

# A Compression-Based Toolkit for Modelling and Processing Natural Language Text

Teahan, William

**Information**

*DOI:*
10.3390/info9120294

Published: 22/11/2018

Publisher's PDF, also known as Version of record

information

MDPI

*Article*

# A Compression-Based Toolkit for Modelling and Processing Natural Language Text

**William John Teahan**

School of Computer Science and Electronic Engineering, Bangor University, Dean Street , Bangor, Gwynedd LL57 1UT, UK; w.j.teahan@bangor.ac.uk; Tel.: +44-1248-282703

Received: 30 July 2018; Accepted: 25 September 2018; Published: 22 November 2018



**Abstract:** A novel compression-based toolkit for modelling and processing natural language text is described. The design of the toolkit adopts an encoding perspective—applications are considered to be problems in searching for the best encoding of different transformations of the source text into the target text. This paper describes a two phase 'noiseless channel model' architecture that underpins the toolkit which models the text processing as a lossless communication down a noise-free channel. The transformation and encoding that is performed in the first phase must be both lossless and reversible. The role of the verification and decoding second phase is to verify the correctness of the communication of the target text that is produced by the application. This paper argues that this encoding approach has several advantages over the decoding approach of the standard noisy channel model. The concepts abstracted by the toolkit's design are explained together with details of the library calls. The pseudo-code for a number of algorithms is also described for the applications that the toolkit implements including encoding, decoding, classification, training (model building), parallel sentence alignment, word segmentation and language segmentation. Some experimental results, implementation details, memory usage and execution speeds are also discussed for these applications.

**Keywords:** text compression; text processing; encoding; decoding

## 1. Introduction

This paper describes the design and implementation of a compression-based toolkit for text processing. The toolkit can be applied to any type of text (for example, source code, terminal session transcripts, and bibliographic references) since it involves the application of text compression algorithms designed for general text. However, this paper will focus specifically on its application to natural language processing (NLP) and the processing of written natural language text.

The toolkit is based on an earlier Application Programming Interface (API) designed by Cleary & Teahan [1] for modelling sequential text using text compression models. The purpose of the toolkit is to simplify the design of applications where textual models are needed such as text compression, categorisation, and segmentation, by shielding the user from the details of the modelling and probabilistic estimation process. Another purpose is to enable different implementations of models to be replaced transparently in application programs. The design behind the toolkit is probabilistic: that is, it supplies the probability of the next symbol in the sequence. It is general enough to deal accurately with Markov-based models that include escapes for probabilities such as the Prediction by Partial Mapping (PPM) text compression scheme [2,3].

During the development of the API and earlier versions of the toolkit, it became apparent that broadening its scope to encompass a much wider range of text operations was possible. It was found that many of the API-based programs that were devised had remarkable similarities in the underlying source code—despite the programs being complex, only a few changes were needed to define the key differences between the applications [4]. These observations have led to the creation of the Tawa Toolkit

described in this paper. (Tawa is a common broadleaf tree found in New Zealand. The acronym stands for **T**ext **A**nalyser from **Wa**ikato since it was first developed at Waikato University, and subsequently improved over two decades at Lund, Robert Gordon and Bangor Universities.) The toolkit is now available for download (https://bangoroffice365-my.sharepoint.com/:f:/g/personal/eesa0e_bangor_ac_uk/Ej-84lAJuUNOuSVz44daZCIBt__4NQyanF_5qH8FLSJaIQ?e=rFNW3P) and is a comprehensive extension of the original Cleary & Teahan API and an updated version of the TMT toolkit [4] that was based on the API. The toolkit provides a powerful method for classifying, mining and transforming text. The purpose of this paper is to describe the design and implementation behind the toolkit including its architecture, libraries, and sample applications.

The contributions of this paper are as follows:

- The compression-based architecture which underpins the toolkit, and how it differs from the standard architecture for NLP, the noisy channel model, is fully described for the first time. This paper explains the encoding approach taken by the toolkit's architecture as opposed to the decoding approach of the noisy channel model and how it provides a wider range of processing possibilities.
- This paper is the first to discuss in detail the toolkit's libraries, including the methods and objects used. The toolkit provides an extensive update of previous versions including an overhaul of the transformation libraries that simplifies and consolidates the processes previously used.
- The algorithms behind each of the toolkit's main applications are also described for the first time.

This paper is organised as follows. In the next section, we detail the design of the toolkit and its underlying architecture. The toolkit's libraries are detailed in Section 3 and sample methods discussed in Section 3.1. The pseudo-code for nine applications built using the libraries that demonstrate how they work are detailed in Section 4. This section also discusses sample use, implementation details, memory usage, execution speeds and search algorithms used. These applications provide useful tools for many NLP operations that have often produced state-of-the-art experimental results as detailed in previous publications listed in Section 5. The paper then discusses related work and provides a conclusion and discussion of future work in the final section.

## 2. The Toolkit's Design

In this section, the overall design of the Tawa toolkit shown diagrammatically in Figure 1 is described. The aim of the toolkit is to simplify the conceptualisation and implementation for a broad range of text mining and NLP applications involving textual transformations. The toolkit presently consists of eight main applications (**align**, **classify**, **codelength**, **decode**, **encode**, **markup**, **segment** and **train**). These applications are built using three libraries—`TXT` for text-based operations, `TLM` for language modelling operations, and `TTM` for transformation operations.

Underlying the design of these libraries and applications is a novel compression-based architecture called the "Noiseless Channel Model" to distinguish it from the standard "Noisy Channel Model". The key insight is that each application performs a search to find the best encoding of the target message rather than performing a decoding of the observed message. The noisy channel model is a common architecture for many NLP applications, such as OCR, spelling correction, parts-of-speech (POS) tagging and machine translation [5]. In this architecture, applications are formulated as a communication process with the source text conceptualised as being sent down a communication channel by the sender, with the receiver trying to recover the original message in the presence of noise by correcting the observed text that is received. This process of text correction involves the search for the most probable corrected target text which is often referred to as decoding (see Figure 2a). Note that the correct source text is unknown—the search process for the target text makes a best guess as to what it might be.

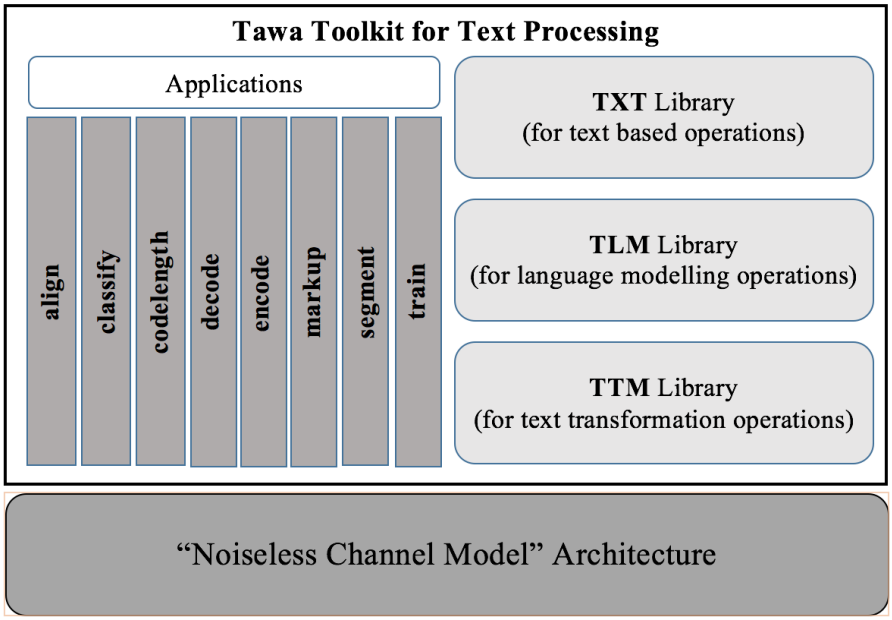


**Figure 1.** A diagrammatic overview of the Tawa Toolkit and its design.

The noisy channel model leads to robust and effective performance for many NLP tasks despite it being perhaps a rather arbitrary and abstract characterisation for NLP. As an alternative, we wish to adopt language models based on well performing text compression techniques such as PPM which have already been found to be highly effective in many NLP applications [4,6–11]. Therefore, we have also investigated in previous research [4] an alternative design perspective with an emphasis based on encoding rather than decoding. This has culminated in the architectural design shown in Figure 2b. This design adheres to two main principles—the communication process must be both *lossless* and *reversible*. A lossless encoding by definition is reversible with no loss of information, with no noise being added to the message being transmitted. Note that any transformation that occurs prior to the encoding in our approach also needs to be reversible. This is so that there is no loss in information during the communication in order to ensure the message is transmitted correctly and as efficiently as possible and this can be verified during decoding.

The Tawa toolkit based on this architecture has now gone through several design cycles culminating in the latest version of the toolkit described in this paper. In this updated toolkit, an NLP process is not thought of as being one of noisy communication, therefore requiring a decoding process to recover the original message. Instead, the approach taken by the toolkit is to characterise an NLP process as a noiseless or noise-free communication. In this setting, the sender will search for the best encoding of the target message and this is then transmitted losslessly down the communication channel. In this case, the decoding process is of secondary importance, mainly to verify the encoding process—the emphasis instead is placed on how efficiently the target message can be encoded. Also encoded is additional information on how to transform the source message into the target message. In this case, the search process—that of finding the best transformation, plus the information required to transform between them—occurs prior to the encoding phase rather than during the decoding phase. (See Figure 2b).

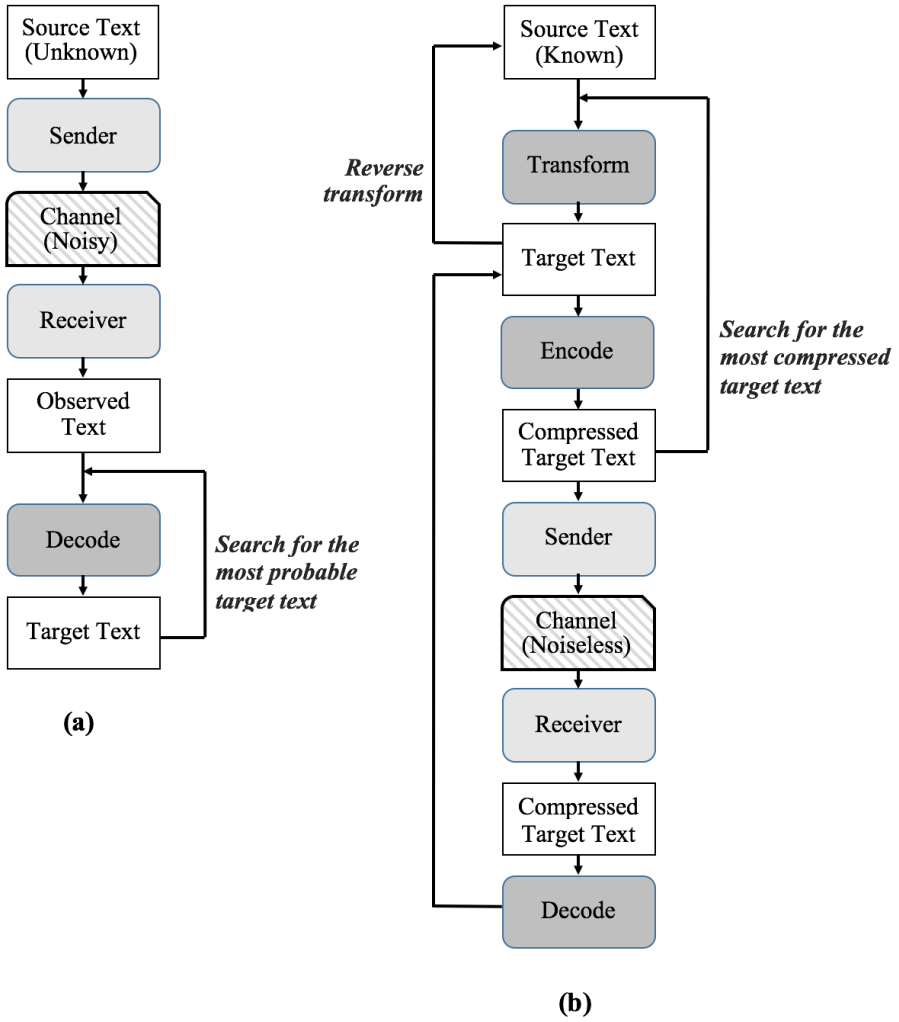


**Figure 2.** Two architectures for NLP: (**a**) the standard decoding-based 'Noisy Channel Model' versus (**b**) the encoding-based 'Noiseless Channel Model' adopted by the Tawa Toolkit.

Following the workflow in the figure, the sender first starts from a known source text and transforms it in some way (as shown by the dark gray box labelled 'Transform') into the target text. This is the text that is the desired output for some application (such as compression, text mining, spelling correction, machine translation and so on). In order to illustrate the variety of possible transformations which includes but goes beyond those traditionally covered by the noisy channel model, some example transformations are listed below. (The first two examples below are transformations possible using a related tag insertion approach [12] that was similar to what was implemented in an earlier version of the toolkit. However, the subsequent examples above go beyond the tag insertion approach and are used to illustrate what is possible in the updated toolkit):

- inserting markup tags into the text (called tag insertion), such as to indicate where different languages are in the text for language segmentation [7,13];
- inserting extra characters, such as spaces for word segmentation [9];
- changing the alphabet, such as replacing common character bigraphs or word bigrams with new symbols to aid compression [6,14];
- transforming the text into a grammar, such as for grammar-based compression [15];
- encoding extra information such as POS tags for POS tagging and tag-based compression [6,16];
- switching between models used to encode each symbol in the text [7,17].

The transformations are done in such a way that they are *reversible* (as shown by the line labelled 'Reverse transform' in the figure). If tags or spaces are added, for example, then these can simply be removed to recover the original source text without any loss of information. If the alphabet is altered, then there is a one-to-one mapping between the old and new alphabets so that the original source text is easily recoverable. This property of reversibility ensures that information is communicated losslessly.

The target text is then encoded to produce the compressed target text (as depicted by the dark gray box labelled 'Encode'). A search process shown in the figure by the rightmost arrow is used to find the most compressed text by repeatedly trying out different variations of the transformation (for example, by inserting or not inserting spaces after each character if performing word segmentation). This search can be thought of as the transformation & encoding phase whose purpose is to generate a suitable target text. This search may also require extra transformations in order to ensure that the encoding is fully reversible (for example, reversible word segmentation requires replacing any existing spaces with another unique character so that the original text can be unambiguously recovered).

The remaining parts of the workflow shown in the figure is the verification & decoding phase. This step is not necessarily needed depending on the application since just producing the target text will usually be sufficient in order to complete the task(s) required. However, as argued below, verification is of fundamental importance in ensuring the output is correct. This is done by sending the compressed target text down a noise-free communication channel, and then checking that the decoded target text matches the original target text.

Mahoui et al. [4] pointed out that the noiseless channel model could be viewed simply as "the noisy channel model in another guise". This is because the search processes (e.g., the Viterbi algorithm [18]) produce a similar result with the best target text being found in both cases. However, as Mahoui et al. have argued, there are important differences. Firstly, when designing text compression systems, it is well-known that the encoding and decoding processes are not necessarily equivalent (for example, often either the encoding or decoding process can take noticeably longer as further effort is required to maintain probability tables in the form required for coding).

Secondly, the perspective has changed from correcting 'noise' (which is just one approach to NLP) to one of efficiently encoding the source text plus further information to transform the message into the target text (which can be considered an equally plausible alternative approach to NLP).

Thirdly, as stated, the decoding process for the noiseless channel model provides a means for physically *verifying* the correctness of the communication. That is, we can directly apply standard techniques used for lossless text compression by checking that we can directly encode and decode the target message to and from a file on disk. (That is, there is a physical measurable object that exists in the real world). The noiseless channel model makes it clear that all information must be transmitted. Many current approaches using the noisy channel model base the decoding on implicit information and make the assumption that this information has little or no effect on the accuracy of the transformation. This may be correct, or erroneous, but this first needs to be verified, which is one of the main arguments for supporting the alternative approach in Figure 2b. All necessary information must be encoded explicitly otherwise it would not be possible to get the decoder to work. As a consequence, the encoding codelengths (how much it takes in bits to physically encode the information) becomes a useful and verifiable means for measuring the effectiveness of the language models being used.

And fourthly, the changed perspective opens up further possibilities for improved language models based on models found effective for lossless text compression, such as: alternative encoding methods, not necessarily probabilistic, e.g., the Burrows Wheeler Transform [19] and Lempel-Ziv methods [20,21]; methods for preprocessing the text prior to encoding [22,23]; and hybrid methods for combining or switching between models [17,24].

It is also clear from Figure 2 that the source text for the noiseless channel model is known when it is encoded which contrasts with the noisy channel model where the original source text is unknown. This is a fundamental difference that must be considered when designing applications based on these two architectures. In the noiseless case, the designer has access to all the information from the

source text (since it is known) during the search which may be significantly more efficient as a result. For example, one method found effective at improving compression of natural language text is to replace common bigraphs with new single unique symbols prior to compression [15,23]. This is a very straightforward task for an encoder working from a known source to find the most common bigraphs. However, bigraph replacement at the decoding stage is not possible in the same way since the original source text is not known.

## 3. The Toolkit's Libraries

In this section, the libraries provided with the toolkit are described. As with the original Cleary and Teahan API [1] whose design the libraries are based upon, the intention of the libraries is to shield the user from details of the modelling and probability estimation process. A primary goal is to allow the rapid prototyping of applications where the use of textual models is needed, such as text compression, text classification, text mining, and text correction, with a minimal amount of code needed for implementation.

Table 1 lists the objects the libraries use and their description. When compared to standard text operations that most programming languages provide, the *Symbol* and *Text* objects have a similar functionality to characters and strings, and the *File* object is closely related to its programming language equivalent. However, the *Symbol* object provides a much greater range of symbol numbers for the developer to use rather than just the 256 possibilities that 8-bit ASCII characters provide (for example, the current toolkit implementation supports symbol numbers up to a maximum just below $2^{31}$ for PPM models). Although the *Text* object provides similar operations that strings do in standard programming languages, the length of the text is not restricted and can be varied at run-time by some operations such as appending onto the end of the text.

The toolkit defines various further complex objects such as *Model* and *Coder* that facilitate the design of context-based modelling and compression-based applications and are used for specifying the 'model' and 'coder' that are at the heart of these types of compression applications (and are to be treated as 'black box' by developers in order to avoid the very complex details that are associated with their implementation). Rissanen and Langdon [25] have noted the important distinction between *modelling* and *coding*, and the toolkit uses this insight in its design. Uniquely, the toolkit also explicitly incorporates a *Context* object (this is similar to that used in pseudo-code for an adaptive statistic encoder and decoder proposed by Teahan [6] (p. 112)) as well in its design to aid in the traversal of the model and the probability estimation process as the text is processed sequentially.

The final *Transform* is an object whose purpose is to ease the design of applications requiring some form of transformation on the text such as used for correction-based or 'tag insertion' style algorithms (i.e., insertion of markup tags). The type of search algorithm, such as the Viterbi algorithm [18] or the stack decoding algorithm [26], is specified when the *Transform* object is created. Specific transformations of the form $source\ text \rightarrow target\ text$ are then added to the object. These specify the source text sequence that must be matched before the target sequence is generated. Multiple transformations will generate multiple paths in the search tree. A similar convention used for regular expressions is used here to represent (and match) the source and target text. For example: a wildcard form can be used to match any character; ranges denoted by [...] denote character disjunction; and functions can also be used to match character sequences. A method is used to start the transformation process using a specific *Model* object to define what model to use for the encoding of the corrected target sequence and this can be called multiple times for different *Model* objects. Another method is then called to perform the transformation process which returns a transformed *Text* object upon completion which represents the most compressed transformed target sequence found by the search.

**Table 1.** A summary of the objects used by the toolkit's libraries.

| Object | Description |
|---|---|
| *Symbol* | A symbol is a number that is being modelled. It is represented in C as an unsigned int. There is no specific notion of what a symbol means. However, it is restricted by the alphabet size ($A$) when a model is created. e.g., $A = 256$ is used for ASCII text; $A = 0$ means an unbounded alphabet used for words. The mapping between symbols and text characters is user-defined such as ASCII byte values or a different mapping (for example, required for UTF-8 coded text). |
| *File* | A file is associated with a physical file on disk. It is used to open, read and load text files. |
| *Text* | A text is used for storing a sequence of symbols. They are essentially variable length arrays of symbols. |
| *Model* | A model provides predictions for sequences of symbols. It has an alphabet size ($A$) and order ($O$) associated with it. A model can also be static or dynamic. They are used for specifying the type of language model being used and the type of operations being performed (such as a PPM [2,3] language model). |
| *Coder* | A coder is used for encoding and decoding a sequence of symbols from compressed files on disk using an arithmetic coder, for example, or a range coder. |
| *Context* | A context is used for accessing and updating a model. The coding context is updated from the symbols that have just been encoded. (For a PPM model, the maximum length of the context is fixed as specified when the model is created). |
| *Transform* | A 'transform' defines a possible transformation for altering or correcting the sequence. These are used for defining and performing transformations on the text such as used for correction-based or 'tag insertion' style algorithms (i.e., insertion of markup tags). |

### *3.1. A Selection of Methods Provided by the Toolkit's Libraries*

Table 2 provides a selection of the main methods provided by the libraries for each of the seven types of objects described in the previous section. Space precludes listing all the methods provided by the toolkit which can be found by downloading the software distribution. The table lists methods for each object type in alphabetical order. The C function name used in the libraries and its arguments are listed in column three of the table. Note that the prefixes `TLM`, `TXT` and `TTM` used in the function names are used to signify the library being used for language modelling operations, text-based operations and transformation operations respectively. Column one lists the main type of object being operated upon, and column two lists the type of object returned by the method. (Here *Void* indicates the method returns nothing, $Unsign.\,Int.$ indicates the method returns an unsigned integer, $Transform$ a transform object, *Boolean* a Boolean and *Char* a character with the other return types being one of the toolkit's object types).

A brief description of these methods for each of the toolkit's object types now follows:

- Methods for *Symbol* objects: `TLM_decode_symbol` (*Model*, *Context*, *Coder*) returns the symbol decoded using the arithmetic *Coder* and also updates the *Context* for the *Model* so that the last symbol in the context becomes the decoded symbol. `TLM_encode_symbol` (*Model*, *Context*, *Symbol*, *Coder*) encodes the specified *Symbol* using the arithmetic *Coder* and also updates the *Context* for the *Model* so that the last symbol in the context becomes the encoded symbol. `TXT_sentinel_symbol()` returns the "sentinel symbol". This is used where there is a break required in the updating of the context, such as when the end of text has been reached or when more than one model is being used to encode different parts of a text. 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).
- *File* methods: `TXT_close_file` (*File*) closes the *File* and releases memory used by the file object. `TXT_open_file` (*Filename*, *Mode*, ...) opens the file named *Filename* with the specified

*Mode* and creates and returns a file object for the file. `TXT_readline_file` (*File*, *Text*) loads the *Text* using the next line of input from the *File*. It returns the last character read or *EOF*. `TXT_write_file` (*File*, *String*) writes out the *String* to the *File*.

- *Text* methods: `TXT_create_text()` creates and returns a text object for storing, accessing and updating a string of symbols. `TXT_is_alphanumeric` (*Symbol*) returns *True* if the *Symbol* is an alphanumeric character. `TXT_load_text` (*File*) creates and returns a new text object, loading it using text from the *File* (assuming standard ASCII text. `TXT_length_text` (*Text*) returns the length of the *Text*. `TXT_release_text` (*Text*) frees the memory used by the *Text*.
- *Model* methods: `TLM_create_model` (*ModelType*, *Title*, ...) creates and returns a new empty dynamic model of the specified type *ModelType*, *Title* (such as `TLM_PPM_Model` for a PPM model) and other parameters depending on the *ModelType*. `TLM_load_model` (*File*) loads a model which has been previously saved to the *File* into memory and allocates it a new model object which is returned. `TLM_next_model()` returns a model object for the next valid model in the list of models that have been loaded into memory (or *Nil* if there are none). `TLM_release_model` (*Model*) frees the memory used by the *Model*. `TLM_reset_model()` resets the current model so that the next call to the method `TLM_next_model` will return the first valid model in the list of models that have been loaded into memory (or *Nil* if there are none). `TLM_write_model` (*File*, *Model*, *ModelForm*) writes out the *Model* to the *File* (which can then be loaded by other applications later). The argument *ModelForm* must have the value `TLM_Static` or `TLM_Dynamic` and determines whether the model is static or dynamic when it is later reloaded using the method `TLM_load_model`.
- *Coder* methods: `TLM_create_arithmetic_decoder` (*SourceFile*, *CompressedFile*) creates an arithmetic coder object used for decoding from the compressed file *CompressedFile* back to the source file *SourceFile*. The method `TLM_create_arithmetic_encoder` (*SourceFile*, *CompressedFile*) creates an arithmetic coder object used for encoding from the source file *SourceFile* to the compressed file *CompressedFile*. `TLM_release_coder` (*Coder*) frees the memory used by the *Coder*.
- *Context* methods: `TLM_create_context` (*Model*) creates a context object for the *Model* used during operations such as encoding, decoding and update on which predictions of the upcoming symbol are made. `TLM_update_context` (*Model*, *Context*, *Symbol*) updates the *Context* for the *Model* using the current *Symbol*. `TLM_release_context` (*Model*, *Context*) frees the memory used by the *Context* for *Model*.
- *Transform* methods: `TTM_add_transform` (*Transform*, *Codelength*, *SourceFormat*− *String*, *TransformFormatString*, ...) adds the formatted correction to the *Transform* object. The argument *SourceFormatString* is the format of the original text begin corrected; *TransformFormatString* 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 or marked up. `TTM_create_transform` (*TransformAlgorithm*, ...) creates and returns a transform object which can be used to transform or "mark up" text. The argument *TransformAlgorithm* specifies the type of search algorithm to use such as `TTM_Viterbi`. `TTM_perform_transform` (*Transform*, *SourceText*) creates and returns a text object that contains the result of the search for the most compressible *SourceText* when corrected according to the transform object and models associated with it. `TTM_release_transform` (*Transform*) frees the memory used by the *Transform*. `TTM_start_transform` (*Transform*, *Type*, *SourceText*, *Model*) starts a new search process for correcting or marking up the *SourceText* using the specified *Model* by applying the transforms specified by the *Transform*. This routine must be called at least once before `TTM_perform_transform`. Multiple calls to this procedure will cause separate transform paths to be initiated for each of the specified models.

**Table 2.** A selection of methods provided by the toolkit.

| Object | Returns | C Function Name and Arguments |
|---|---|---|
| *Symbol* | *Symbol* | `TLM_decode_symbol`(*Model*, *Context*, *Coder*) |
| | *Void* | `TLM_encode_symbol`(*Model*, *Context*, *Symbol*, *Coder*) |
| | *Symbol* | `TXT_sentinel_symbol()` |
| *File* | *Void* | `TXT_close_file`(*File*) |
| | *File* | `TXT_open_file`(*Filename*, *Mode*, ...) |
| | *File* | `TXT_load_text`(*File*) |
| | *Char* | `TXT_readline_file`(*File*, *Text*) |
| | *Void* | `TXT_write_file`(*File*, *String*) |
| *Text* | *Text* | `TXT_create_text()` |
| | *Boolean* | `TXT_is_alphanumeric`(*Symbol*) |
| | *Unsign. Int.* | `TXT_length_text`(*Text*) |
| | *Void* | `TXT_release_text`(*Text*) |
| *Model* | *Model* | `TLM_create_model`(*ModelType*, *Title*, ...) |
| | *Model* | `TLM_load_model`(*File*) |
| | *Model* | `TLM_next_model()` |
| | *Void* | `TLM_release_model`(*Model*) |
| | *Void* | `TLM_reset_model()` |
| | *Void* | `TLM_write_model`(*File*, *Model*, *ModelForm*) |
| *Coder* | *Coder* | `TLM_create_arithmetic_decoder`(*SourceFile*, *CompressedFile*) |
| | *Coder* | `TLM_create_arithmetic_encoder`(*SourceFile*, *CompressedFile*) |
| | *Void* | `TLM_release_coder`(*Coder*) |
| *Context* | *Context* | `TLM_create_context`(*Model*) |
| | *Void* | `TLM_release_context`(*Model*, *Context*) |
| | *Void* | `TLM_update_context`(*Model*, *Context*, *Symbol*) |
| *Transform* | *Void* | `TTM_add_transform`(*Transform*, *Codelength*, *SourceFormatString*, *TransformFormatString*, ...) |
| | *Transform* | `TTM_create_transform`(*TransformAlgorithm*, ...) |
| | *Text* | `TTM_perform_transform`(*Transform*, *SourceText*) |
| | *Void* | `TTM_release_transform`(*Transform*) |
| | *Void* | `TTM_start_transform`(*Transform*, *Type*, *SourceText*, *Model*) |

## 4. Sample Applications Provided by the Toolkit

This section gives sample pseudo-code for the following nine applications:

1. a method that encodes symbols using the PPM compression scheme (Algorithm 1);
2. a context-based encoding method (Algorithm 2);
3. a context-based decoding method (Algorithm 3);
4. a context-based method for computing compression codelength without coding (Algorithm 4);
5. a context-based method for building ("training") models (Algorithm 5);
6. a compression-based method for text classification (Algorithm 6);
7. a compression-based method for verifying sentence alignment in a parallel corpus (Algorithm 7);
8. a compression-based method for performing word segmentation on a text file (Algorithm 8);
9. a compression-based method for performing language segmentation on a text file (Algorithm 9).

These examples are used to illustrate the use of methods from the toolkit's libraries. Also, a further purpose of the examples is to illustrate the brevity of the solutions especially the algorithms for word segmentation and language segmentation considering the complexity of each of these applications which is hidden to the most extent from the developer.

In addition, these algorithms adopt the key insight of separation of encoding into *modelling* and *coding* processes as proposed by Rissanen and Langdon [25] which was an important and seminal contribution for the field. There are a number of papers that have improved on this simple design in often subtle ways that at first glance seem trivial (see, for example, [6,27]). These subtle variations are also very important contributions to the field. We argue that the algorithms included in this paper also provide similar subtle variations that can be considered contributions in the same vein.

### *4.1. A PPM Encoder*

The method `PPMEncodeFile` as detailed in Algorithm 1 encodes symbols (such as bytes or characters) in an input file using the PPM compression scheme. The input file is passed to the method as the argument *SourceFile* with the compressed output file specified by the argument *CompressedFile*.

**Algorithm 1:** A method for order 5 PPMD encoding

```
1 method PPMEncodeFile (SourceFile, CompressedFile)
2 Model = TLM_create_model (TLM_PPM_Model,"Title", 256, 5,"D", True, True);
3 Coder = TLM_create_arithmetic_encoder (SourceFile, CompressedFile);
4 EncodeFile (SourceFile, Model, Coder);
5 TLM_release_model(Model);
6 TLM_release_coder(Coder);
7 end
```

The variable *Model* specifies the model number of the PPM compression language model—this number is returned by the call to `TLM_create_model` for creating a new dynamic PPM model (line 2). The arguments to the call specify the model's title, that the alphabet size is 256, that the order of the model is 5, that escape method D is being used (that is, the model is PPMD), and that both full exclusions and update exclusions are being used. As an alternative, `TLM_load_model` can be used to load an existing static or dynamic model directly from disk rather than have it created as this method does.

The variable *Coder* is set to the coder number of the arithmetic coder created for the method that is returned by the `TLM_create_arithmetic_encoder` method (line 3). The coder is used to perform the arithmetic encoding and decoding used for encoding and decoding a stream of bits that represent the PPM model's predictions to a compressed file on disk.

Algorithm 1 calls the `EncodeFile` method (line 4) which is defined in Algorithm 2 and described in the next subsection. The method finishes by releasing the memory allocated to the *Model* and *Coder* that were created (lines 5 and 6). These release methods are necessary for implementations in programming languages that do not automatically free memory on exit of the method. Many of the objects used by the toolkit (such as the *Context*, *Model*, *Coder* and *Transform* objects) require complex multiple data structures for implementation and can take up considerable resources when many objects are created during Viterbi-style search operations, for example. Freeing up these resources is non-trivial and dedicated methods for performing these operations are required and therefore explicitly defined by the toolkit.

### *4.2. Context-Based Encoding and Decoding*

Algorithms 2 and 3 provides pseudo-code for context-based encoding and decoding of a file respectively. They assume that a *Model* has already been created or loaded and a *Coder* has been created prior to these methods being called with these being passed as arguments. The encoder requires the input file (*SourceFile*) as an argument, and the decoder requires the compressed file (*CompressedFile*) as an argument.

**Algorithm 2:** A context-based method for encoding

```
1 method EncodeFile (SourceFile, Model, Coder)
2 Context ← TLM_create_context (Model);
3 for each Symbol in SourceFile do
4 |   TLM_encode_symbol (Model, Context, Symbol, Coder)
5 end
6 TLM_encode_symbol (Model, Context, Coder, TXT_sentinel_symbol());
7 TLM_release_context (Context);
8 end
```

**Algorithm 3:** A context-based method for decoding

```
1 method DecodeFile (CompressedFile, Model, Coder)
2 Context ← TLM_create_context (Model);
3 while Symbol ← TLM_decode_symbol (Model, Context, Coder) != TXT_sentinel_symbol() do
4 |   Output Symbol
5 end
6 TLM_release_context (Context);
7 end
```

Both methods require that a *Context* variable be created first using the `TLM_create_context` method (lines 2). Then each symbol is processed sequentially, either encoded using the `TLM_encode_symbol` method (line 4, Algorithm 2) or decoded using the `TLM_decode_symbol` method (line 3, Algorithm 3). (These methods also update the *Context* at the same time for the current symbol. The *Context* is released just before the end of both methods).

Both methods process symbols sequentially which are specified using the variable *Symbol*. The way a file is preprocessed (during encoding) and postprocessed (during decoding) is user-defined. Symbols for English text are usually mapped directly to the ASCII byte values of the characters in the file. For Unicode (e.g., UTF-8) encoded files, on the other hand, a different mapping is required. Further preprocessing/postprocessing of the symbol stream may also be performed such as replacement of common bigraphs [6,14,15] (also see Section 5 below) as this can often lead to improved compression. However, these steps are not detailed in Algorithms 2 and 3.

The special sentinel symbol (returned by `TXT_sentinel_symbol`) has been used to signify the end of the encoding stream and is encoded once all the other symbols in the file have been encoded (line 6, Algorithm 2). It is a unique character which effectively expands a model's alphabet by one that is encoded at the end of an encoding stream and forces the model to back off to the default order -1 context. The decoder simply decodes symbols until the sentinel symbol is decoded (line 3, Algorithm 3). An alternative way of encoding the end of the stream which the toolkit implements is to encode the length of the stream at the beginning using (for example) a start-stop-step encoder [28]. However, this requires a first pass through the stream being encoded to find its length, and using a unique extra symbol such as `TXT_sentinel_symbol` does not incur much overhead except for the cost of the escapes down to the order $-1$ context and the cost of encoding this symbol in the lowest order context which is ameliorated by PPM's full exclusions mechanism.

### *4.3. Calculating Compression Codelengths without Coding*

The method `CodelengthText` in Algorithm 4 calculates the compression codelength for a text object without performing any coding operation. It is similar to the encoding and decoding methods above, except that it uses the routine `TLM_update_context` to update the model (line 5) instead of either `TLM_encode_symbol` or `TLM_decode_symbol` as above. The argument *Model* specifies

the language model being used. The text *SourceText* passed as an argument would usually have the sentinel symbol marking the end of the text so therefore this does not need to be explicitly processed at the end of the stream (as was the case with Algorithm 2). The variable *Codelength* (line 2) is used to accumulate the totals of the compression codelengths as each symbol in the text is processed. It is returned by the method at the end (line 9). The codelength values that are the cost of encoding each symbol are obtained by accessing the variable `TLM_codelength` (see line 6) and reflect the theoretically optimum encoding of each symbol for which the actual (physical) encoding to disk using an arithmetic coder is a very close approximation [27,29]. This method can be used to speed up processing as it does not perform any physical coding and has wide applications. A variant of the algorithm has been used to perform the automatic cryptanalysis of transposition ciphers, for example [30].

**Algorithm 4:** A context-based method for computing compression codelength without coding

```
1  method CodelengthText (SourceText, Model)
2  Codelength ← 0.0;
3  Context ← TLM_create_context (Model);
4  for each Symbol in SourceText do
5  |  TLM_update_context (Model, Context, Symbol);
6  |  Codelength += TLM_Codelength;
7  end
8  TLM_release_context (Context);
9  return Codelength;
10 end
```

### *4.4. Building Models*

The method `TrainFile` in Algorithm 5 builds (or 'trains') a model from the text in the file *SourceFile* passed as an argument. The way the algorithm works is similar to Algorithm 5 above as it uses `TLM_update_context` (line 4) to process each symbol sequentially (line 3) without performing any coding. However, no codelength calculations need to be performed in this case since all we are doing is building a model.

**Algorithm 5:** A context-based method for building models

```
1  method TrainFile (SourceFile, Model, ModelType, ModelFilename)
2  Context ← TLM_create_context (Model);
3  for each Symbol in SourceFile do
4  |  TLM_update_context (Model, Context, Symbol)
5  end
6  TLM_update_context (Model, Context, TXT_sentinel_symbol());
7  TLM_release_context (Context);
8  TLM_write_model (ModelFilename, Model, ModelType);
9  end
```

The method can be used to build a dynamic model (which keeps on being updated when further files are processed at a later time) or a static model (which does not subsequently change and therefore will always give the same probability estimates). This explains the need for the argument *Model* to specify an already existing model which may be dynamic. (It will have been initially created using a call to `TLM_create_model` similar to that on line 2 in Algorithm 1). The argument *ModelType* specifies whether the model is dynamic or static. The model is written out to the file named *ModelFilename*

(using the call to `TLM_write_model` on line 8) and can be latter loaded directly using a call to `TLM_load_model` without the need for symbol processing. (Note that the size of a model, such as a PPM model, will usually be significantly smaller than the size of the text used to train it, even for higher order models).

Once a dynamic model is written out as a static model, it cannot be subsequently updated. The use of static models can lead to better performance in some tasks (such as text classification) and optimisation of model memory allocation and data structures can be performed once it is known that the model will not be updated in the future.

Usually the symbols in the text will be processed directly from the source training file (from *SourceFile*) rather than from an intermediate text (from *SourceText*). For example, in the following text classification application, compression codelength calculations for multiple models need to be performed on the same source file, and therefore it is more efficient to load the source file once into a text object and have it available for subsequent processing without the need for further I/O operations being performed on the same file.

### *4.5. Text Classification*

The method `ClassifyFile` in Algorithm 6 performs text classification on a text file. It classifies the file *SourceFile* by finding which model compresses the text in the file best. This symbol-based method of text classification has been found to be competitive in many classification tasks compared to standard feature-based (word) classification schemes often with state-of-the-art results [31]. It has been used to accurately identify the natural language in which the source text is written [32], and to identify the most likely author of the text [17], for example. Recent results have shown that it can classify gender [33], recognise emotions [34] and even identify whether the text has been written by a person who is dyslexic or not [35].

**Algorithm 6:** A compression-based method for text classification

```
1  method ClassifyFile (SourceFile)
2  BestCodelength ← 0.0;
3  BestModelTitle ← NULL;
4  SourceText ← TXT_load_text (SourceFile);
5  TLM_reset_model ();
6  while Model ← TLM_next_model () do
7  |  Codelength ← CodelengthText (SourceText,Model);
8  |  if (BestCodelength = 0.0) or (Codelength < BestCodelength) then
9  |  |  BestCodelength ← Codelength;
10 |  |  BestModelTitle ← TLM_get_title (Model);
11 end
12 TXT_release_text (SourceText);
13 return BestModelTitle;
14 end
```

The method calls `TLM_reset_model` and `TLM_next_model` to cycle through all the models that have previously been loaded into memory (lines 5 and 6). The compression codelength is calculated on line 7 by calling the `CodelengthText` method that was defined in Algorithm 4. `TXT_load_text` is used to load the source file once into the text object *SourceText* (line 4). Note that the routine `TXT_load_text` appends the sentinel symbol onto the end of the text.

The method works by cycling through all the models to compute their compression codelengths on the source text, and returns the title (obtained using the call to `TLM_get_title` on line 10) of the best performing model.

### *4.6. Verifying Sentence Alignment of a Parallel Corpus*

The method `VerifyParallelFiles` in Algorithm 7 checks the quality of sentence alignment between two files (*File1* and *File2*) in a parallel corpus. It assumes that the corpus has already been aligned and the purpose of this method is to verify its quality prior to being used to build language models for statistical machine translation. The method also assumes that the sentences in the two different languages are on corresponding separate lines in the two files. Therefore, the `TXT_readline_file` (line 5) is being used to read each line of the two files simultaneously into two text objects *Text1* and *Text2*.

The method `CodelengthText` from Algorithm 4 is used to calculate the compression codelengths of the two sentences in *Text1* and *Text2* according to the models *Model1* and *Model2* respectively. In order for this method to be effective, it requires the two models to be separately trained on a large amount of text that is representative of the respective languages. Recent research [36,37] has shown that comparing compression codelength in this manner is an effective way for verifying that corresponding sentences are aligned, as well as measuring the quality of translation.

The method specifically computes the codelength ratio (*CR*) measure on lines 9 to 12 which is the maximum of the two ratios *Codelength1*/*Codelength2* and *Codelength2*/*Codelength1*. This ensures that *CR* is always greater than or equal to 1.0. A value of 1.0 indicates that the two codelengths are the same value and values close to or equal to 1.0 provide strong evidence that the sentences are probably aligned. This is based on the assumption that the information (from an information theoretical sense) between corresponding sentences that are co-translations should be the same or similar, and compression codelength provides an excellent means for measuring the information [37].

**Algorithm 7:** A compression-based method for verifying parallel sentence alignment

```
1  method VerifyParallelFiles (File1,File2, Model1, Model2, CRThreshold)
2  Text1 ← TXT_create_text ();
3  Text2 ← TXT_create_text ();
4  LineNo ← 0;
5  while TXT_readline_file (File1, Text1) and TXT_readline_text (File2, Text2) do
6  |  LineNo ← LineNo + 1;
7  |  Codelength1 ← CodelengthText (Text1,Model1);
8  |  Codelength2 ← CodelengthText (Text2,Model2);
9  |  if Codelength1 > Codelength1 then
10 |  |  CR ← Codelength1/Codelength2
11 |  else
12 |  |  CR ← Codelength2/Codelength1
13 |  if CR > CRThreshold then
14 |  |  Output "Check alignment for Line " LineNo
15 end
16 TXT_release_text (Text1);
17 TXT_release_text (Text2);
18 end
```

The method on line 13 compares *CR* to a threshold *CRThreshold* which is passed as an argument. If *CR* exceeds this threshold, then a message is output with the line number of the suspicious sentence (line 14). Values of 1.8 and above for *CRThreshold* have been found effective at locating unsatisfactory translations in experiments with Chinese/English [37] and Arabic/English [36] parallel corpora. These experiments also show that codelength-based measurements are more effective than sentence length-based measurements for verifying sentence alignment.

### *4.7. Word Segmentation*

The method `WordSegmentlFile` in Algorithm 8 performs word segmentation on the file *SourceFile*. Specifically, it will insert spaces into the file which usually contains no spaces to indicate where the words are. Word segmentation is an important NLP application for languages that are not naturally word segmented such as Chinese. The method shown in Algorithm 8 is based on the PPM-based word segmentation method [9] that has been used as a baseline model for an international evaluation on Chinese word segmentation [38].

The method uses a transform object *Transform* to perform the segmentation. It is created on line 3 using `TTM_create_transform` which specifies that the Viterbi search algorithm be used to find the best segmentation (as measured by the sequence of text with spaces inserted which has the lowest compression codelength). The method then specifies using `TTM_add_transform` that two transformations be used for the search (lines 4 to 8). This defines the cost of the transformation in codelength (second argument to `TTM_add_transform`), the symbol sequence that needs to be matched (third argument) specified using a `printf`-like string syntax borrowed from the C programming language, and the transformed sequence (fourth argument) also specified using `printf`-like string syntax. The first transformation added to the transform model in the method on line 4 specifies that whatever the current symbol is (using the wildcard "%w" for the third argument to indicate that any symbol should be matched), it should always be included unchanged as one pathway in the search tree. This is because the wildcard "%w" is also being used for the fourth argument meaning that whatever symbol matches it should be transformed unchanged. Line 4 is typically found in most transform-based applications since usually the original text symbols need to form one search pathway rather than be totally discarded and/or transformed into something else.

**Algorithm 8:** A compression-based method for word segmentation

```
1  method WordSegmentFile (SourceFile, Model, Alphanumeric?)
2  SourceText ← TXT_load_text (SourceFile);
3  Transform ← TTM_create_transform (TTM_Viterbi);
4  TTM_add_transform (Transform, 0.0, "%w", "%w");
5  if not Alphanumeric? then
6  |  TTM_add_transform (Transform, 0.0, "%w", "%w  ");
7  else
8  |  TTM_add_transform (Transform, 0.0, "%f", "%f  ", TXT_is_aplhanumeric);
9  TTM_start_transform (Transform, TransformType, SourceText, Model);
10 TransformText ← TTM_perform_transform (Transform, SourceText);
11 TTM_release_text (SourceText);
12 TTM_release_transform (Transform);
13 return TransformText;
14 end
```

The method uses the argument *Alphanumeric*? to indicate which of the second transformations (line 6 or line 8) should be applied to the search. If this argument is not true, then a space symbol is inserted prior to each symbol (there is a space included at the beginning of the transformed sequence " %w" for the fourth argument on line 6). This transformation would typically be used for word segmentation of Chinese text, for example, since in this case, we would wish to search through all the possible insertions of spaces after each of the symbols in the text. If the argument *Alphanumeric*? is true, on the other hand, the transformation on line 8 specifies that spaces only be inserted after alphanumeric characters as a possible search pathway (using the " %f" option for the matching and transformed sequences to specify that the function `TXT_is_alphanumeric` be used to match the symbol). This transformation is a useful optimisation if English text without spaces is being

segmented (by finding where to put the spaces back) since it will substantially reduce the search as non-alphanumeric characters will remain unchanged and not cause an extra pathway to be added to the search tree.

The effect of having two transformations used in this method is that two search pathways will be generated for every symbol being transformed. Hence, the maximum size of the search tree is essentially $2^n$ where $n$ is the number of symbols being transformed. The search is started using `TTM_start_transform` on line 9 which initialises the search for the best transformation of the *SourceText* using the *Model*. The Viterbi algorithm that is executed by the operation `TTM_perform_transform` (line 10) substantially reduces this search using trellis-based techniques and other optimisations so that for Markov-based models (such as PPM models) the search is completed essentially in linear time. The method returns the transformed text *TransformText* which has the best compression codelength compared to alternatives once the search performed by `TTM_perform_transform` has been completed.

### *4.8. Language Segmentation*

The method `LanguageSegmentlFile` in Algorithm 9 performs language segmentation. That is, it marks the boundaries between different languages in the same file. This is an important NLP application for texts that contain multiple languages which are not explicitly marked up, and also for languages where code-switching is common, for example Arabic and Welsh, as an important early step that should be performed on the NLP pipeline.

**Algorithm 9:** A compression-based method for language segmentation

```
1  method LanguageSegmentFile (SourceFile)
2  SourceText ← TXT_load_text (SourceFile);
3  Transform ← TTM_create_transform (TTM_Viterbi);
4  TTM_add_transform (Transform, 0.0, "%w", "%w");
5  TLM_reset_model ();
6  while (Model ← TLM_next_model ()) do
7  |  TTM_add_transform (Transform, 0.0, "%w", "%m%w", Model);
8  end
9  TLM_reset_model ();
10 while (Model ← TLM_next_model ()) do
11 |  TTM_start_transform (Transform, TransformType, SourceText, Model);
12 end
13 TransformText ← TTM_perform_transform (Transform, SourceText);
14 TTM_release_text (SourceText);
15 TTM_release_transform (Transform);
16 return TransformText;
17 end
```

The method creates a transform object *Transform* on line 3 and specifies that the Viterbi algorithm be used to perform the search for the best transformation. The method then adds a transformation using `TTM_add_transform` on line 4 that uses the wildcard "%w" form to ensure that the current symbol unchanged is included as one of the search pathways. Then the "%m" form is inserted as an alternative pathway for each model that has been loaded into memory (lines 5 to 8). This form states that for the transformed text, the model which is passed as the argument to `TTM_add_transform` on line 7 will be used to predict subsequent symbols in the transformed text.

As with the previous word segmentation in Algorithm 8, the effect of having the transformations used in this method is that multiple search pathways will be generated for every symbol being processed. In this case, all possible segmentations of the text using the models that have been loaded will be searched. Hence, the maximum size of the search tree in this case is $m^n$ where $m$ is the number of models used and $n$ is the length of the text. The search for the best transformation of the *SourceText* is started using `TTM_start_transform` for each model on line 11. Again, the Viterbi algorithm which is executed by `TTM_perform_transform` (line 13) substantially reduces the search using trellis-based techniques and other optimisations and the method returns the transformed text *TransformText* which has the best compression codelength once the search has been completed.

### *4.9. Applications Implemented by the Toolkit*

The algorithms detailed above have all been implemented using the Tawa toolkit's libraries in the C programming language. Table 3 summarises these applications that come bundled with the toolkit. The table lists for each application its name (in bold font), a short description (in bold and italic fonts), the algorithms detailed above they are based on and selected arguments. The **encode**, **decode** and **train** applications for PPM compression, decompression and model building, for example, use similar arguments which specify amongst other things the argument size, order of the model, and whether the model performs full exclusions and update exclusions. The **train** application has additional arguments for specifying the title of the model, and whether the model should be written out as a static model. The **markup** and **segment** applications use arguments for specifying whether Viterbi search should be adopted, and the stack depth if the stack algorithm should be used instead.

These application arguments provide a wide range of possible settings that have proven to be very effective in many NLP experiments as summarised in Section 5. Many further possible uses still exist for these applications which have the advantage that they can be used "off-the-shelf" without the need to perform any further development using the toolkit's libraries.

**Table 3.** Applications implemented by the Tawa toolkit with selected arguments.

| |
|---|
| **align—*tool for verifying sentence alignment*** (based on Algorithm 7)<br>`-1 fn` (*fn = parallel text filename for language 1*) `-2 fn` (*fn = parallel text filename for language 2*) `-3 fn` (*fn = model filename for language 1*) `-4 fn` (*fn = model filename for language 2*) |
| **classify—*tool to classify text files*** (uses Algorithms 4 and 6)<br>`-m fn` (*fn = filename for list of models*) `-t fn` (*fn = filename for list of test files*) |
| **codelength—*writes out compression codelengths*** (uses Algorithms 4 and 6)<br>`-c` (*print outs codelengths for each character*) `-e` (*calculates cross-entropy and not codelength*) `-m fn` (*fn = filename for list of models*) |
| **decode—*PPM decoder (decompression tool)* &** (uses Algorithm 3)<br>**encode—*PPM encoder (compression tool)*** (uses Algorithms 1 and 2)<br>`-a n` (*n = size of alphabet*) `-e c` (*c = escape method for the model*) `-F` (*does not perform full exclusions*) `-m fn` (*fn = model filename*) `-N` (*input is unsigned integers*) `-O n` (*n = max order of model*) `-p n` (*report progress every n symbols*) `-U` (*does not perform update exclusions*) |
| **markup—*segments a file by languages*** (uses Algorithm 9)<br>`-D n` (*n = stack depth*) `-m fn` (*fn = model filename*) `-V` (*Use Viterbi algorithm*) |
| **segment—*segments a file by words*** (uses Algorithm 8)<br>`-A` (*Alphanumeric characters only*) `-D n` (*n = stack depth*) `-m fn` (*fn = model filename*) `-N` (*input is unsigned integers*) `-V` (*Use Viterbi algorithm*) |
| **train—*builds PPM models*** (uses Algorithm 5)<br>*same arguments as for* **encode** and **decode** *plus:*<br>`-S` (*writes out the model as a static model*) `-T str` (*title of the model*) |

Some examples of shell commands and output using these applications are shown in Table 4. The purpose of including these here is to demonstrate typical use and also some of the possibilities offered by the commands. Line 1 provides an example of typical use for the **encode** application. In this case the raw text from the million-word Brown corpus of American English [39] in the file `Brown.txt` is being encoded (compressed) to the file `Brown.encoded` using the order 5 PPMD algorithm (which is the default) with an alphabet of 256. Some summary data is output when the program completes such as the number of input bytes, the number of output bytes and the compression ratio of 2.152 bpc (bits per character). Line 2 provides the command that can be used to decode (decompress) the encoded file, and the `diff` command on Line 3 confirms that the decoded file is the same as the original file since no differences are produced as output.

**Table 4.** Sample commands and output using applications implemented by the toolkit.

```
1   encode -a 256 -O 5 <Brown.txt >Brown.encoded
    bytes input 5998525 bytes output 1613517 2.152 bpc
2   decode -a 256 -O 5 <Brown.encoded >Brown.decoded
3   diff Brown.decoded Brown.txt
4   train -S -a 256 -O 5 -e D -T "Brown model" <Brown.txt >Brown.model
    Trained on 5998525 symbols
5   encode -m Brown.model <LOB.txt >LOB.encoded
    bytes input 5877271 bytes output 1609902 2.191 bpc
6   encode -a 256 -O 5 <LOB.txt >/dev/null
    bytes input 5877271 bytes output 1565250 2.131 bpc
7   codelength -e -m models.txt <Brown.txt
    English                     3.514
    French                      5.421
    German                      5.618
    Italian                     5.664
    Latin                       5.400
    Maori                       5.820
    Spanish                     5.602
    Minimum cross-entropy for English =     3.514
8   segment -m Brown.model -V -i woods_unsegm.txt -o woods_segm.txt
9   diff woods_segm.txt woods_correct.txt
    < Of easy wind and down yflake
    > Of easy wind and downy flake
10  markup -m models.txt <e_and_m.txt -V -F
    <Maori>TE MANUHIRI TUARANGI </Maori><English>AND MAORI INTELLIGENCER
    LET THE PAKEHA AND THE MAORI BE UNITED.
    Salutations to you friends the Maori people of New Zealand!  Chiefs and
    commoners, great and small, old and young salutations to you all !
    It is customary among the Maories to welcome the advent of a stranger
    With cries of </English><Maori>Haere mai! Haere mai! </Maori><English>
    The welcome is commenced with the approach
    of the visitor, and is prolonged until he has fairly entered the Kainga,
    Now therefore, the </English><Maori>Manuhiri Tuarangi </Maori><English>
    awaits the welcome of the Maori people.
```

Line 4 provides an example of how the **train** command can be used to build a model. In this case, the Brown corpus is being used to train a static order 5 PPMD model which is written out to the file `Brown.model` when the command completes. Line 5 loads the model directly into memory (without the need for processing the original training text which is much slower), and then uses it to encode another text, the raw text from the million-word LOB corpus of British English [40] in the file `LOB.txt` with a compression ratio of 2.191 bpc for the encoded output file `LOB.encoded`. Line 6 encodes the LOB corpus directly without loading a model, therefore will use a dynamic model that adapts to the text being encoded (i.e., at the beginning, the model is empty but is updated as the text is processed sequentially). The compression ratio of 2.131 for this adaptive approach is slightly better

than the static approach used in Line 5 despite the dynamic model being empty at the start. This is a typical result for PPM models and reflects the effectiveness of these models at adapting quickly to the specific nature of the text. Also, the earlier language in the source document (British English in this case from the LOB corpus) is often a better predictor of subsequent text in the same document rather than using some other source to prime the encoding model in advance (American English from the Brown corpus).

Line 7 provides an example of the use of the **codelength** tool. This application computes the compression codelength without the need for performing any physical coding which is much slower since it requires performing I/O operations. In the example, a list of models trained on Bibles in various languages (English, French, German, Italian, Latin, Māori and Spanish) is specified in the text file (`models.txt`). These models are loaded and then the compression cross-entropies for each of these models is calculated for the source text (the Brown corpus). The output clearly shows that the English Bible model is a better predictor of American English with a compression ratio of 3.514 bpc compared to the compression ratios produced by models for the other languages (all above 5.4 bpc).

The command on Line 8 uses the **segment** tool to perform word segmentation using the Viterbi algorithm on a sample text (`woods_unsegm.txt`) which contains the text for Robert Frost's poem *Stopping by Woods on a Snowy Evening* with spaces removed. The Brown corpus model is used as the training model. The output text file produced by the tool is then compared to the correct text using the `diff` command on Line 9. The output produced shows that the word segmentation tool has made just one mistake: "*down yflake*" instead of "*downy flake*".

The language segmentation tool is being used on Line 10. In this case, the same Bible models specified in the file `models.txt` that were used on Line 7 are used again to define the possible languages for the segmentation, with the Viterbi search being used to find the most compressible sequence for all possible segmentations of languages at the character level. The output produced on the sample text (an extract from the Māori-English Niupepa collection from the NZ Digital Library [41] from the mid-19th to early-20th centuries) shows that only two languages were detected (Māori and English) and the other languages did not appear in the final segmented text that was output. The output shown contains only one error ("*Kainga*" should be tagged as Māori text).

### *4.10. Implementation Details, Memory Usage and Execution Speeds*

This section discusses some implementation details for the toolkit applications, as well as their typical memory usage and execution speeds. Table 5 provides execution speeds and memory usage for the commands in Table 4 (ignoring commands 3 and 9 which do not use any of the toolkit's applications). The execution speed is calculated from the average of 10 runs of the command on a MacBook Pro (15-Inch Retina display running OSX El Capitan Version 10.11.6 with 2.5 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 memory). The memory usage is provided in megabytes and represents the maximum resident set size as reported by the `/usr/bin/time -l` command.

**Table 5.** Average execution speeds across 10 runs and memory usage (maximum resident set size) for the commands in Table 4.

| Command | 1 | 2 | 4 | 5 | 6 | 7 | 8 | 10 |
|---|---|---|---|---|---|---|---|---|
| Avg. exec. speed (secs) | 7.272 | 8.041 | 7.124 | 5.027 | 7.217 | 60.029 | 0.185 | 1.586 |
| Memory usage (MB) | 21.05 | 21.07 | 21.65 | 14.17 | 21.95 | 56.60 | 14.36 | 34.18 |

From the average execution speeds in Table 4, commands 1 and 2 show that for the Brown corpus, unprimed encoding is slightly faster than decoding (7.3 s as opposed to 8.0 s). This is a typical result for PPM compression and is due to way the arithmetic coding ranges are decoded by the toolkit. (This result backs up the statement in Section 2 concerning that "the encoding or decoding process are not necessarily equivalent"). In the PPM case, the encoding process is usually quicker but this

depends on the implementation. The average speed for command 4 shows that building a model from the Brown corpus takes a similar amount of time to the encoding. This is because the text processing is similar, and the difference is because the **train** command needs to perform I/O to write out the model at the end whereas the encoding program needs to write out the arithmetic encoding stream as each symbol in the text is processed one-by-one. The result for command 5 shows that using a static (pre-primed) model for encoding purposes is significantly quicker than using a purely adaptive dynamic model (5.0 s versus 7.2 s for command 6). Command 7 loads seven static models and uses each of them to process the Brown corpus text and therefore this takes considerably longer than the other commands as the Brown corpus is processed multiple times using the approach adopted in Algorithm 6. The results for Commands 9 and 10 shows that word segmentation is very fast (just 0.18 s), and language segmentation is also relatively fast (1.6 s). The slower time for the latter is because the branching factor of the tree being searched is seven (for the seven Bible models in different languages) as opposed to just two for the word segmentation problem.

Referring to the memory usage figures in Table 4, commands 1, 2, 3 and 6 all use similar amounts of memory as they need to build in memory a dynamic PPM model for either the Brown corpus or the LOB corpus. Commands 5 and 9 in contrast load a static model of the Brown corpus, so hence use significantly less memory (roughly 14 Mb as opposed to 21 or 22 Mb). Command 7 needs to load the seven static Bible models so therefore uses the most memory of any of the commands; similarly, command 10 also needs to use the same seven models but uses less memory (34 Mb as opposed to 56 Mb) as the text being processed is much smaller.

Table 6 lists the sizes (in bytes) of some sample texts including the Brown and LOB corpora, the English and Latin bibles, and the Wall Street Journal (WSJ) texts from the Tipster corpus available from the Linguistic Data Consortium. The sizes (in bytes) of the static or dynamic PPMD models for different orders when trained on the texts using the toolkit's **train** tool are listed in the table (in columns 4 and 6). These models store the trie data structure that records all the contexts in the training text and their predictions (i.e., using the frequencies of symbols that follow the context) up to the maximum order of the model. So an order 6 model, for example, will contain information about all contexts of lengths 6 down to 0 plus their predictions. The symbol frequencies stored in the trie are used by the toolkit's libraries for estimating probabilities and for arithmetic coding purposes. In order to reduce the size of the dynamic model as it is being updated, each dynamic model also maintains pointers back into a copy of the input text when a context becomes unique and this is stored in conjunction with the model's trie data structure.

**Table 6.** Sizes (in bytes) of text and models, average training execution speeds (in seconds) and dynamic model compressed input percentages for various sample texts.

| Text | Size (Bytes) | Order | Static Size (Bytes) | Avg. Time (Seconds) | Dynamic Size (Bytes) | Avg. Time (Seconds) | Comp. Input (%) |
|---|---|---|---|---|---|---|---|
| LOB corpus | 5,877,271 | 6 | 23,114,666 | 7.859 | 41,067,518 | 8.279 | 16.23 |
| LOB corpus | 5,877,271 | 5 | 12,815,126 | 6.811 | 22,667,634 | 7.177 | 8.35 |
| LOB corpus | 5,877,271 | 4 | 5,766,170 | 6.101 | 10,141,414 | 6.476 | 3.39 |
| LOB corpus | 5,877,271 | 3 | 1,920,470 | 5.996 | 3,350,666 | 6.023 | 0.96 |
| Brown corpus | 5,998,525 | 6 | 24,168,398 | 7.734 | 42,957,274 | 8.343 | 16.73 |
| Brown corpus | 5,998,525 | 5 | 13,357,088 | 7.512 | 23,628,424 | 7.892 | 8.54 |
| Brown corpus | 5,998,525 | 4 | 5,979,854 | 6.061 | 10,515,322 | 6.517 | 3.43 |
| Brown corpus | 5,998,525 | 3 | 1,993,502 | 6.170 | 3,477,030 | 6.468 | 0.97 |
| Brown corpus × 2 | 11,997,050 | 5 | 18,679,790 | 8.050 | 31,132,970 | 8.461 | 0.00 |
| Brown corpus × 3 | 17,995,575 | 5 | 18,679,790 | 8.544 | 31,132,970 | 8.815 | 0.00 |
| English Bible | 4,138,511 | 5 | 4,523,104 | 2.927 | 7,846,200 | 3.035 | 2.79 |
| Latin Bible | 3,399,860 | 5 | 4,229,580 | 2.454 | 7,324,732 | 2.520 | 3.04 |
| Tipster WSJ | 132,193,779 | 5 | 53,915,630 | 233.626 | 95,069,366 | 223.100 | 1.48 |

Also included in the table are two further texts added for illustration purposes to highlight aspects of the static and dynamic model mechanisms provided by the toolkit. The first, indicated by '. . . × 2' in the table, is for when a dynamic model was first created from the Brown corpus and written out, then

the same Brown corpus text was added again after the model was reloaded by the **train** tool, and then eventually the model was written out to a different model file (either as a static or dynamic model). The second, indicated by '$\ldots \times 3$' in the table, is for the case when the Brown corpus was used three times to train the final model.

The table also provides average execution times to build the models across 10 runs (on the same MacBook Pro used for the experiments reported in Table 4) in columns 5 and 7 for static and dynamic models respectively. In general, the time for building a static model takes roughly 1 MB per second and is on average slightly quicker than for building a dynamic model. However, the time required to build models will depend on the complexity of the model and the length of the text with the models trained on repeated Brown corpus text, for example, being more complex and therefore slower to build, and the order 5 model for the much longer Tipster WSJ text having a much faster build rate of 1.85 MB per second.

For the model sizes, the results show that the sizes of the model grow substantially with order, but the rate of growth in model size reduces with higher orders. For example, for the Brown and LOB corpus, the size of the order 6 models is just under twice the size of the order 5 models. This compares with order 4 models being roughly three times the size of the order 3 models. Experiments with PPM models show that model size eventually plateaus even with higher orders [6] (pp. 138–139). A comparison between the static model sizes and the dynamic model sizes shows that the former is consistently between 0.56 and 0.6 of the latter. This is due to the method used to prune the dynamic model's trie stored in the model when it is written out to a file on disk in static form. The size of the static models compared to the size of the Brown and LOB source texts ranges from approximately four times for the order 6 models, twice for the order 5 models and down to once and 0.3 times for the order 4 and 3 models respectively. With much larger training size, however, the ratio of the model size to text size reduces significantly. For example, for the Tipster WSJ text, the size of the static order 5 model is pruned down to just 0.4 of the size of the text.

Further pruning is also possible when a dynamic model is written out in dynamic form as opposed to static form. When the model is written out as a dynamic model, the copy of the input text that is stored in the model can be substantially pruned by removing those parts which no longer have any possibility of being required for future updates of the trie (since the model has a fixed maximum order and the maximum depth of the trie has already been reached for some contexts). The trie pointers have to be changed to point at the new positions in the pruned input text copy. This pruning can lead to substantial reductions in the size of the models being stored. The percentages of the input text that need to be kept are shown in the last column of Table 6. For example, when the order 6 models of the Brown and LOB corpora are written out in dynamic form, the input text copy stored in the model can be pruned to just over 16% of the training text size. This reduces even more to roughly 8% for order 5 models, 3% for order 4 models and just under 1% for order 3 models which represents a substantial reduction for these models whose training texts are close to 6 Mb in size. When the training text is much longer, as for Tipster WSJ, then a much greater percentage of the input text can be pruned out (over 98% with just 1.48% remaining). For the cases where the Brown corpus text is used to update a dynamic model twice or three times, then the entire input text copy can be pruned away (i.e., 0% remains) since all leaf nodes in the trie are at maximum depth according to the maximum order of the model with all contexts now being repeated.

### *4.11. Search Algorithms*

The Tawa toolkit implements several search algorithms for the method that is used to search for the best transformation `TTM_perform_transform` method). Various search algorithms can be used to prune the search space produced when the transformations specified by one or more calls to the `TTM_add_transform` method. Both the Viterbi dynamic programming algorithm [18] and the stack decoding algorithm [26] have been implemented in the toolkit and are available for the developer to trade off accuracy, memory and execution speed. The Viterbi algorithm guarantees that

the segmentation with the best compression will be found by using a trellis-based search—all possible segmentation search paths are extended at the same time and the poorer performing alternatives that lead to the use of the same conditioning context are discarded.

One of the drawbacks of the Viterbi algorithm is that it is an exhaustive search rather than a selective one, unlike sequential decoding algorithms commonly used in convolutional coding which rely on search heuristics to prune the search space [42]. Teahan et al.'s [9] Chinese word segmentation method used a variation of the sequential decoding algorithm called the stack algorithm—an ordered list of the best paths in a segmentation tree is maintained, and only the best path in terms of some metric is extended while the worst paths are deleted. The metric they used was the compression codelength of the segmentation sequence in terms of the order 5 PPMD model. This variation has also been implemented in the toolkit.

In the stack algorithm's ordered list, paths vary in length (essentially, it is like a best first search whereas the Viterbi search is more like a breadth first search). For deleting the worst paths, the traditional method is that the list is set to some maximum length, and all the worst paths longer than this length are deleted. For Teahan et al.'s [8] PPM word segmenter and in the toolkit, another variation was used instead. All the paths in the ordered list except the first one were deleted if the length of the path plus $m$ was still less than the length of the first's path, where $m$ is the maximum order of the PPM model. The reasoning behind this pruning heuristic is that it is extremely unlikely (at least for natural language sequences) that the bad path could ever perform better than the current best path as it still needs to encode a future $m$ characters despite being already worse in codelength. One further pruning method was also found to significantly improve execution speed. As for the Viterbi algorithm, poorer performing search paths that lead to the same conditioning context are discarded. However, unlike the Viterbi algorithm, search paths vary in length, so only poorer performing search paths with both the same length and conditioning context are discarded. These heuristics have been implemented in the toolkit.

In order to illustrate some important aspects of the search process, Figure 3 provides examples of initial search trees for the **segment** and **markup** tools for the small sample text '*the*'. The search tree for this text for the word segmentation problem (using the tool **segment**) is shown in the left box of the figure. The transformed sequences delimited by quotes '' are shown in each node of the tree with an underscore being used (to make it more visible) to indicate where a space has been inserted into the sequence. The root node is for the start of the search where no symbol has been inserted yet. The branching factor of the tree is two as there are two possible transformations—the symbol being processed in the source text is kept unchanged (as represented by the gray nodes); and a space is added after the symbol being processed (as represented by the hashed nodes). So, for example, there are two nodes at level 1 of the tree. Here the first symbol to be processed is the letter '*t*'. Two possible search paths are generated—one for the letter unchanged ('*t*') and one with a space added to it ('*t*_'). At the next level, two further nodes are added for each node at level 1, so there are four nodes at level 2, and consequently 8 nodes at the next level and so on.

The search tree for the language segmentation problem (using the tool **markup**) is shown in the right box. The tree in this example also has a branching of two as only two transformations are being used here for the languages being segmented—English and Māori. Hence the search process is analogous to the word segmentation problem—instead of a space being added or not, the transformation simply decides to encode the symbol being processed using the English model for the gray nodes or using the Māori model for the hashed nodes. The subscripts shown in the transformed sequences within each node indicate which model was used to code each symbol—the subscript 'E' if the English model was used and the subscript 'M' if the Māori model was used instead. If a symbol switches coding from one model to the other, then the sentinel symbol is encoded which will incur a significant cost as it will have to back off to the null context as a result.

Shown to the right of each node in both trees is the compression codelength for the transformed sequence as calculated by the toolkit (based on a static order 5 PPMD model trained on the Brown

corpus for the left box, and static order 5 PPMD models trained on the English and Māori bibles for the right box). The smallest codelength for each level of the tree is shown in bold font. Note that these trees show the complete search tree as if a breadth-first search was being applied. In reality, many of the nodes in the full trees would be pruned away depending on the search algorithm being applied based on these compression codelength values. (For example, for the Viterbi algorithm, all nodes at the same level of the tree with the same conditioning context for the maximum order of the model would be pruned away except for the one with the lowest codelength).

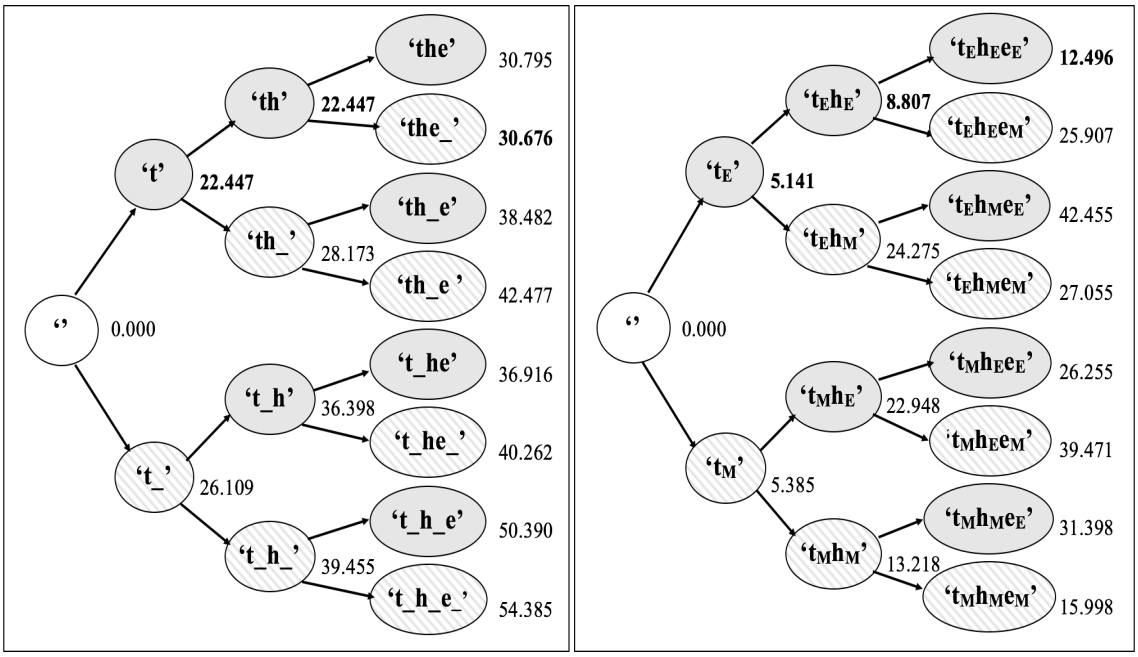


**Figure 3.** Search trees for the sample text ‘*the*’ for the word segmentation problem (**on the left**) and for the language segmentation problem (**on the right**). The compression codelengths in bits for the different sequences are shown on the right of each node with the minimum shown in bold font at each level of the tree.

Table 7 lists some experimental results for the two search algorithms implemented by the toolkit—the Viterbi algorithm and the stack algorithm. In this case, the **segment** tool was used to segment the first 10,000 words of the unsegmented LOB corpus using a static order 5 PPMD model trained on the Brown corpus with the **train** tool. Shown in the table are the following: a description of the algorithm used (including the size of the stack if the stack algorithm was used); the Levenshtein editdistance ($e$) between the correct LOB source text and the output produced by the tool (where the cost of a deletion and an insertion was counted as 1, and of a substitution as 2); the editdistance accuracy as a percentage calculated as $100 - 100 \times e/N$ where $N$ was the size of the source text; the average time in seconds to complete the segmentation task for 10 runs on the MacBook Pro; and the number of nodes that were generated in the search tree as a measure of the memory used. The results show that the Viterbi algorithm produces the least number of errors, and the number of errors reduces with a larger stack size for the stack algorithm as expected. However, the time and memory results show that a smaller stack size runs more quickly (twice as fast when the stack size is only 5) and uses less memory but at a cost in reduced accuracy. Therefore, if speed and/or memory is a concern (when processing many texts or texts that are much larger in size), then accuracy can be traded for improved speed and memory by choosing the stack algorithm. These results mirror those published for the Chinese word segmentation problem using an earlier version of the toolkit [7] and confirm that the latest version produces similar behaviour.

**Table 7.** Results from using various search algorithms for the **segment** tool to segment the first 10,000 words of the unsegmented LOB corpus using a static order 5 PPMD model trained on the Brown corpus with the **train** tool.

| Algorithm | Editdistance | Edistance Accuracy (%) | Average Time (Seconds) | Number of Nodes |
|---|---|---|---|---|
| Viterbi | 493 | 99.17 | 2.657 | 2,391,680 |
| Stack (size = 5) | 787 | 98.69 | 1.315 | 2,338,160 |
| Stack (size = 10) | 620 | 98.96 | 2.339 | 2,345,280 |
| Stack (size = 20) | 568 | 99.05 | 4.369 | 2,348,080 |
| Stack (size = 50) | 533 | 99.11 | 10.386 | 2,443,560 |

## 5. Experimental Results Obtained Using the Toolkit

This section provides a summary in Table 8 of a selection of published experimental results that were produced using the applications in the toolkit. The table lists the following: the publication; the toolkit tools that were used to obtain the experimental results; the specific application area, such as emotion recognition in text, gender and authorship categorisation and so on; and selected results (detailing accuracy, precision and recall where applicable on the various tasks). The purpose is to show the wide variety of application areas that the toolkit can be applied to and also to provide supporting evidence of the effectiveness of the algorithms detailed in Section 4. For example, the classification results achieved using the **classify** tool consistently outperform well-known feature-based approaches such as SVM and Naïve Bayes in many application areas. The reader should refer to the publications directly for full details concerning the experimental methodology involved and the results obtained.

**Table 8.** A summary of selected experimental results produced using the toolkit.

| Reference | Tools Used | Application Area | Selected Results |
|---|---|---|---|
| Alamri and Teahan [35] | **classify**, **train** | Classifying dyslexic text | 100% accuracy on English and Arabic texts |
| Al-Kazaz et al. [30] | **codelength**, **train** | Cryptanalysis of transpos- ition ciphers | 100% decryption of 90 cryptograms; 0.963 recall and precision after space insertion |
| Alkahtani et al. [36] | **align**, **train** | English-Arabic bilingual sentence alignment | 100% at identification of satisfactory and unsatisfactory English-Arabic translations |
| Alkhazi and Teahan [13] | **classify**, **markup**, **train** | Classifying and segmenting Arabic text | Acc. 96%, prec. 0.96, recall 0.96 at classifying classical Arabic and Modern Standard Arabic |
| Al-Mahdawi and Teahan [34] | **classify**, **train** | Emotion recognition in text | Acc. 96%, prec. 0.90, recall 0.99 at classifying Ekman's 6 emotions in LiveJournal blogs |
| Altamimi and Teahan [33] | **classify**, **train** | Gender and authorship categorisation of Arabic tweets | Acc., prec. recall: 91%, 0.89, 0.93 for gender classification; 96%, 0.96, 0.94 for authorship |
| Liu and Teahan [37] | **align**, **train** | English-Chinese bilingual sentence alignment | 94% at identification of satisfactory and unsatisfactory English-Chinese translations |
| Mahoui et al. [4] | **markup**, **train** | Identification of gene function in biological publications | Prec., recall for identication of gene functions: 0.89 and 0.85; 0.85 and 0.94 |
| Teahan and Aljehane [15] | **encode**, **decode** | Grammar-based compression | Improvement in compression between 11% and 20% on the Calgary Corpus |
| Teahan and Alhawiti [23] | **encode**, **decode** | Compression of natural language text | Improvement in compression over standard PPM (e.g., 25% for Arabic, 35% Russian, 32% Persian) |

## 6. Related Work

Many different toolkits, suites, architectures, frameworks, libraries, APIs and/or development environments have been produced over the years for NLP purposes and more specifically for statistical NLP. The use of these terms to describe the functionality and nature of the various NLP systems and approaches have often been inconsistent in the literature. For example, Cunningham [43] stated for the GATE system—a general 'architecture' for text engineering—that it "*is generally referred to as an 'architecture', when, if we were a little more pedantic, it would be called a framework, architecture and development environment*". The meaning of the term 'framework' in software engineering has also evolved over the years—its more specific meaning in recent years requires that a framework has the key characteristic of 'inversion of control' where a framework must handle the flow of control of the

application. This characteristic is not appropriate for the design used for the Tawa toolkit's libraries which do not handle the flow of control in many of its applications (except for when *Transform* objects are used). The meaning of the term 'architecture' is clearer in the literature—it involves the abstract design concepts and structures behind an application. For these reasons for the work described in this paper, the well accepted terms toolkit, libraries and architecture have been used to describe the system, and its design and implementation.

Table 9 lists a selection of systems that have been developed for NLP. The table lists the name of the system, a reference, the type of system described using terms used by the developers of the system and the language(s) in which the system has been developed. The table illustrates that there have been many systems developed to date in a number of different programming languages with Java predominating. The preferred type of system is a toolkit and/or an external library being available. These systems in general provide a suite of tools for various applications, such as tokenisation, classification, POS tagging, named entity recognition and parsing being the ones provided the most.

**Table 9.** Some systems for NLP and their details.

| System | Reference | Type of System | Language(s) |
|---|---|---|---|
| CoreNLP | [44] | suite of tools | Java |
| FreeLing | [45] | library | C++ |
| GATE | [43] | architecture, suite of tools | Java |
| LingPipe | [46] | toolkit, API | Java |
| MALLET | [47] | toolkit, API | Java |
| NLTK | [48] | library | Python |
| OpenNLP | [49] | library | Java |
| Tawa | (This paper) | toolkit, libraries | C |
| UIMA | [50] | architecture, middleware | Java, C++ |

A number of further freely available systems that have been developed by the NLP research community are based on a range of underlying architectures including those listed in Table 9. For example, DKPro and ClearTK are two examples that use the UIMA framework architecture for analysing unstructured information. Alternatively, some of the systems are primarily designed around a core external library or libraries that can be called from an application. Often these systems are not described as having an underlying architecture (for example, FreeLing, NLTK and OpenNLP in the table). The two main approaches all of these systems use for text processing involve a rule-based approach (either manually designed or automatically generated) and statistical. These systems also predominantly process natural language text in a word-based manner.

What distinguishes the Tawa toolkit from most of these systems is threefold: firstly, the toolkit is character-based; secondly, the design of the applications is based around an integrated compression-based architecture; and thirdly, none of the toolkits in the table (apart from Tawa itself), provide explicit support for compression-based applications. Tawa is also written with carefully memory-managed C code for each object type and is designed to be memory efficient and fast.

Like the Tawa system, the LingPipe system also uses character language models but uses the noisy channel model approach for various applications. Using a character or symbol-based approach means that methods used for solving the tasks required for applications can be quite different to the traditional word-based approach. For example, the Tawa toolkit for classification avoids the need for explicit word-based feature extraction and simply processes all the characters in the text being classified.

The application of compression-based models to NLP has had a long history and one of the main purposes of the Tawa toolkit is to make these models more easily available for developers to use. In many cases, the applications implemented using these models have produced state-of-the-art results. For example, PPM character based language models have been applied successfully to many applications in NLP [6] such as language identification and cryptography [51] including various

applications for automatically correcting words in texts such as OCR and word segmentation [8,9] (also see Table 8 for a further selection of results).

Text mining and named entity extraction in Tawa can also be performed using character-based language models using an approach similar to that used by Mahoui et al. [4] to identify gene function. Bratko [52] have also used data compression models for text mining, specifically for spam filtering. Witten et al. [10,11] have shown how text compression can be used as a key technology for text mining. Here, character-based language modelling techniques based on PPM to detect sub-languages of text instead of explicit programming are used to extract meaningful low-level information about the location of semantic tokens such as names, email addresses, locations, URLs and dates. In this approach, the training data for the model of the target text contains text that is already marked up in some manner. A correction process is then performed to recover the most probable target text (with special markup symbols inserted) from the unmarked-up source text. Yeates et al. [12] show how many text mining applications can be recast as 'tag insertion' problems such as word segmentation and identifying acronyms in text [53] where tags are re-inserted back into a sequence to reveal the meta-data implicit in it. Teahan & Harper [17] describe a competitive text compression scheme which also adopts this tag insertion model. The scheme works by switching between different PPM models, both static and dynamic (i.e., that adapt to the text as it is being encoded). Tags are inserted to indicate where the encoder has switched between models. Tag insertion problems, however, are just part of a much broader class of applications encompassed by the noiseless channel model and therefore possible using the Tawa toolkit. The toolkit diverges substantially from the tag insertion model of Yeates et al. [12] (by allowing regular expressions and functions in the specification of both the matching source sequence and corrected target sequence) and arguably also the noisy channel model approach (for example, by making it easier to combine models using hybrid encoding methods or by using preprocessing techniques to modify alphabets).

## 7. Conclusions and Future Work

A novel toolkit for modelling and processing natural language text has been described. (The toolkit can also process general text not just natural language text and has been applied successfully to the processing of the following types of textual data, for example: compression of source code, bibliographic references, Unix terminal session transcriptions and executable program files; analysis of complex system output including cellular automata and flocking simulations; and even the textual transcription of bird song.) The underlying architecture is based on using an encoding approach: that of characterising applications as a problem of searching for the best encoding of the target text given the source text. The toolkit implements various types of compression models, as well as several search algorithms, and various NLP applications. These compression-based models process the text sequentially and base their predictions of the upcoming character on a fixed length finite context of preceding characters. Experimental results have shown that these models are competitive against alternative word-based approaches adopted by most other NLP systems.

Like the noisy channel model, the 'noiseless channel model' architecture adopted by the toolkit models the text processing as sending a message down a communication channel. The design of the toolkit adheres to two main principles—the communication process must be both *lossless* and *reversible*. All information in the message is communicated without loss, and therefore any transformation of the source text to the target text is done in such a way that the source text is recoverable. Unlike the noisy channel model where the source text is unknown when it is decoded, the source text in the approach adopted by the toolkit is known when it is encoded, therefore providing an opportunity that a more effective encoding can be found more efficiently.

Future work will involve the application of this new model to further text processing problems that have not yet been investigated fully such as POS tagging, named entity recognition, OCR and machine translation. Also, other approaches similar to the joint source-channel model approach of Haizhou et al. [54] for machine transliteration will also be investigated where the best encoding of

both the source and target messages are generated simultaneously as this can be easily accommodated using the toolkit's architecture.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The author declares no conflict of interest.

## References

1. Cleary, J.G.; Teahan, W.J. Unbounded length contexts for PPM. *Comput. J.* **1997**, *40*, 67–75.
2. Cleary, J.G.; Witten, I.H. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.* **1984**, *32*, 396–402.
3. Moffat, A. Implementing the PPM data compression scheme. *IEEE Trans. Commun.* **1990**, *38*, 1917–1921.
4. Mahoui, M.; Teahan, W.J.; Thirumalaiswamy Sekhar, A.K.; Chilukuri, S. Identification of gene function using prediction by partial matching (PPM) language models. In Proceedings of the 17th ACM Conference on Information and Knowledge Management, Napa Valley, CA, USA, 26–30 October 2008; pp. 779–786.
5. Kernighan, M.D.; Church, K.W.; Gale, W.A. A spelling correction program based on a noisy channel model. In Proceedings of the 13th Conference on Computational Linguistics, Association for Computational Linguistics, Helsinki, Finland, 20–25 August 1990; Volume 2, pp. 205–210.
6. Teahan, W.J. Modelling English Text. Ph.D. Thesis, University of Waikato, Hamilton, New Zealand, 1998.
7. Teahan, W.J. Text classification and segmentation using minimum cross-entropy. In *Content-Based Multimedia Information Access*; Le Centre de Hautes Etudes Internationales D'Informatique Dcoumentaire; IEEE: Piscataway, NJ, USA, 2000; Volume 2, pp. 943–961.
8. Teahan, W.J.; Inglis, S.; Cleary, J.G.; Holmes, G. Correcting English text using PPM models. In *Proceedings DCC'98*; Storer, J.A., Cohn, M., Eds.; IEEE Computer Society Press: Washington, DC, USA, 1998.
9. Teahan, W.J., Wen, Y.Y., McNabb, R.; Witten, I.H. Using compression models to segment Chinese text. *Comput. Linguist.* **2000**, *26*, 375–393.
10. Witten, I.H.; Bray, Z.; Mahoui, M.; Teahan, W.J. Text mining: A new frontier for lossless compression. In *Proceedings DCC'99*; Storer, J.A., Cohn, M., Eds.; IEEE Computer Society Press: Washington, DC, USA, 1999.
11. Witten, I.H.; Bray, Z.; Mahoui, M.; Teahan, W.J. Using language models for generic entity extraction. In Proceedings of the ICML-99 Worskhop: Machine Learning in Text Data Analysis, Bled, Slovenia, 30 June 1999.
12. Yeates, S.; Witten, I.H.; Bainbridge, D. Tag insertion complexity. In Proceedings of the Data Compression Conference 2001, Snowbird, UT, USA, 27–29 March 2001; pp. 243–252.
13. Alkhazi, I.S.; Teahan, W.J. Classifying and Segmenting Classical and Modern Standard Arabic using Minimum Cross-Entropy. *Int. J. Adv. Comput. Sci. Appl.* **2017**, *8*, 421–430.
14. Alhawiti, K.M. Adaptive Models of Arabic Text. Ph.D. Thesis, Bangor University, Wales, UK, 2014.
15. Teahan, W.J.; Aljehane, N.O. Grammar based pre-processing for PPM. *Int. J. Comput. Sci. Inf. Secur.* **2017**, *9*, doi:.5121/ijcsit.2017.9101.
16. Alkhazi, I.S.; Alghamdi, M.A.; Teahan, W.J. Tag based models for Arabic text compression. In Proceedings of the Intelligent Systems Conference, London, UK, 7–8 September, 2017.
17. Teahan, W.J.; Harper, D.J. Combining PPM models using a text mining approach. In Proceedings of the Data Compression Conference 2001, Snowbird, UT, USA, 27–29 March 2001; pp. 153–162.
18. Viterbi, A.J. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Trans. Inf. Theory* **1967**, *13*, 260–269.
19. Burrows, M.; Wheeler, D.J. *A Block-Sorting Lossless Data Compression Algorithm*; Technical Report 124; Digital Equipment Corporation: Palo Alto, CA, USA, 1994.
20. Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343.
21. Ziv, J.; Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* **1978**, *24*, 530–536.
22. Abel, J.; Teahan, W. Universal text preprocessing for data compression. *IEEE Trans. Comput.* **2005**, *54*, 497–507.
23. Teahan, W.J.; Alhawiti, K.M. Preprocessing for PPM: Compressing UTF-8 Encoded Natural Language Text. *Int. J. Comput. Sci. Inf. Technol.* **2015**, *7*, 41–51.

24. Volf, P.A.; Willems, F.M. Switching between two universal source coding algorithms. In Proceedings of the Data Compression Conference, Noordwijkerhout, The Netherlands 30 March–1 April 1998; pp. 491–500.
25. Rissanen, J.; Langdon, G.G. Universal modeling and coding. *IEEE Trans. Inf. Theory* **1981**, *27*, 12–23.
26. Jelinek, F. Fast sequential decoding algorithm using a stack. *IBM J. Res. Dev.* **1969**, *13*, 675–685.
27. Witten, I.H.; Neal, R.M.; Cleary, J.G. Arithmetic coding for data compression. *Commun. ACM* **1987**, *30*, 520–540.
28. Bell, T.C.; Cleary, J.G.; Witten, I.H. *Text Compression*; Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1990.
29. Rissanen, J.; Langdon, G.G. Arithmetic coding. *IBM J. Res. Dev.* **1979**, *23*, 149–162.
30. Al-Kazaz, N.R.; Irvine, S.A.; Teahan, W.J. An Automatic Cryptanalysis of Transposition Ciphers Using Compression. In *International Conference on Cryptology and Network Security*; Springer International Publishing: Berlin, Germany, 2016; pp. 36–52.
31. Teahan, W.J.; Harper, D.J. Using compression-based language models for text categorization. In *Language Modeling for Information Retrieval*; Springer: Amsterdam, The Netherlands, 2003; pp. 141–165.
32. Khmelev, D.V.; Teahan, W.J. A repetition based measure for verification of text collections and for text categorization. In Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Toronto, ON, Canada, 28 July–1 August 2003; pp. 104–110.
33. Altamimi, M.; Teahan, W.J. Gender and authorship categorisation of Arabic text from Twitter using PPM. *Int. J. Comput. Sci. Inf. Technol.* **2017**, *9*, 131–140.
34. Al-Mahdawi, A.; Teahan, W.J. Emotion Recognition in Text Using PPM. In Proceedings of the AI-2017: The Thirty-seventh SGAI International Conference, Cambridge, UK, 12–14 December 2017.
35. Alamri, M.; Teahan, W.J. *Distinguishing Dyslexic Text from Regular Text Using PPM*; Submitted to Dyslexia; John Wiley and Sons: Hoboken, NJ, USA, 2017.
36. Alkahtani S.; Liu, W.; Teahan, W.J. A new hybrid metric for verifying parallel corpora of Arabic-English in In Proceedings of the Fifth International Conference on Computer Science, Engineering and Applications (CCSEA-2015), Dubai, UAE, 7–8 March 2015.
37. Liu, W.; Teahan, W.J. PPM Compression-based Method for English-Chinese Bilingual Sentence Alignment. In Proceedings of the Statistical Language and Speech Processing Second International Conference, SLSP 2014, Grenoble, France, 14–16 October 2014; Lecture Notes in Computer Science; Besacier, L., Dediu, A.-H., Martín-Vide, C., Eds.; Springer: Berlin, Germnay, 2014; Volume 8791.
38. Sproat, R.; Emerson, T. The first international Chinese word segmentation bakeoff. In Proceedings of the Second SIGHAN Workshop on Chinese Language Processing, Sapporo, Japan, 11–12 July 2003; Volume 17, pp. 133–143.
39. Francis, W.N.; Kučera, H. *Brown Corpus Manual*; Brown University: Providence, RI, USA, 1979.
40. Johansson, S.; Atwell, E.; Garside, R.; Leech, G. *The Tagged LOB Corpus*; Norwegian Computing Centre for the Humanities: Bergen, Norway, 1986.
41. Witten, I.H.; McNab, R. The New Zealand Digital Library: Collections and Experience. *Electron. Libr.* **1997**, *15*, 495–504.
42. Anderson, J.B.; Mohan, S. Sequential coding algorithms: A survey and cost analysis. *IEEE Trans. Commun.* **1984**, *32*, 169–176.
43. Cunningham, H. GATE, a general architecture for text engineering. *Comput. Hum.* **2002**, *36*, 223–254.
44. Manning, C.D.; Surdeanu, M.; Bauer, J.; Finkel, J.; Bethard, S.J.; McClosky, D. The Stanford CoreNLP Natural Language Processing Toolkit. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, Baltimore, MD, USA, 23–24 June 2014; pp. 55–60.
45. Carreras, X.; Chao, I.; Padró, L.; Padró, M. FreeLing: An Open-Source Suite of Language Analyzers. In Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC'04), Lisbon, Portugal, 26–28 May 2004.
46. Carpenter, B. Character language models for Chinese word segmentation and named entity recognition. In Proceedings of the Fifth SIGHAN Workshop on Chinese Language Processing, Sydney, Australia, 22–23 July 2006; pp. 169–172.
47. McCallum, A.K. MALLET: A Machine Learning for Language Toolkit. 2002. Available online: http://mallet.cs.umass.edu. (accessed on 17 October 2017).
48. Bird, S. NLTK: The natural language toolkit. In Proceedings of the COLING/ACL on Interactive Presentation Sessions, Association for Computational Linguistics, Sydney, Australia, 17–18 July 2006; pp. 69–72.

49. Baldridge, J. The Opennlp Project. 2005. Available online: http://opennlp.apache.org/index.html (accessed on 18 October 2017).
50. Ferrucci, D.; Lally, A. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.* **2004**, *10*, 327–348.
51. Irvine, S.A. Compression and Cryptology. Ph.D. Thesis, University of Waikato, Hamilton, New Zealand, 1997.
52. Bratko, A. Text Mining Using Data Compression Models. Ph.D. Thesis, University of Ljubljana, Ljubljana, Slovenia.
53. Yeates, S.; Bainbridge, D.; Witten, I.H. Using compression to identify acronyms in text. *arXiv* **2000**, arXiv preprint cs/0007003.
54. Haizhou, L.; Min, Z.; Jian, S. A joint source-channel model for machine transliteration. In Proceedings of the 42nd Annual Meeting on association for Computational Linguistics, Association for Computational Linguistics, Barcelona, Spain, 21–26 July 2004.