphabetic input from a mobile telephone application, ranges of symbols are used to correct each digit into its corresponding alphabetic equivalent; similarly for spelling correction, the ranges are used to denote that each alphanumeric character can be replaced or followed by any other alphanumeric character. The markups listed for the OCR spelling correction application are a small sample of typical confusions taken from Teahan *et al.* (1998). Finally, the language identification application specifies that the language model being used to predict future symbols should change for every symbol in the observed sequence.

The extended API supports this powerful formulation in the following manner. A sub-library of routines (see Appendix C) has been added that can be used especially for correcting text. These routines make use of a new object, called a *markup record*, which is created using the routine `TMM_create_markup`. Like the other API creation routines, this routine returns a markup number as a pointer to a markup record. The API manages these numbers internally and allocates them when a markup record is created and deallocates them when a text record is released. A markup sequence is initiated by the `TMM_start_markup` routine—it takes an argument `language_model` which is the model number of the language model to be used to predict the corrected sequence. Multiple calls to this procedure will cause separate searches to be initiated for each of the specified language models. This routine must be called at least once before the routine `TMM_perform_markup` is called, which is used to perform the markup search process itself. This routine returns the number of a newly created text record that contains the corrected text.

The specification of the markup transformations that define the markup model is accomplished by the `TMM_add_markup` routine. This takes three arguments—`codelength`, a real value (float) which is set to the cost in bits of making the correction; and two for-

mat specifications which define the markup transformation, `observed_text_format` and `markup_text_format`, the formats of the observed and corrected texts respectively.

The codelength value determines how well the transformation will perform against the other transformations during the search process. In some applications (such as word segmentation), a value of 0 may be sufficient. Other applications may require the codelength to be determined from training data; for example, for OCR spelling correction, the codelength of each of the transformations is calculated from frequency counts tabulated from common confusions made from typical OCR output.

The format specifications take the same form as that used for the well-known C "printf" routines. These are composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the corrected text); and conversion specifications which follow the character %, each of which results in fetching zero or more subsequent arguments. There are conversion characters which enable the specification of wildcard and non-ASCII symbols, as well as range symbols and model symbols (as described above). Special *function* symbols also allow boolean functions such as `TXT_is_alphanumeric` to be used in the specification.

5 Some sample C source code

This section gives sample C source code for the following six applications:

1. a character-based encoding procedure;
2. a character-based decoding procedure;
3. a procedure for building ("training") static and dynamic models;
4. a procedure for finding the "best" model by calculating the codelength required to compress some input text (this can be used to identify the natural language in which the input text is written, or to identify the most likely author of the text, for example);
5. a procedure that can segment either English or Chinese text into words; and
6. a procedure that can be used to translate numeric input keyed from a mobile telephone into alphabetic input.

5.1 A character based encoder

The following procedure encodes characters from the standard input file. The argument `model` specifies the model number of the language model—this number has been returned by an earlier call to either the routine `TLM_create_model` (for a new dynamic model) or to `TLM_load_model` (for a static or dynamic model loaded from disk). The argument `coder` specifies the coder number of the arithmetic coder that is used to perform the arithmetic encoding and decoding (this is returned by the `TLM_create_coder` routine). The sentinel symbol has been used to encode the eof character.

```
void encode_text (unsigned int model, unsigned int coder)
/* Encodes the text using the model and arithmetic coder. */
{
  unsigned int context, symbol;
  int cc;
```

```
  context = TLM_create_context (model);
  for (;;) {   /* repeat until EOF */
    cc = getc (stdin);
    if (cc != EOF)
      symbol = cc;
    else
      symbol = TLM_sentinel_symbol ();
    TLM_encode_symbol (context, coder, symbol);
    if (symbol == TLM_sentinel_symbol ())
      break;
  }
  TLM_release_context (context);
}
```

5.2 A character based decoder

The following decoding procedure is equivalent to the encoding one above—it decodes characters from the standard input file. Like above, the arguments `model` and `coder` specify the language model and arithmetic coder being used; and the sentinel symbol is used to decode the eof character.

```
void decode_text (unsigned int model, unsigned int coder)
/* Encodes the text using the model and arithmetic coder. */
{
  unsigned int context, symbol;

  context = TLM_create_context (model);
  for (;;) {   /* repeat until sentinel symbol */
    symbol = TLM_decode_symbol (context, coder);
    if (symbol == TLM_sentinel_symbol ())
      break;
  }
  TLM_release_context (context);
}
```

5.3 Building a static or dynamic model

The following procedure updates the model using text from standard input, and writes the model to standard output. It is similar to the encoding and decoding procedures above, except that it uses the routine `TLM_update_symbol` to update the model instead of either `TLM_encode_symbol` or `TLM_decode_symbol` as above. Similarly, the argument `model` specifies the language model being used; and the sentinel symbol is used for the eof character. The boolean argument `static` specifies whether the model should be written out as a static or dynamic model when finished.

```
void train_model (unsigned int model, boolean static)
/* Trains the model from the characters obtained from standard input. */
{
  unsigned int context, symbol;
  int cc;
```

```
  context = TLM_create_context (model);
  for (;;) {   /* repeat until EOF */
    cc = getc (stdin);
    if (cc != EOF)
      symbol = cc;
    else
      symbol = TLM_sentinel_symbol ();
    TLM_update_symbol (context, symbol);
    if (symbol == TLM_sentinel_symbol ())
      break;
  }
  TLM_release_context (context);

  if (static)
    TLM_write_model (stdout, Model, TLM_static);
  else
    TLM_write_model (stdout, Model, TLM_dynamic);
}
```

5.4 Identifying the "best" model

The following procedure finds the model that is best able to compress text from standard input. It assumes that a number of models have already been created or loaded using either the `TLM_create_model` or `TLM_load_model` routines. The procedure also demonstrates the use of various text routines to load and obtain the symbols from the standard input.

If the models have been trained on natural language texts (such as text that is representative of English, German or Spanish, say), then this code will be able identify the natural language of the input text with a high degree of accuracy. The method is also quite successful at distinguishing dialects such as American and British English. On the other hand, if the models have been trained on text that is representative of the writing styles of a number of different authors, then the code will be able to identify the most likely author of the text instead.

```
void find_best_model ()
/* Finds the ``best'' model at compressing the characters obtained
   from standard input. */
{
  unsigned int model, source_text, pos;
  float codelength, codelen, min_codelength;
  char *title, min_title;

  min_codelength = 0.0;
  min_title = NULL;
  source_text = TXT_load_text (stdin);
  TLM_reset_modelno ();
  while ((model = TLM_next_modelno ()))
  { /* Compute codelength for each language model */
    codelength = 0.0;
```

```
    context = TLM_create_context (model);
    pos = 0;
    while (TXT_getsymbol_text (source_text, pos++, &symbol))
      { /* Calculate codelength for each symbol */
        TLM_update_codelength (context, symbol, &codelen);
        codelength += codelen;
      }
    TLM_release_context (context);
    title = TLM_get_title (model);
    if ((min_codelength == 0.0) || (codelength < min_codelength)) {
      min_codelength = codelength;
      min_title = title;
    }
    printf ("%-32s %9.3f\n", title, codelength);
  }
  printf ("Minimum codelength for %s model\n", min_title);
}
```

5.5 English and Chinese word segmentation

The following procedure demonstrates the use of a simple markup model for segmenting words in either English or Chinese text. The markup model used in both cases first generates one markup for each symbol in the input, then generates an alternative markup by prepending a space before each input symbol. This is done for all input symbols for Chinese text, but only for alphabetic symbols for English text. The argument `language_model` specifies the model number of the language model being used for the segmentation. The boolean argument `English` indicates whether the model should segment for English or Chinese text.

```
void segment_text (unsigned int language_model, boolean English)
/* Segments the text from standard input using the language model. */
{
  unsigned int markup_model, source_text;

  markup_model = TMM_create_markup ();
  TMM_add_markup (markup_model, 0.0, "%w", "%w");
  if (!English)
    TMM_add_markup (markup_model, 0.0, "%w", " %w");
  else
    TMM_add_markup (markup_model, 0.0, "%f", " %f", TXT_is_alphanumeric);

  source_text = TXT_load_text (stdin);

  TMM_start_markup (markup_model, language_model);
  markup_text =
      TMM_perform_markup (markup_model, language_model, source_text);
  TXT_dump_text (stdout, markup_text, TXT_dump_symbol);

  TMM_release_text (source_text);
```

```
  TMM_release_text (markup_text);
  TMM_release_markup (markup_model);
}
```

5.6 Translating alphabetic input from a mobile telephone

The following procedure demonstrates the use of a markup model for translating numeric input keyed in from a mobile telephone into it's most likely alphabetic equivalent. The markup model generates the alphabetic alternatives for each numeric character found in the text obtained from standard input, and the most likely translation is written out to standard output. The argument `language_model` specifies the model number of the language model being used.

```
void translate_text (unsigned int language_model)
/* Translate the numeric text into alphabetic text using the language
   model. */
{
  unsigned int markup_model, source_text;

  markup_model = TMM_create_markup ();
  TMM_add_markup (markup_model, 0.0, "2", "%[abcABC]");
  TMM_add_markup (markup_model, 0.0, "3", "%[defDEF]");
  TMM_add_markup (markup_model, 0.0, "4", "%[ghiGHI]");
  TMM_add_markup (markup_model, 0.0, "5", "%[jklJKL]");
  TMM_add_markup (markup_model, 0.0, "6", "%[mnoMNO]");
  TMM_add_markup (markup_model, 0.0, "7", "%[pqrsPQRS]");
  TMM_add_markup (markup_model, 0.0, "8", "%[tuvTUV]");
  TMM_add_markup (markup_model, 0.0, "9", "%[wxyzWXYZ]");

  source_text = TXT_load_text (stdin);

  TMM_start_markup (markup_model, language_model);
  markup_text =
      TMM_perform_markup (markup_model, language_model, source_text);
  TXT_dump_text (stdout, markup_text, TXT_dump_symbol);

  TMM_release_text (source_text);
  TMM_release_text (markup_text);
  TMM_release_markup (markup_model);
}
```

6 Future directions

There are a number of directions in which future work with this API could proceed. The following is a short list of a few possibilities, some of which are currently in progress:

- It was noted in the paper describing the original API implementation that the decision to use the C programming language for the API was largely a pragmatic one—Java was posed as an alternative, for the following reasons: it provides special support for

defining interfaces; models and contexts and the like could be defined as different object classes with strong type-checking; and it provides a standard exception handling mechanism that would allow the API to deal with error-handling in an elegant way. Java would also overcome, to some extent, the problems of portability.

- Work is proceeding at improving the API's efficiency in a number of areas, in terms of memory usage, execution speed, predictive performance and accuracy. For example, it is possible to substantially reduce the offline storage space for static models, as well as online storage for both static and dynamic models. Predictive performance can be significanly improved using current state of the art techniques (such as state selection methods used by Bunton, 1996). There are also much faster alternative algorithms for performing a Viterbi-based search, which may or may not impede on the accuracy of the output.

- The implementation does not currently handle hierarchical markup models as described by Witten *et al.* (1999). A number of different ways of handling this are currently being investigated.

- Certain text modelling problems still do not elegantly fall within the reach of the extended API. For example, building a part of speech tagger using the API based on certain classes of model would be difficult (such as models that base the prior and future context on both words *and* part of speech tags). These modelling problems need to be investigated further, and the API modified or further extended as needed.

7 Acknowledgments

I would like to thank Jan Åberg for the many discussions and suggestions about this work.

8 References

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

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.

Cleary, J.G. & Teahan, W.J. 1999. "An open interface for probabilistic models of text" in *Proceedings DCC'99*, edited by Storer, J.A. & Cohn, M., IEEE Computer Society Press.

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

Larsson, N.J. 1999. *Structures of string matching and data compression.* Ph.D. thesis, Lund University, Sweden.

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.

Shannon, C.E. 1948. "A mathematical theory of communication," *Bell System Technical Journal*, **27**, 379–423, 623–656.

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

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.

Witten, I.H., Bray, Z., Mahoui, M. & Teahan, 1998. "Text mining: A new frontier for lossless compression" in *Proceedings DCC'99*, 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.

9 Appendix A—Language model routines

This appendix describes all of the language model routines that can be used for modelling sequential text—these correspond to the routines in the original API description. All these routines may be identified by their common prefix `TLM_`.

```
#define NIL 0
#define TLM_static 0
#define TLM_dynamic 1

typedef unsigned char boolean;

unsigned int TLM_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 TLM_create_coder (unsigned int max_frequency,
             void (*arithmetic_encode) (unsigned int, unsigned int, unsigned int),
             void (*arithmetic_decode) (unsigned int, unsigned int, unsigned int),
             unsigned int (*arithmetic_decode_target) (unsigned int));
```

Creates and returns an unsigned integer which provides a reference to a coder record associated with an arithmetic coder. `max_frequency` specifies the maximum frequency allowed for the coder. The arguments `arithmetic_encode`, `arithmetic_decode` and a `arithmetic_decode_target` are pointers to the necessary routines required for encoding and decoding. Both `arithmetic_encode` and `arithmetic_decode` take three unsigned integers as arguments that specify the current arithmetic coding range (low, high and total); `arithmetic_decode_target` takes just a single unsigned integer as an argument, which is set to the total of the current coding range.

```
unsigned int TLM_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.

```
unsigned int TLM_create_model (unsigned int alphabet_size, char *title, ...);
```

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`). An `alphabet_size` of 0 specifies that the alphabet is unbounded. (This is useful for word-based alphabets, for example). In this case, allowable symbol numbers range from 0 up to a special "`expand_alphabet`" symbol which is equal to the current maximum symbol number (this is one more than the highest previously seen symbol number). If the current symbol becomes the `expand_alphabet` symbol, then the current

maximum symbol number is incremented by 1, thus effectively expanding the size of the alphabet by 1. The current maximum symbol number may be obtained by calling the routine `TLM_get_model`.

The argument `title` is intended to be a short human readable text description of the origins and content of the model. This argument is followed by a variable number of parameters used to hold model information which differs between implementations. For example, this implementation uses it to specify the maximum order of the PPM model, and whether the model should perform update exclusions.

```
unsigned int TLM_decode_symbol (unsigned int context, unsigned int coder)
```

Returns the symbol decoded using the arithmetic coder. Updates the context record so that the last symbol in the context becomes the decoded symbol.

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

Prints a human readable version of the model (intended mainly for debugging). The argument `dump_symbol_function` is a pointer to a function for printing symbols. If this is `NULL`, then each symbol will be printed as an unsigned int surrounded by angle brackets (e.g. `<123>`), unless it is human readable ASCII, in which case it will be printed as a char.

```
void TLM_dump_models (FILE *fp,
    void (*dump_symbol_function) (FILE *, unsigned int));
```

Writes a human readable version of all the currently valid models to the file. The argument `dump_symbol_function` is a pointer to a function for printing symbols. If this is `NULL`, then each symbol will be printed as an unsigned int surrounded by angle brackets (e.g. `<123>`), unless it is human readable ASCII, in which case it will be printed as a char.

```
void TLM_encode_symbol (unsigned int context, unsigned int coder,
    unsigned int symbol);
```

Encodes the specified symbol using the arithmetic coder. Updates the context record so that the last symbol in the context becomes the encoded symbol.

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

Finds the predicted symbol in the context. The argument codelength is set to the cost in bits of encoding the specified symbol given the current context. The context record is not updated.

```
unsigned int TLM_getcontext_modelno (unsigned int context);
```

Returns the model number associated with the context.

```
unsigned int TLM_getcontext_position (unsigned int context);
```

Returns an unsigned integer which uniquely identifies 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 TLM_get_model (unsigned int model, unsigned int *model_type,
        unsigned int *alphabet_size, unsigned int *max_symbol, char **title, ...);
```

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 `TLM_create_model`. `model_type` is set to `TLM_dynamic` or `TLM_static` depending on whether the model is static or dynamic. `max_symbol` is set to the current maximum symbol number. `title` is followed by a variable number of parameters used to hold model information which differs between implementations. For example, this implementation uses it to specify the maximum order of the PPM model, and whether the model should perform update exclusions.

```
unsigned int TLM_load_model (FILE *fp, char *new_title);
```

Loads a model which has been previously saved to the file into memory and allocates it a new model number which is returned. If `new_title` is non-null, then the model is assigned the new title, replacing whatever the old title was.

```
unsigned int TLM_minlength_model (unsigned int model);
```

Returns the minimum number of bits needed to write the model out to disk as a static model and recover it later. This is useful for computing minimum description lengths of messages.

```
unsigned int TLM_next_modelno (void);
```

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

```
boolean TLM_next_symbol (unsigned int context, unsigned int *symbol,
        float *codelength);
```

Returns the next predicted symbol in the context and the cost in bits of encoding it. The context record is not updated. If a sequence of calls to `TLM_next_symbol` 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 function returns `FALSE` when there are no more symbols to process. `TLM_reset_symbol` will reset the current position to point back at the first predicted symbol of the current context.

The codelength value is the same as that returned by the routines `TLM_find_symbol` and `TLM_update_codelength` which may use faster search methods to find the symbol's codelength more directly (rather than sequentially as `TLM_next_symbol` does). A call to the `TLM_find_symbol` routine or other routines will have no affect on subsequent calls to the routine `TLM_next_symbol`.

```
unsigned int TLM_numberof_models ();
```

Returns the number of currently valid models.

```
void TLM_release_coder (unsigned int coder);
```

Releases the memory allocated to the coder and the coder number (which may be reused in later `TLM_create_coder` calls).

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

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

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

Releases the memory allocated to the model and the model number (which may be reused in later `TLM_create_model` or `TLM_load_model` calls). A run-time error will be generated if an attempt is made to release a model that still has active contexts pointing at it.

```
void TLM_release_models ();
```

Releases the memory used by all the models.

```
void TLM_reset_modelno ();
```

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

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

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

```
unsigned int TLM_sentinel_symbol (unsigned int model);
```

Returns an unsigned integer that uniquely identifies a special "sentinel" symbol. The sentinel symbol is used where there is a break required in the updating of the context, such as when the end of string has been reached or when more than one model is being used to encode different parts of a string. The effect of encoding the sentinel symbol is that the prior context is forced to the null string i.e. the subsequent context will contain just the sentinel symbol itself. This is useful during training if there are statistics that differ markedly at the start of some text than in the middle of it (for example, individual names, and titles within a long list).

This routine is usually used with the update and search routines (see section **??**). For example, the following source code will the context record so that the current symbol becomes the sentinel symbol: `TLM_update_symbol (context, TLM_sentinel_symbol ())`.

```
unsigned int TLM_sizeof_model (unsigned int model);
```

Returns the current number of bits needed to store the model in memory.

```
void TLM_update_codelength (unsigned int context, unsigned int symbol,
     float *codelength);
```

Updates the context record so that the current symbol becomes symbol, returning at the same time the code length of the specified symbol in bits (i.e. the cost of encoding it given the current context).

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

Updates the context record so that the current symbol becomes symbol.

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

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

10 Appendix B—Text routines

This appendix describes all of the text routines that can be used for manipulating sequential text (i.e. sequences of symbols). All these routines may be identified by their common prefix `TXT_`.

```
int TXT_compare_text (unsigned int text1, unsigned int text2);
```

Compares the text records `text1` with `text2`. Returns zero if they are the same, negative if `text1` $<$ `text2`, and positive if `text1` $>$ `text2`. The sentinel symbol returned by the routine `TLM_sentinel_symbol` is regarded as having a value lower than all other symbols.

```
unsigned int TXT_copy_text (unsigned int text);
```

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

```
unsigned int TXT_create_text (void);
```

Creates and returns an unsigned integer which provides a reference to a text record.

```
void TXT_dump_symbol (FILE * fp, unsigned int symbol);
```

Writes the ASCII symbol out in human readable form (excluding `'\n'` and `'\t'` characters).

```
void TXT_dump_text (FILE *fp, unsigned int text,
     void (*dump_symbol_function) (FILE *, unsigned int));
```

Dumps out a human readable version of the text record to the specified file. The argument `dump_symbol_function` is a pointer to a function for printing symbols. If this is NULL, then each symbol will be printed as an unsigned int surrounded by angle brackets (e.g. `<123>`), unless it is human readable ASCII, in which case it will be printed as a char.

```
void TXT_extract_text (unsigned int text, unsigned int subtext,
     unsigned int subtext_pos, unsigned int subtext_len);
```

Extracts the text from out of the text record. The argument `subtext` is set to the extracted text; `subtext_len` is the length of the text to be extracted; and `subtext_pos` is the position from which the text should be extracted from. The extracted subtext is filled with nulls for any part of it that extends beyond the bounds of the text record.

```
boolean TXT_getpos_text (unsigned int text, unsigned int symbol,
    unsigned int *pos)
```

Returns TRUE if the symbol is found in the text. The argument pos is set to the position of the first symbol in the text that matches the specified symbol if found, otherwise it remains unchanged.

```
boolean TXT_getrpos_text (unsigned int text, unsigned int symbol,
    unsigned int *pos)
```

Returns TRUE if the symbol is found in the text. The argument `pos` is set to the position of the last symbol in the text that matches the specified symbol if found, otherwise it remains unchanged.

```
boolean TXT_getsymbol_text (unsigned int text, unsigned int pos, *symbol);
```

Returns TRUE if there exists a symbol at position exists in the text. The argument `symbol` us set to the specified symbol.

```
int TXT_getline_text (FILE *fp, unsigned int text);
```

Reads in a line of text from the specified file. Returns the last character read or EOF.

```
void TXT_insert_text (unsigned int text, unsigned int symbol);
```

Inserts symbol into the text.

```
boolean TXT_is_alpha (unsigned int symbol);
```

Returns TRUE if symbol is an alphabetic character.

```
boolean TXT_is_alphanumeric (unsigned int symbol);
```

Returns TRUE if symbol is an alphanumeric character.

```
boolean TXT_is_consonant (unsigned int symbol);
```

Returns TRUE if symbol is a consonant.

```
boolean TXT_is_control (unsigned int symbol);
```

Returns TRUE if symbol is a control character.

`boolean TXT_is_digit (unsigned int symbol);`

Returns TRUE if symbol is a digit.

`boolean TXT_is_graph (unsigned int symbol);`

Returns TRUE if symbol is a printable character except space.

`boolean TXT_is_lower (unsigned int symbol);`

Returns TRUE if symbol is a uppercace character.

`boolean TXT_is_print (unsigned int symbol);`

Returns TRUE if symbol is a printable character.

`boolean TXT_is_punct (unsigned int symbol);`

Returns TRUE if symbol is a punctuation character.

`boolean TXT_is_space (unsigned int symbol);`

Returns TRUE if symbol is a white space character.

`boolean TXT_is_upper (unsigned int symbol);`

Returns TRUE if symbol is a uppercace character.

`boolean TXT_is_vowel (unsigned int symbol);`

Returns TRUE if symbol is a vowel.

`unsigned int TXT_length_text (unsigned int text);`

Returns the number of symbols in the text record.

`unsigned int TXT_load_text (FILE *fp);`

Creates a new text record and text number, then loads it using text from the specified file.

`void TXT_putsymbol_text (unsigned int text, unsigned int symbol, unsigned int pos)`

Inserts the symbol at position pos into the text. Inserting a symbol beyond the current bounds of the text will cause a run-time error.

`void TXT_release_text (unsigned int text);`

Releases the memory allocated to the text record and the text number (which may be reused in later `TXT_create_text` calls).

`unsigned int TXT_setlength_text (unsigned int text, unsigned int length);`

Sets the length of the text record to be at most length symbols long. If the current length of the text is longer than this, then the text will be truncated to the required length, otherwise the length will remain unchanged. Setting the length of the text to be 0 will set the text to the null string.

`void TXT_sprintf_text (unsigned int text, char *format, ...);`

Sets the text record to the symbols specified by the format and variable length argument list.

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the text record; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- `%` The % (percentage) character is inserted into the text.
- `s` The argument list contains a symbol number (unsigned int) which will be inserted into the text record at the location specified by the format specification.

11 Appendix C—Markup model routines

This appendix describes all of the markup model routines that can be used for correcting or "marking up" sequential text. All these routines may be identified by their common prefix `TMM_`.

```
void TMM_add_markup (unsigned int markup_model, float codelength,
     char *observed_text_format, char *markup_text_format, ...);
```

Adds the formatted correction to the markup model. The argument `observed_text_format` is the format of the observed text begin corrected; `markup_text_format` is the format of the text it will be corrected to; and codelength is the cost in bits of making that correction when the text is being corrected. These arguments are followed by a variable length list of arguments as specified by the observed text and markup text formats.

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the markup text (i.e the corrected text); and conversion specifications which follow the character %, each of which results in fetching zero or more subsequent arguments.

The conversion specifications for the observed text format are the following:

- `%` This will match the % (percentage) character.
- `m` This will match when the predicting model has the same model number as the corresponding one specified in the argument list.
- `s` This will match the corresponding symbol in the argument list.
- `f` Function symbols: the argument list contains a pointer to a boolean function that takes an unsigned int symbol number as its only argument and returns a non-zero value if the current context symbol number is a valid match. The format of the boolean function is the following: `boolean (*function) (unsigned int symbol)` (example: `TXT_is_alphannumeric (symbol)`).
- `w` Wildcard symbol: this will match the current symbol in the context.
- `[..]` Range symbols: this will match any symbol specified between the square brackets. Example: `%[aeiou]` matches vowels.

The conversion specifications for the markup text format are the following:

- `%` The % (percentage) character is inserted into the markup text.
- `m` The argument list contains a model number which will be used to predict subsequent symbols in the markup text.
- `s` The argument list contains a symbol number (unsigned int) which will be inserted into the markup text.
- `f` Function symbol: this will insert the matching function symbol from the context into the markup text.
- `w` Wildcard symbol: this will insert the matching symbol from the context into the markup text.
- `r` Range symbol: this will insert the matching range symbol from the context into the markup text.
- `[..]` Range symbols: this will generate markup texts for all the symbols specified between the square brackets.

Examples:

```
TMM_add_markup (markup_model, codelength, "1", "a");
```

This generates a single markup that replaces the character "1" with the letter "a".

```
TMM_add_markup (markup_model, codelength, "%w", "%w ");
```

This generates a single markup that inserts an extra space after each symbol.

```
TMM_add_markup (markup_model, codelength, "%f", "%f ", TXT_is_alphabetic);
```

This generates a single markup that inserts an extra space after each alphabetic symbol.

```
TMM_add_markup (markup_model, codelength, "%[xy]", "%r%[abc]");
```

This generates the following markup corrections: "x" is corrected to "xa", "xb" and "xc"; "y" is corrected to "ya", "yb" and "yc".

```
unsigned int TMM_create_markup (void);
```

Creates and returns an unsigned integer which provides a reference to a record associated with a markup model used to correct or "mark up" text. The argument `markup_algorithm` specifies the type of markup algorithm to use such as `TMM_Viterbi`, and is followed by a varying number of arguments as determined by the `markup_algorithm`.

```
void TMM_dump_markup (FILE *fp, unsigned int markup_model);
```

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

```
unsigned int TMM_perform_markup (unsigned int markup_model,
    unsigned int language_model, unsigned int source_text);
```

Creates and returns an unsigned integer which provides a reference to a text record that contains the observed text corrected according to the markup and language models.

```
void TMM_release_markup (unsigned int markup_model);
```

Releases the memory allocated to the markup record and the markup number (which may be reused in later `TMM_create_markup` calls).

```
void TMM_start_markup (unsigned int markup_model, unsigned int language_model);
```

Starts a new process for correcting or "marking up" text using the specified language model. This routine must be called at least once before `TMM_perform_markup` is called. Multiple calls to this procedure will cause separate searches to be initiated for each of the specified language models.