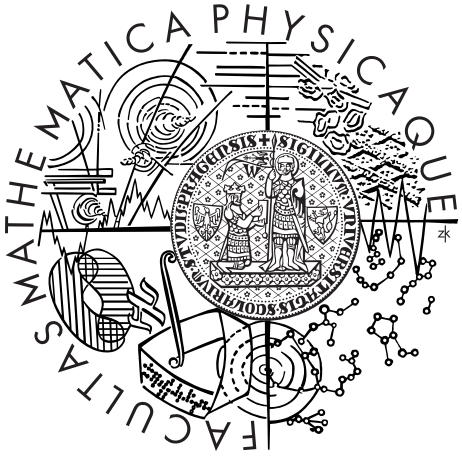
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Jiří Maršík

Fast and Trainable Tokenizer for Natural Languages

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis: RNDr. Ondřej Bojar, Ph.D.

Study program: Computer Science

Specialization: General Computer Science

Prague 2011

Dedicated to the work of Béla Tarr.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature

Název práce: Rychlý a trénovatelný tokenizér pro přirozené jazyky

Autor: Jiří Maršík

Katedra: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: RNDr. Ondřej Bojar Ph.D.

Abstrakt: V této práci představujeme systém pro dezambiguaci hranic mezi tokeny a větami. Charakteristickým znakem programu je jeho značná konfigurovatelnost a všestrannost, tokenizér si dokáže poradit např. i s nepřerušovaným čínským textem. Tokenizér používá klasifikátory založené na modelech s maximální entropií, a jedná se tudíž o systém strojového učení, kterému je nutné předložit již tokenizovaná ukázková data k trénování. Program je doplněn nástrojem pro hlášení úspěšnosti tokenizace, což pomáhá zejména při rychlém vývoji a ladění tokenizačního procesu. Systém byl vyvinut pouze za pomoci multiplatformních knihoven a při vývoji byl kladen důraz zejména na efektivitu a správnost. Po nezbytném přehledu jiných tokenizérů a krátkém úvodu do teorie modelů s maximální entropií se většina textu práce zabývá vlastní implementací tokenizéru a vyhodnocením jeho úspěšnosti.

Klíčová slova: tokenizace, segmentace, maximální entropie, předzpracování textu

Title: Fast and Trainable Tokenizer for Natural Languages

Author: Jiří Maršík

Department: Institute of Formal and Applied Linguistics

Supervisor: RNDr. Ondřej Bojar Ph.D.

Abstract: In this thesis, we present a data-driven system for disambiguating token and sentence boundaries. The implemented system is highly configurable and versatile to the point its tokenization abilities allow to segment unbroken Chinese text. The tokenizer relies on maximum entropy classifiers and requires a sample of tokenized and segmented text as training data. The program is accompanied by a tool for reporting the performance of the tokenization which helps to rapidly develop and tune the tokenization process. The system was built with multi-platform libraries only and with emphasis on speed and correctness. After a necessary survey of other tools for text tokenization and segmentation and a short introduction to maximum entropy modelling, a large part of the thesis focuses on the particular implementation we developed and its evaluation.

Keywords: tokenization, segmentation, maximum entropy, text preprocessing

Contents

Introduction

The goal of this thesis was to provide a fast implementation of a system for disambiguating token and sentence boundaries and to evaluate the implementation both in terms of its accuracy and its speed.

Token and sentence boundary disambiguation may seem trivial at first, and it usually is, but in some occasions it might turn out to be quite complex. Consider the following cases:

(1) On Friday, the 22nd, at around 2 a.m. Dr. T. Adams finished the preliminary examination.

(2) The field tests were to begin on Friday, the 22nd, at around 2 a.m. Dr. T. Adams finished the preliminary examination the night before.

(3) "314 159.26$, about half of the yearly budget, was spent on office redecoration!", protested the disgruntled employee of Vanity, S.p.A.

Even as I was typesetting these examples in LaTeX, I had to explicitly mark some of the periods in the above examples as not being sentence boundaries, as LaTeX likes to insert slightly larger spaces after sentence terminators (so called French spacing). The heuristic used by LaTeX is very simple: if a word-final potential sentence terminator (a period, a question mark or an exclamation mark) follows a capital letter, then it is most likely a part of an abbreviation (or an initial) and so it does not mark the end of a sentence[1] [4].

Such a simple system runs into problems in the examples given above, as we can see that abbreviations do not necessarily end with capital letters and on top of that, a period may serve both as part of an abbreviation and as a sentence terminator. Examples 1 and 2 also show us that the context needed to disambiguate the sentence boundary may be quite far from the boundary in question.

While getting the size of a space correctly down to the last millimeter is certainly a noble goal, there are also some important uses for a more reliable segmenter and tokenizer. When text is being processed and parsed by automatic tools, a common first step is to divide the text into tokens and sentences. A lot of the tools that then work with these tokens assume they are correct and try to analyze them further. As a lot of these tools are getting more and more accurate, it is important we step up the quality of the tokenization process, so that the system's quality is not determined by something as basic as tokenization and segmentation of input.

[1] A more intuitive approach might be to check the case of the following, not the preceding, word.

In the last 20 years, the problem started getting some recognition and several systems were demonstrated. This thesis does not aim to create a new system for tokenization. This work is based on an already existing tokenizer implemented by Ondřej Bojar during the construction of the UMC 0.1 Czech-Russian-English Multilingual Corpus [15, 10].

A key feature of the original tokenizer is its strict segregation of language-dependent knowledge into configurable files. The new implementation expands on this idea and assumes next to nothing about the language being processed except that the sentence and token boundaries are disambiguated by a limited context window described by binary predicates expressed as regular expressions. The tokenizer thus offers a great deal of customizability and a lot of effort has been put into ensuring that the tokenizer will behave as expected and that the behaviour is easy to understand without diverging too much from the original.

Performance, being the motivation behind the current implementation, was also important. Both the original and the new tokenizer rely on a C++ toolkit which handles the mechanics of machine learning [16]. However, the original implementation, being written in Perl, had to access the functionality through a command-line interface passing data through files. The new implementation will have the benefits of using the C++ API directly. Where the old implementation used regular expressions to partition the input and detect potential token and sentence boundaries, the new implementation uses a lexical analyzer generator [26] to generate fast C++ code, compile it and load it at runtime. The new implementation also benefits from the multiple CPUs found on modern computers and uses a high-level parallelism library [3] to perform the various time-consuming tasks of tokenization in parallel.

In Chapter 1, we will look at other systems which tried to tackle the problem and compare them to our tokenizer. In Chapter 2, a brief overview of the maximum entropy method of machine learning will be given. Chapter 3 will familiarize us with the implementation of the tokenizer. Finally, in Chapter 4, we evaluate the speed and accuracy of the tokenizer on several datasets.

1. A Survey of Other Solutions

In this chapter we present an overview of existing systems designed to disambiguate sentence and token boundaries. We examine systems based both on hand-written rules and systems using machine learning methods such as maximum entropy models and decision trees. Next, we look at a system that uses part of speech data to disambiguate sentence boundaries and another system which uses collocation detection techniques. Finally we describe a state-of-the-art Chinese word segmenter. For each of these systems, we describe how our tokenizer can be used to express the same ideas about sentence and token boundary disambiguation.

1.1 RE

The system [13] referred to as RE in [14] is an example of a purely *rule-based* system. It does not need any training data, but instead it relies on explicit linguistic knowledge such as lists of abbreviations and custom regular expressions. The RE system in particular works by scanning the input text for periods and then inspecting the tokens surrounding it. If the surrounding tokens do not match a combination of the user's regular expressions, the period is marked as a sentence boundary.

Our tokenizer also allows the user to define regular expressions against which neighboring tokens will be checked (not only neighboring tokens, a token at any distance can be examined, which can be important as we saw in the introduction). The crucial difference between the RE system and our tokenizer is that the outcomes of all these regular expression tests are not explicitly mapped to the disambiguation of the potential boundary by the programmer or the user. Instead, our system relies on already tokenized data from which it learns how to combine the outcomes of these regular expression tests into a tokenization decision.

1.2 MxTerminator

Contrary to RE, MxTerminator [24] is a *supervised machine-learning* system. This means that the tool has to be supplied with already tokenized data from which the classifier infers the logic behind tokenization. The classifier in this case is based on maximum entropy models, the same mathematical foundation on which our system is built.

The MxTerminator scans the text for a list of potential sentence terminators

and presents the classifier with features of the neighboring tokens. The hard-coded features include the word containing the potential sentence terminator, the words preceding and following it, the presence of particular characters in the current word and whether the current word is a honorific or a corporate designator (e.g. Corp.). All of these are easily expressed using regular expressions and lists of tokens and so it should be quite easy to produce a system very similar to MxTerminator using a specific configuration.

There is also a more general version of the MxTerminator which does not rely on precompiled lists of honorifics and other abbreviations. In this version, the MxTerminator first scans the training data and searches for words containing a period which does not serve as a sentence terminator. The features passed to the maximum entropy classifier then consist only of the trigram of words containing the potential sentence terminator and values describing whether the individual words belong to the abbreviations induced from training data in the previous step. With our tokenizer, the user is free to scan the data ahead and store the induced abbreviations in a file. The tokenizer can then be configured to use the file as a definition for the induced abbreviation feature.

1.3 Riley

Riley [25] uses a method of classification different from the MxTerminator. Instead of using a maximum entropy classifier, he builds a regression tree. The following features are used to disambiguate the period (let a be the word containing the period in question and b the following word):

- Probability of a occurring at the end of a sentence
- Probability of b occurring at the beginning of a sentence
- Length of a
- Length of b
- Case of a
- Case of b
- Any punctuation after the period
- Abbreviation class of a

A training dataset the size of approximately 25 million words was used to estimate the probabilities of individual words occurring near sentence boundaries.

Thanks to such detailed information, the system was found to perform notably well.

The first two features used in the regression tree have a natural counterpart in the maximum entropy model. When the text of a token is being passed to the maximum entropy classifier during training, it estimates a parameter for each type of token encountered and each possible outcome (no boundary, token boundary, sentence boundary). What this parameter does, basically, is that it describes and retains in the model the probability of encountering a specific type together with a specific outcome. The equivalent of a probability of a certain type occurring near the sentence boundary would therefore be the maximum entropy model's parameter corresponding to the event of that type appearing together with the sentence boundary outcome.

As for the length features, the maximum entropy toolkit we employed uses a more general form of a maximum entropy feature which allows for real feature values instead of only binary values (the only such feature supported by our tokenizer is the length of a token). The remaining parameters can be described by binary features defined as regular expressions supplied by the user.

1.4 Satz

The Satz system [21] is another supervised machine-learning system for sentence boundary disambiguation. It is very unique in that it does not rely on the superficial characteristics of the shape of the surrounding tokens. Instead, it passes to the underlying classifier the probability distribution of parts of speech for every token within the context of the potential sentence boundary. It is therefore necessary to supply a lexicon giving the part of speech distribution. If a word is not part of any lexicon, a series of heuristics try to guess a safe probability distribution given the word's suffix, case, internal punctuation etc... Thanks to the generalization provided by the part of speech categories, the system required relatively small amounts of training data to achieve solid performance.

In our system, the user is limited to defining binary features and so passing the probability distributions to the classifier would be out of the question. However, the authors of the Satz system performed an experiment wherein they replaced the non-zero probabilities with ones (basically switching from part of speech probabilities to flags indicating if a given part of speech is possible). The results of this experiment showed that the resulting system was trained faster and performed better than the original. Luckily our tokenizer allows the user to easily define binary features using lists of tokens, i.e. lexicons. The only problem would be the heuristics employed with out of vocabulary words. While all of

them can be easily expressed as regular expressions in our system, there is yet no mechanism to make the tokenizer treat a part of speech found in a lexicon and a part of speech guessed by a regular expression heuristic as the same feature which inhibits generalization.

1.5 Punkt

The Punkt system [14] is an example of an *unsupervised machine-learning* system. This means that Punkt does not need manually tokenized data for training, it learns from raw untokenized text. The data Punkt actually uses for training is the text to be tokenized and so besides the obvious advantage of not having to manually annotate data, the Punkt system does not have to be afraid of different text domains and genres.

The Punkt system processes the input in multiple stages. In the first stage, it tries to determine which period-terminated words are abbreviations. A likelihood ratio is assigned to every such token type in the text describing the strength of the collocational tie between the type and its terminating period. A collocation between a type and a following period is taken as evidence that the type is an abbreviation type. This collocational score is further penalized by the length of the type and multiplied by the number of token-internal periods. Finally, a type's abbreviation likelihood is also exponentially penalized for each instance not followed by a period (so that common verbs in head-final languages are not picked up as abbreviations). All types that score higher than a set threshold are considered abbreviations.

After the abbreviations have been determined, every period not following an abbreviation, an initial or a number is marked as a unambiguous sentence boundary. Now that some sentence boundaries have already been disambiguated, the system studies the input again to infer e.g. frequent sentence starters, which are types which form collocations with preceding sentence boundaries. The rest of the periods are disambiguated in the second stage which examines the specific tokens and their contexts. Disambiguation may come from the orthographic heuristic which examines the case of the following token with respect to how often its type occurred lower-case and upper-case both at the start of a sentence and mid-sentence. The orthographic heuristic is very robust and takes into account that many words are written with upper-case first letters even mid-sentence (such as proper nouns and German nouns). The second stage also uses the collocational tie between the types surrounding the period and whether the following type is a frequent sentence starter as evidence against, resp. for, a sentence boundary.

Punkt also demonstrates its language independence by giving remarkable re-

sults on 11 different languages, all without the need to provide annotated data or perform lengthy parameter tweaking. Emulating Punkt's behaviour using our tokenizer would be nearly impossible, as it would necessarily lose its independence on available annotated data and its ability to train from the input before tokenizing it. On the other hand, our system is able to perform nontrivial tokenization tasks (such as Chinese word segmentation) on top of the sentence boundary disambiguation. It is due to the fact that the Punkt system was designed to solve a very specific problem using linguistic knowledge common to a lot of languages. Our tokenizer is very general, permitting the user to tokenize and segment the text in basically any way that is learnable through binary features expressed with regular expressions or lexicons.

1.6 Chinese Word Segmentation

Several attempts at Chinese word segmentation were made using a maximum entropy classifier. The one developed by Jin Kiat Low, Hwee Tou Ng and Wenyuan Guo in 2005 [17] ranked amongst the highest in the Second International Chinese Word Segmentation Bakeoff [12]. It classifies each character as either a single-character word or as a first, intermediate or last character of a multi-character word. The basic set of features passed to the classifier is:

1. $C_n(n = -2, -1, 0, 1, 2)$
2. $C_n C_{n+1}(n = -2, -1, 0, 1)$
3. $C_{-1} C_1$
4. $Pu(C_0)$
5. $T(C_{-2})T(C_{-1})T(C_0)T(C_1)T(C_2)$

C_n refers to a character at a position relative to the current one, Pu is a predicate checking whether a character is a punctuation symbol and T is a function assigning a character class to characters. The 4 used classes are numbers, dates (symbols for "day", "month" and "year"), English letters and others. Feature templates 2, 3 and 5 use conjunctions of features, which means that for all the possible combinations of values, there is a maximum entropy feature and its corresponding parameter. It was this classifier which motivated the implementation of conjunction features in our tokenizer.

The Chinese word segmenter relies on even more features derived from searching the text for words in a lexicon of known words. In our tokenizer, it would be

quite complicated to check for these words due to the fact that every position is a potential token boundary. This means that the preliminary rough tokens, on which user-defined predicates are tested, are exactly one character long. However, this improvement to the Chinese word segmenter is not that crucial. A bigger issue might be the fact that the Chinese word segmenter trains a classifier to predict the role of a character in a single or multi character word, whereas our classifier predicts whether potential token boundaries are real token boundaries (this means that during training the set of features for maximum entropy is quite different).

2. Maximum Entropy Modelling

In this chapter we present the principles of maximum entropy modelling, how maximum entropy models relate to exponential models and how a maximum entropy model is induced from data. We also discuss which implementations of the technique are available and which one was used in our system.

2.1 Maximum Entropy Models

We want to construct a probabilistic model which gives us a probability $p(a, b)$ of an outcome[1] a occurring with context b. We want this model to be very close to the observed training data, meaning that the data's probability given our model p is high.

However, we do not want the maximum likelihood model because we are aware that the observed data does not cover all the possible situations. Instead, we want a model that shares only some important properties with the observed data. We express these properties as binary functions on the space of events E and we call these functions *features*[2]. In most implementations, including ours, these binary features are restricted to the following form

$$f(a, b) = \begin{cases} 1 & \text{if } a = o \text{ and } \phi(b) \\ 0 & \text{else} \end{cases} \tag{2.1}$$

where o is an outcome and ϕ is a context predicate. We want the constructed model p to share the expected values of these feature functions with the empirical distribution $\bar{p}$. This means that we want the probability of $f(a, b)$ being 1 to be the same in both distributions.

Let us say we have chosen several such features we want retained in our model, now we need to select some model from the set of complying models. This is the point where the maximum entropy principle comes into play. The basic idea of the maximum entropy principle was nicely hinted at by Laplace in his "Principle of Insufficient Reason" [8]:

> When one has no information to distinguish between the probability of two events, the best strategy is to consider them equally likely.

[1] The terminology used in computational linguistics often clashes with the one used in probability theory. What is in probability theory usually known as an outcome is here referred to as an *event*. These events are pairs of *contexts* and *outcomes*, where the context is the data we have available when we want a prediction and the outcome is what we want to predict.

[2] The term features is also commonly used in machine learning to denote a part of the context. When it will be important to differentiate these two meaning in other parts of the work, the term *maximum entropy features* will be used to refer to the features defined here.

We would like to have a distribution which conforms to the requirements imposed by the features but is otherwise unbiased, it is as close to uniform as possible without violating the features' requirements. A standard measure of the uniformity of a distribution is entropy

$$H(p) = -\sum_{x \in E} p(x) \log p(x) \tag{2.2}$$

We would like to find a distribution which adheres to the features' constraints and maximizes entropy. It can be shown [8, 23] that such a distribution is of the following form

$$p(x) = \pi \prod_{j=1}^{k} \alpha_j^{f_j(x)} \tag{2.3}$$

where f_j for $j \in \{1, \ldots, k\}$ are the features we want to retain and $0 < \alpha_j, \pi < \infty$. More interestingly, the maximum entropy model adhering to the features' constraints is equal to the maximum likelihood model having the shape of 2.3 (we call such models *exponential models*).

Given the set of features we want to retain in our model, we can now employ an unrestricted optimization algorithm to find the parameters of the exponential model which maximize the likelihood of the training data.

Once we wrap our minds around the definition of an exponential model and restrain ourselves to the features from 2.1, we can easily imagine what happens when predicting an outcome given a context (i.e. evaluating the probabilities of the context appearing with all the possible outcomes). For each feature f_j of the shape 2.1, the probability of an outcome is multiplied by α_j if and only if the feature's predicate ϕ holds for the current context and the outcome which we are evaluating is equal to the feature's desired outcome o (then the feature function's value is 1). So, for each pair of a predicate ϕ which holds for the given context and an outcome o which forms a feature f_j with the predicate as in 2.1, the predicate votes either for or against the outcome o depending on the value of α_j. The value of α_j, estimated from the training data, is higher if the context predicate ϕ usually implies that we will see the outcome o and lower in the opposite case.

In practice, the features (in the machine learning sense of the word) being passed to the maximum entropy classifier are the predicates which hold for the context in question. The classified outcome is the one voted the most by the above process.

2.2 Available Implementations

There are several notable implementations of maximum entropy estimators available. The one we chose for our tokenizer was the Maximum Entropy Modeling Toolkit for Python and C++ written by Zhang Le [16]. The toolkit offers a nice, clean API with which we are able to feed training events to the estimator and then launch a training procedure which finds the optimal parameters. The resulting model can be easily saved to a file and loaded later. The API is complete with functions for evaluating the probabilities of $(context, outcome)$ pairs and derived convenience functions for predicting outcomes from contexts. The supported parameter estimation algorithms include GIS and L-BFGS. The L-BFGS implementation provided by Jorge Nocedal is written in Fortran with large scale datasets in mind. When the various algorithms for estimating the parameters of a maximum entropy model were evaluated, L-BFGS clearly outperformed the GIS, IIS, gradient and conjugate gradient algorithms [18].

Other implementations were contemplated. However, they would require more effort to integrate seamlessly into our tokenizer. The main reason behind this is that the above-mentioned toolkit is the only one supplying a C++ API. The other toolkits only have either command line interfaces or are written in different languages. Of these toolkits, only a few offer anything worth the effort.

Our problem with the toolkits written in Python and Java (the most notable being NLTK [9] and the Stanford Classifier [22]) is that while predicting an outcome of a potential boundary, we rely on the disambiguation of the preceding boundaries. This means that if we were to use NLTK or the Stanford Classifier for prediction, we would have to switch from C++ to Java or Python to perform the disambiguation for every ambiguous boundary individually. This could be worked around by using the toolkits only for training and then writing our own implementation of the exponential model's evaluate functions.

But even if we did integrate these alternative implementation, the Java implementations would have a hard time outperforming the Fortran L-BFGS implementation. The methods in SciPy might be more viable though.

The only alternative solution which looked intriguing is the Toolkit for Advanced Discriminative Modeling (TADM) by Robert Malouf [19]. The toolkit lacks an API and relies only on a command line interface. This would mean that during training the collected events would have to be stored in a file and TADM would have to be invoked using the `system` function. An implementation of an exponential model would then be needed including loading the model from the file, evaluating the probabilities of events and predicting the most probable outcome.

However, if too much time is spent estimating the model's parameters on a machine which might benefit from the PETSc and TAO optimizations used in TADM, it might be worthwhile to change the tokenizer to use TADM.

3. Implementation

In this chapter we describe the internal design of the tokenizer and provide rationale for the choices behind it. We explore the problem of rough tokenization more deeply as it posed one of the biggest challenges in building the system. Finally, we talk about the multi-threading tools which were used to enable parallelism in the tokenizer.

3.1 Overview of the System

The data flow between the various subsystems can be seen in Figure 3.1.

3.1.1 TextCleaner

Any input which is read by the tokenizer is first processed by the **TextCleaner**. This unit is responsible for decoding the stream of text and optionally removing XML markup and expanding HTML entities and character references. These changes to the input stream (referred to as *cutouts* in the program) are conveyed to the **OutputFormatter** so that they can be undone in the output. This allows the tokenizer to process XML marked up content as if it was plain text. The XML markup thus cannot be broken by and does not interfere with the tokenization process.

3.1.2 RoughTokenizer

The **RoughTokenizer**'s goal is to examine the cleaned input stream and identify both unambiguous and ambiguous token and sentence boundaries. It does so by

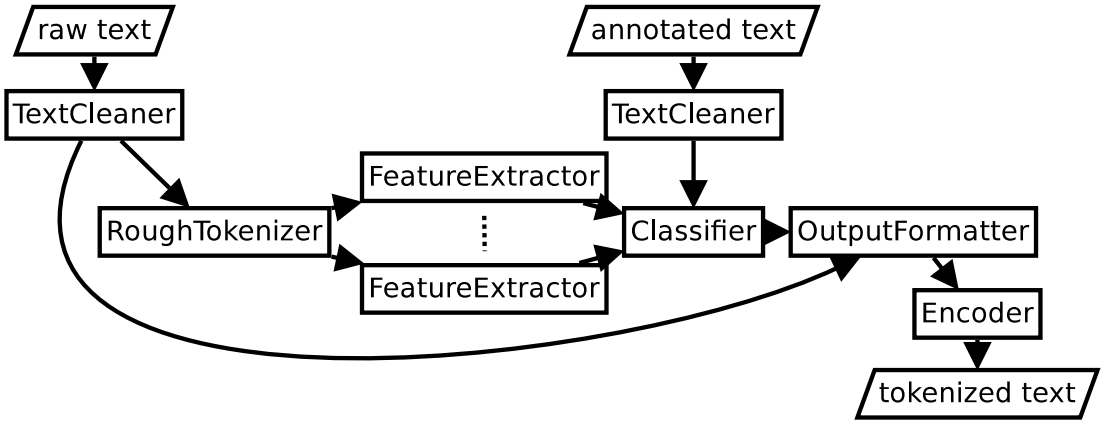


Figure 3.1: Data flow in the entire system

splitting the text into what we call *rough tokens*. In the simplest case, rough tokens are the whitespace delimited words of the text (the term *word* will be used to mean a maximal subsequence of nonwhite characters). However, the user can write regular expressions to define certain points within and between these strings of nonwhitespace characters which may split them up into what end up being the rough tokens. These user-defined points are called *decision points* and they represent the ambiguous token/sentence boundaries.

There are three types of decision points. There is the MAY_SPLIT, which occurs within words and signals a potential token boundary. Then there is the MAY_BREAK_SENTENCE, which occurs before and after certain characters and marks a potential sentence boundary. MAY_SPLIT and MAY_BREAK_-SENTENCE are the decision points which split words into rough tokens. The third type of decision point is MAY_JOIN which occurs between words and turns the space between them from a token boundary to a potential token boundary, making it possible for the two words to join into a single token.

The rough tokenizer detects all decision points in the text and produces a stream of discrete rough tokens with metadata about surrounding whitespace and decision points.

3.1.3 FeatureExtractor

The rough tokens produced by the **RoughTokenizer** are tagged with user-defined *properties* in the **FeatureExtractor**. These predicate properties are defined either using regular expressions or lists of rough tokens. In the case of a regular expression, a rough token is said to have the property the expression defines if and only if the regular expression matches the entire rough token. When a property is defined using a token list, a rough token is said to have the property if and only if it is on the list.

Because the task carried out by the **FeatureExtractor** is a context free function of a single rough token's contents, multiple **FeatureExtractor**s can run simultaneously, each processing a different part of the token stream.

3.1.4 Classifier

The **Classifier** is the interface to the Maximum Entropy Toolkit. It scans the rough token stream for decision points and collects evidential properties from the tokens in the surrounding context. When the tokenizer is being trained, the **Classifier** also reads in an annotated version of the input and aligns it with the rough tokens (the annotated versions have one sentence per line with the tokens delimited by spaces). It then bundles the values of the properties in the context

with the correct outcome inferred from the annotated data and sends them both to the Maximum Entropy Toolkit for training.

When a model is already trained and the tokenizer is tokenizing other data, it queries the model for a predicted outcome given the context and uses the outcome to annotate the rough tokens. The rough tokens are then processed by the **OutputFormatter** which implements the token and sentence breaks predicted by the model.

3.1.5 OutputFormatter

After all the token and sentence boundaries have been disambiguated by the Classifier, it is up to the **OutputFormatter** to convert the stream of rough tokens into plain text where token boundaries are represented by spaces and sentence boundaries by line breaks. It is also the duty of the **OutputFormatter** to undo the changes done by the **TextCleaner**, which means that XML is reinserted into the proper places and former HTML entities and character references replace their expanded counterparts.

3.1.6 Encoder

The **Encoder** receives the text output by the **OutputFormatter** and transcodes it from the internal (UTF-8) encoding to the target encoding. In addition to changing the coding of the characters, the **Encoder** and the **TextCleaner** also serve as additional buffers for I/O operations so that the threads which run the pipeline from **RoughTokenizer** to **OutputFormatter** are less likely to stall on I/O.

3.2 Modes of Execution

The tokenizer has to be trained on annotated data, it has to be able to use that training to tokenize new input and it should also provide accurate feedback on its performance when developing and evaluating a *tokenization scheme* (a tokenization scheme is a set of configuration files controlling the action of the **RoughTokenizer**, the **FeatureExtractor** and the **Classifier**). The tokenizer thus has a few different setups for performing these varied tasks.

3.2.1 Training

When running in the training mode, the tokenizer cleans the input, identifies decision points signalling potential token and sentence boundaries, tags the rough

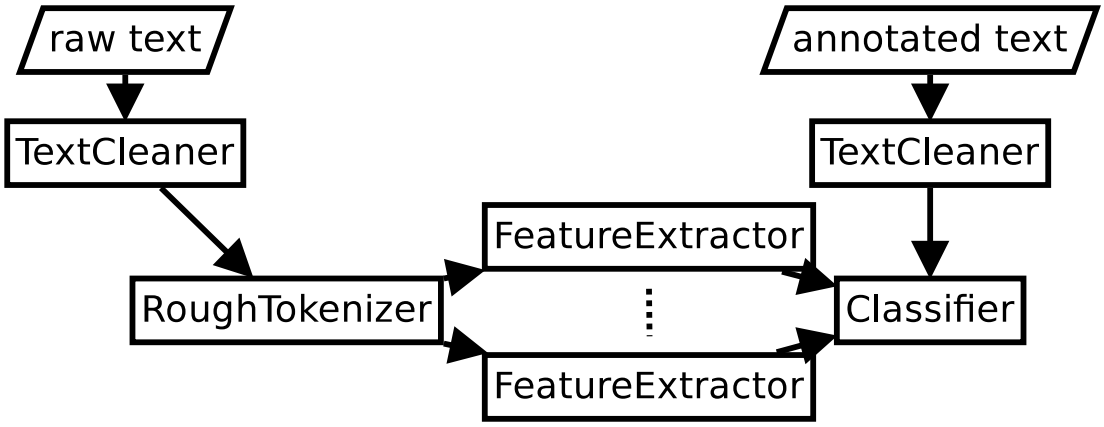


Figure 3.2: Data flow of the system in the training and evaluation configurations

tokens with the user's properties and sends them to the **Classifier**. The **Classifier** aligns this stream of rough tokens with the annotated text. For each decision point, the properties of the tokens within context and the outcome inferred from the aligned data are sent to the Maximum Entropy Toolkit to serve as training data. After all the input files have been processed and the training examples collected, the maximum entropy model is computed and stored in a file for later use.

There is no output processing in the training mode as the only output produced, apart from the saved maxent model file, are warning messages about token and sentence boundaries found in the annotated version which are not even marked as potential boundaries in the raw input. This is a signal to the user that he should perhaps modify the tokenization scheme to account for more possible boundaries or to check his annotated data. The setup of the system can be seen on Figure 3.2.

3.2.2 Tokenization

After a model has been trained, the tokenization mode becomes available. In this mode the text is cleaned, converted into rough tokens and tagged with properties. The **Classifier** has the trained model loaded and predicts the outcome (sentence boundary, token boundary or no boundary) for every decision point given its context. This outcome is used to resolve the MAY_SPLIT, MAY_JOIN and MAY_BREAK_SENTENCE ambiguities and the disambiguation is stored in the relevant rough token's metadata. These annotated tokens are then printed through the **OutputFormatter** and encoded with the **Encoder**. See the setup of the system of this mode on Figure 3.3.

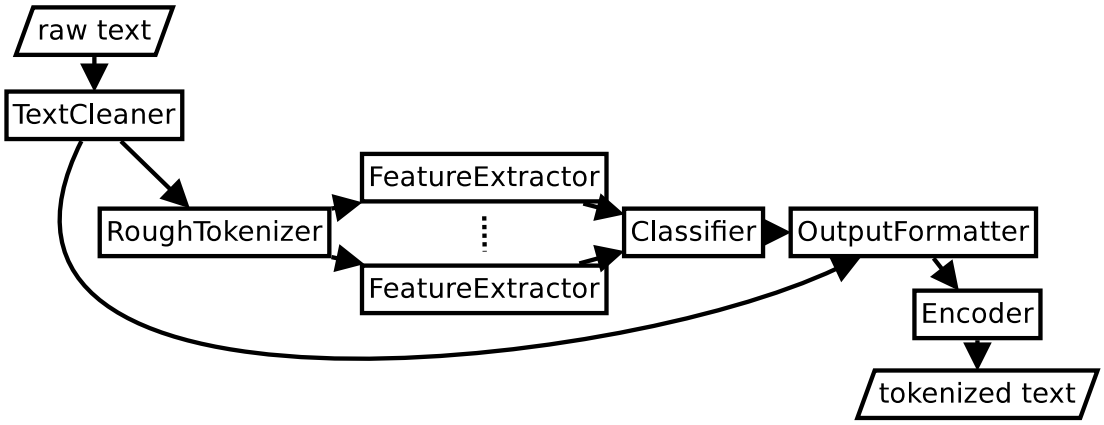


Figure 3.3: Data flow of the system in the tokenization and preparation configurations

3.2.3 Evaluation

When tweaking and developing a tokenization system (the selected training data, the configured parameters in the tokenization scheme) it is vital to have feedback on the shortcomings of your system. The evaluation mode was designed just for this purpose. It works in a way similar to the training mode (see Figure 3.2). The **Classifier** aligns the rough tokens with the annotated text and extracts the contextual properties from the tokens and the true outcome from the annotated data. However, instead of recording them it uses an already trained model and queries it for its predicted outcome. The tokenizer then outputs both the true and the predicted outcome along with the contextual properties.

Another tool can then be used to analyze the tokenizer's output and examine the results and errors of the trained model. An example of such a tool would be the included Python script analyze.py, which scans the evaluation's output and reports the accuracy, precision, recall and F-measure of both sentence and token boundary detection.

This log of outcomes and contexts can be written out when using any of the available modes but only the evaluation mode has access to both the true outcomes from the annotated data and the outcomes predicted by the probabilistic model.

3.2.4 Preparation

The preparation is the last and least essential mode of the tokenizer. It is similar to the tokenization mode (see Figure 3.3), but instead of querying the probabilistic model for an outcome, the **Classifier** simply confirms all potential boundaries (MAY_SPLIT becomes a token boundary and MAY_BREAK_SENTENCE be-

comes a sentence boundary). This produces a file in which an annotator only has to remove spaces and line breaks, where inappropriate, to get the correct annotation.

An advantage to using this mode might be that when the user does not demand the logging of contexts as in the evaluation mode, the time-costly **FeatureExtractor** and **Classifier** can be replaced with a **SimplePreparer**, which only removes the ambiguities in the above-mentioned way.

3.3 Rough Tokenization

One of the first problems encountered when designing the tokenizer was the implementation of rough tokenization. The task of rough tokenization is to take the definitions of decision points and then to be able to detect all such points in any given input.

The possible positions for a MAY_SPLIT decision point are defined by pairs of regular expressions: a position is to be marked as a MAY_SPLIT point if and only if the first expression (prefix) matches some of the characters leading to the position and the second expression (suffix) matches some of the characters following it. MAY_JOIN decision points are defined almost the same way, except that the characters following the position of a MAY_JOIN must start with a string of blank characters and then continue with the string matched by the regular expression. MAY_BREAK_SENTENCE points, on the other hand, are defined simply by two sets of characters. If a position follows a character from the first set or precedes a character from the second set, then that position is a MAY_BREAK_SENTENCE. See Figure 3.4 for an example.

3.3.1 Regular Expression Libraries

The reference implementation of the trainable tokenizer written in Perl used a disjunctive regular expression to match the prefix of the unprocessed input. Our original idea was to use PCRE [6] or some other regular expression implementation [11, 7] to write a similar algorithm.

The naive approach might have us trying to search for the possible suffixes of MAY_JOINs and MAY_SPLITs which are preceded by their respective prefixes. Soon we would learn that finding one decision point may lock us out of finding another one. For example, given the string *abcd* and MAY_SPLIT regular expression pairs *a* - *bc* and *b* - *c*, we match the *bc* according to the leftmost longest match convention properly registering the MAY_SPLIT between *a* and *bc*, but we lose the opportunity to find the MAY_SPLIT between *b* and *c*.

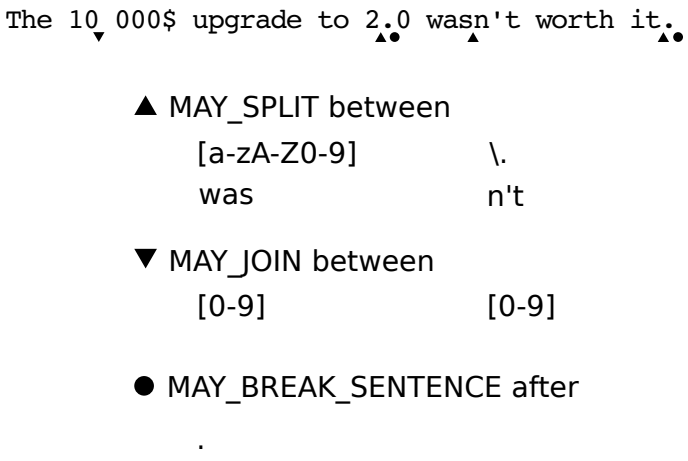


Figure 3.4: An example sentence marked with decision points. The definition of the decision point placement is described below the sentence. The sitting wedge triangle represents a MAY_SPLIT, the upside triangle marks a MAY_JOIN and a circle marks a MAY_BREAK_SENTENCE position. The whitespace and the decision points divide the text into rough tokens.

If we try to search for each of these pairs of regular expressions individually, we might still miss some points as demonstrated by the following example. Let the string in question be *abab* and the MAY_SPLIT regular expression pair *a* - *b(ab)**. Any attempt to search for the suffix *b(ab)** would yield the *bab* substring due to the leftmost longest match convention (and never just the final *b*, which means that position will not trigger a MAY_SPLIT). There are solutions to this problem such as modifying the user's regular expression, modifying the regular expression matching function or searching for the suffix from every position in the text, but they are all either difficult or ineffective.

We do not want to be patching the user's regular expressions because we would probably have to restrict ourselves to a narrower set of regular expressions and even then it would have been challenging to actually implement such a system and prove its correctness. Writing our own regular expression matching engine is also out of the scope of this work. The third option on the list, searching for the suffix (or prefix) from every position in the text, seems like a performance killer. Performance-wise speaking, during the planning phase of development, prototypes of the naive method of regular expression rough tokenization were implemented using both Boost.Regex and PCRE. The average time spent on a 10 MB file with a credible set of splitting and joining rules (breaking English contractions apart, separating words from punctuation etc...) was over 10.8 seconds for Boost.Regex and over 4.9 seconds using PCRE. The tests were performed on a development laptop with the Intel Core 2 Duo T7500 processor.

3.3.2 Lexical Analyzer Generators

During the initial planning, there was another interesting proposal for handling rough tokenization which motivated the early prototypes.

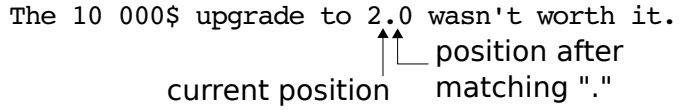
The goal of rough tokenization is to scan large volumes of text and detect patterns described by regular expressions. This kind of problem has been already solved many times using lexical analyzer generators such as flex. These tools take rules, which are pairs of regular expressions and actions written as code. The lexical analyzer generator then creates a program from these rules which reads a stream of text and tries to match a prefix of the yet unmatched input with these regular expressions and reacts to the matches with the supplied actions. More advanced tools enable the definition of several analyzer modes with different rules and enables the actions to switch between them.

The lexical analyzer generator selected for our tokenizer was Quex [26]. Its most important feature is that it is able to work on Unicode code points instead of single-byte characters and that it uses libiconv and ICU to process text in any encoding. Quex can also be very fast because it does not encode the resulting automaton into a table which drives some general program, but instead it generates low level C++ code which mimics the behaviour of the automaton.

The naive way of rough tokenization presented in Subsection 3.3.1 was implemented in a prototype to evaluate the performance benefits stemming from the use of compiled lexers generated by Quex. When run with the same tokenization rules and on the same data as the rough tokenizers in Subsection 3.3.1, the Quex generated lexer finished on average in a little over 0.9 seconds. But the generated lexer was making the same mistakes as the first approaches using regular expressions. As we grew to know more of the functionality available to us in Quex and the specifics of its operation, we were able to arrive at a lexer which detects every possible decision point and does so in about 1.9 seconds on the same data set with which we tested the other methods. The details of this final method are presented in Subsection 3.3.3.

3.3.3 The Solution

Many of the observations about the task at hand made in Subsection 3.3.1 still hold when designing a Quex generated lexer. The final implementation processes the input one character at a time. At each position in the text, rules for matching the suffixes of possible MAY_SPLITs and MAY_JOINs are in play. Each of these rules has a condition in Quex that the preceding text must match the prefix of the respective MAY_SPLIT or MAY_JOIN rule. The MAY_BREAK_SENTENCE rules are implemented in a similar way as their definitions are basically special-



Quex rules:

1. match \. for MAY_SPLIT,
   if preceded by [a-zA-Z0-9]
2. match n't for MAY_SPLIT, if preceded by was
3. match [:space:]*[0-9] for MAY_JOIN,
   if preceded by [0-9]
4. match anything for MAY_BREAK_SENTENCE,
   if preceded by the "." character

Figure 3.5: An example of real-world implemented rough tokenization for the decision points defined in Figure 3.4. The generated Quex lexer is at the signified position in the input. Given its position, the lexer takes into consideration only rules 1 and 3, as these are the only rules whose preconditions have been met. Rule 1 can match the input at the current position and so a MAY_SPLIT is announced and the word read so far ("2") is reported as a rough token. The lexer now automatically advances by the length of the matched string, but we manually step back to the original position in hope of finding more decision points at the current position and the positions within the matched string. When no further decision points are to be found at the current location (as is the case here), we move one character ahead.

izations of the MAY_SPLIT and MAY_JOIN definitions (single characters for prefix or suffix instead of regular expressions). An example of such Quex rules can be seen on Figure 3.5.

When the lexer matches the suffix of a decision point rule, it sends the last characters read since the last decision point or whitespace as a rough token and signals the decision point. Quex would now automatically advance our position in the text right behind the matched suffix, but we override this behaviour and move back to the position of the newly found decision point so other decision points may be found. This alone would cause an infinite loop and so upon returning to the original position we also scratch the detected decision point from the set of applicable rules. If another decision point is found, we do the same until we find all types of decision points at the current location or none of the rules match anymore. In that case, the lowest priority action takes place which reads another character from the stream and starts looking for decision points at the next position.

This scratching out of rules is implemented using 8 different modes for all the different sets of decision points we might be looking for. We start at the topmost mode where we are looking for any of the 3 possible decision points. If one of them is found, we continue at the same location in a mode which looks for the

remaining 2 possible decision points. In the final implementation, there is also a demand for unexpanded HTML entities to be treated as single rough tokens. This demand is met by adding another variable to the state of the lexer (whether we are about to read an entity) which results in the 16 modes seen in the current implementation.

The rough tokenizer thus scans for a regular expression match starting from every possible position within the text, which leads to the worst case time of $O(n^2)$, where n is the length of the text. It is however reasonable to assume that the regular expression matching at any position will never have to process more than a few characters and can thus be regarded as a constant factor irrelevant of the data being processed. This gives us a linear time complexity, which is not unexpected given that we do a single pass over the text with only some small and limited local computation and very little state.

3.3.4 Technical Implementation

In the finished application, the regular expressions which define the placement of decision points are read from user-written configuration files. A Quex source file containing modes for detecting all of the decision points referenced by the user is then output to a temporary file. CMake [1] is invoked to probe the user's system for the compiling essentials, to generate a project for the user's preferred build system and to write the command needed to start the build to a file. This file is read and the command within it is run, which executes Quex on the generated source file and then compiles the result into a shared module. This process is therefore platform-agnostic as it doesn't rely on a specific C++ compiler or build system and uses only CMake and Quex which are multi-platform and are required to build the tokenizer itself.

This compiled shared module is then loaded using the libtool's dynamic loading library [2] which is a wrapper for the platform-specific dynamic loading functions. The tokenizer tracks the set of files used to generate the rough tokenizer along with their timestamps and only regenerates and recompiles it when changes have been made.

3.4 Classification

After the **RoughTokenizer** converts the stream of text into rough tokens annotated with MAY_SPLIT, MAY_JOIN and MAY_BREAK_SENTENCE points, the **FeatureExtractor** goes next. For each rough token in the token stream, it matches the token against a series of regular expressions and word lists which

are represented as one big binary search tree. It marks for each token which regular expressions it matched and in what word lists it was found. These user-defined regular expressions and word lists define what we call "properties", binary predicates describing the aspects of rough tokens relevant to token and sentence boundary disambiguation.

This stream of tokens marked with decision points and user-defined properties then enters the **Classifier**. While the **FeatureExtractor** was driven by the files defining the individual properties used in the tokenization scheme, the **Classifier** is driven by a single mandatory file named "features". In this file, the user selects the features of rough tokens he deems important for the disambiguation. The format of the file has the user specify a range of token offsets followed by a list of relevant properties. The **Classifier** will then look at the rough tokens at the given offsets from the decision point in question and check for the status of the relevant properties evaluated by the **FeatureExtractor** (the offset and the property's name are concatenated into a string and this string is used to name the feature passed to the maximum entropy classifier). However, these properties do not form the only features passed to the maximum entropy classifier:

- For every rough token within the *context* (the smallest possible range of tokens containing all the tokens at the offsets mentioned in the "features" file), a feature is passed describing whether the rough token was followed by whitespace, by a line break or by a paragraph break (multiple newlines).

- For every rough token in the context, any decision points which were found between the token and its successor are passed as features describing the token.

- For every rough token in the context preceding the token in question, the disambiguation of all its decision points is passed as well, so the classifier can see its decisions from the immediate past.

- If several properties at several offsets combined together form compelling evidence for disambiguation, the user can mark them as such in the "features" file and all those properties' values will be concatenated into a large feature string. This way, a single parameter can be trained for each combination of the properties' values, which can be used e.g. to train the tokenizer on specific bigrams (that is the case of the Chinese word segmenter evaluated in Chapter 4).

- Apart from the user-defined regular expression and list properties, the user may call on the predefined "%Word" and "%length" properties. The first

"meta-property" generates a feature string containing the rough token's text, enabling the tokenizer to train not only on the binary features but on unigrams and subsequently bigrams and other n-grams. The "%length" property is special in that it passes a non-binary value to the maximum entropy classifier equal to the length of the rough token.

- For positions within the context which lie past either end of the input (at the beginning or at the end), an out-of-input feature is passed describing the missing token.

A vector containing all these feature strings is passed to the maximum entropy classifier which deciphers the feature names and maps them into the individual trained parameters. It then evaluates the possible outcomes and returns the most probable one. The possible outcomes are JOIN, SPLIT and BREAK_SENTENCE. The outcome describes the space between the current rough token in question (the token at offset 0) and its successor. Depending on the whitespace between the token in question and its successor, the decision points are disambiguated (e.g. a MAY_SPLIT becomes a DO_SPLIT if it lies in a position where there did not use to be any whitespace and the tokenizer classified the position as a SPLIT or BREAK_SENTENCE).

3.5 Parallelism

One of the explicit goals when developing the tokenizer was performance. However, apart from the rough tokenization and the probing of the tokens' properties (the user-defined regular expressions and token lists), the algorithms were quite straightforward. What could be tweaked, however, was the manner of their implementation and execution.

When the task of rough tokenization was isolated from the problem of classifying the potential token and sentence boundaries, a producer/consumer pattern was proposed to process both tasks in parallel. As the design of the system became more detailed, more of the tasks became isolated and the original idea of a producer/consumer pattern changed into the pipeline model seen in the current implementation. Deciding on the pipeline model also let us use libraries which offered high-level pipeline implementations. This meant we did not have to implement the entire system from scratch using threads and synchronization primitives. For the performance payoff of multi-threading, see Subsection 4.2.1

3.5.1 The Pipeline

Threading Building Blocks [3], an open-sourced library developed by Intel, was used for implementing the pipeline. Compared to other multi-platform parallelism solutions, TBB offers high-level algorithms and constructs like the pipeline. It also uses C++ classes and methods to expose its functionality instead of relying on pragma directives like the standardized OpenMP [5].

The pipeline is constructed by setting up an array of filter objects. Each of the filter objects must override the invocation operator and must identify itself as either a parallel or a serial filter (parallel meaning that this filter can be run simultaneously on multiple points of data, serial meaning that the filter processes the input one at a time). In the tokenizer, the **RoughTokenizer**, the **FeatureExtractor**, the **Classifier** and the **OutputFormatter** are all elements of this pipeline (you can see the pipeline as the middle row in Figure 3.1 with **FeatureExtractor** being a parallel filter). The TBB library invokes the first filter, the **RoughTokenizer**, and passes its return value to the **FeatureExtractor** which also produces a value and so on...

Originally, the values to be flowing through the pipeline were individual rough tokens, but the overhead would have been too big. The TBB library doesn't use one thread per pipeline element, instead it is more similar to one thread per value. This way the values are more likely to stay in the cache of the current processor. So it was settled that chunks of rough tokens would be the work units traversing the pipeline. Initially, the idea was to have them statically sized, but since the **Classifier** can consume more tokens than it produces and vice versa, the chunks are now dynamic (e.g. when processing the first chunk, the **Classifier** cannot annotate the final tokens as it has to wait for the next chunk which will inform it about the post-context of those final tokens).

3.5.2 The Input/Output Threads

Initially, the plan was for the pipeline to encapsulate all of the parts of the system. However, it would have been cumbersome to implement the **RoughTokenizer** so that it is a function which receives a chunk of text, feeds it to the buffer of the generated lexer, tries to tokenize the incomplete chunk of text and then sends the retrieved tokens along. The C++ Standard Library already offers a widely used and supported FIFO structure for transmitting continuous text, the **std::iostream**. In its **stringstream** incarnation, it allows one agent to write text to it using the standard output operators of C++ and then later another agent can use the standard input operators to read and parse its contents. Such a standard mechanism would allow us to simply pass a pointer to this stream

to the Quex lexer as if it were a file handle and we would not have to trouble ourselves with any string marshalling.

A class which does just this, the **pipestream**, was implemented by Alexander Nasonov and published on the Boost mailing list in 2003 [20]. However, it didn't meet with much understanding on the list as people tended to associate the class' name, **pipe**, with OS-level pipes. The pipestreams have been resurrected for this project and they made writing the transfer of text between the **TextCleaner**, the **Encoder** and the parts of the TBB pipeline very simple.

The pipestreams were used to connect the **TextCleaner** and **Encoder** to the TBB pipeline. The **TextCleaner** and the **Encoder** both have a do_work method which does all the work. In the case of the **TextCleaner**, it uses a Quex generated lexer to find XML markup and entities in the input file. It optionally transforms these segments, reports them to the **OutputFormatter** and writes the transformed input to an **opipestream** (an output pipestream). The **Encoder** on the other hand reads from an **ipipestream** (an input pipestream), transcodes the text read and writes it to the output file. The use of pipestreams to connect the TBB pipeline world with the I/O world might also have a performance advantage, because TBB pipelines are not optimized for I/O heavy operations and perform badly when stalling on I/O. These input/output threads (those which run the do_work methods) might decrease the probability of a pipeline thread waiting for I/O by filling the pipestream buffers while working on a different CPU.

4. Evaluation

In this chapter, we demonstrate the effectiveness of the tokenizer with several tokenization schemes and on several datasets. In the first section, we study the accuracy of the tokenizer using different tokenization schemes. In the second section, we follow up with an analysis of the speed at which it processes data.

4.1 The Accuracy of the System

4.1.1 Chinese Word Segmentation

Tokenizing Latin-script languages is not very hard. We can usually get by well enough by splitting the text at whitespaces and at boundaries between different classes of symbols. Sometimes, we might want to be more specific and try to tokenize English contractions as separate words. However, these problems are quite easy to solve when compared to the task of tokenizing Chinese text. The absence of any spaces between words forbids the use of any simple heuristic and linguistically empowered methods must be used.

We took inspiration from the system for Chinese word segmentation presented in Section 1.6 [17] which is also based on maximum entropy models. The basic features used in that system were ported to our formalism. The biggest difference between the systems is the fact that the original Chinese tokenizer classified individual characters as being single-character words or the beginning, middle or ending characters of a multi-character word. However, the classifier used in our system is binary and it decides for each character boundary whether it forms a token boundary or not.

We were able to obtain the same data on which the original tokenizer was developed, which happen to be the training data for the Second International Chinese Word Segmentation Bakeoff [12]. The bakeoff was a competition challenging computational linguists to develop word segmentation systems for Chinese using the supplied data for training. The provided data consists of 4 datasets provided by Academia Sinica, City University of Hong Kong, Peking University and Microsoft Research. Each of these datasets adopts slightly different tokenization standards and so we train and test our tokenizer on the datasets individually. Each dataset comes with a training part and a testing part. We strictly used only the training part when developing our tokenizer and used the testing part only at the end to evaluate our results. The only thing we knew about the testing data in advance was its size which helped us choose a reasonable size for our heldout data.

	Training data		Testing data
	Development data	Heldout data	Testing data
Academia Sinica	39686533	1057344	942571
City University	8283422	266247	240767
Peking University	7008808	719430	718331
Microsoft Research	16100177	791333	766786

Table 4.1: The sizes of the individual parts of the bakeoff datasets in bytes.

	Number of iterations
Academia Sinica	420
City University	873
Peking University	708
Microsoft Research	1053

Table 4.2: The number of iterations spent training the maximum entropy model on the individual datasets.

First, we split our training data into a development part and a heldout part. We chose the size of the heldout data to be roughly as big as the testing data so we could trust our system's performance on it to be representative of our system's true accuracy. The sizes of the partitioned datasets can be seen in Table 4.1.

Initially, we set the event cutoff of the maximum entropy trainer to 2 as in [17]. However, we found out we get a sizable improvement in the accuracy of the trained tokenizer if we do not cutoff events (i.e. set the event cutoff to 1). We then experimented with training the tokenizer and testing it on the heldout data. Depending on how much we constrained training time, the tokenizer could either be under-trained or over-fitted. The heldout data served as an independent indicator telling us how close we are to the ideal balance between a detailed and a general model. Experimentation led us to restrain the number of training iterations to the values seen in Table 4.2 (the considerable size of the Academia Sinica combined with the absence of the event cutoff forced us to keep the number of training iterations below 450 lest the training program hit the CPU time limit and terminate). We can see that the number of iterations spent in training to obtain the optimal model correlates with the size of the dataset (with the exception of the Academia Sinica dataset, of course), because a larger dataset usually means more bigrams and unigrams and thus more parameters to estimate.

After we established the training parameters, we trained the system on the entire training data and checked its performance on the gold testing data. The performance of the development system on the heldout data and of the final system on the testing data can be seen in Tables 4.3 and 4.4.

	Accuracy	Precision	Recall	F-measure
Academia Sinica	97.56%	97.89%	97.82%	97.86%
City University	97.70%	98.05%	98.11%	98.08%
Peking University	97.69%	98.29%	97.89%	98.09%
Microsoft Research	97.67%	98.08%	98.02%	98.05%

Table 4.3: The performance of the system trained on the development data when tokenizing the heldout data.

	Accuracy	Precision	Recall	F-measure
Academia Sinica	96.33%	96.13%	97.73%	96.92%
City University	96.87%	97.42%	97.32%	97.37%
Peking University	96.74%	97.85%	96.68%	97.26%
Microsoft Research	97.95%	98.33%	98.06%	98.20%

Table 4.4: The performance of the system trained on the entire training data when tokenizing the gold testing data.

We were encouraged to see such performance and out of curiosity proceeded to score our tokenizer using the same script which scored the contestants in the bakeoff (Table 4.5). While our tokenizer does not perform as well as the original word segmenter by Low, Ng and Guo [17], it achieves a median performance compared to the performance of the other bakeoff submissions. The result is quite pleasing, given that the all we needed to do was to write the feature definitions into a few files and tweak some training parameters.

4.1.2 Tokenization of Czech and English

For evaluating the accuracy of tokenizing Czech and English text, four different methods were implemented. The Absolute Baseline relies on no other piece of information than the current decision point and the whitespace following it to classify boundaries. It is there to show the minimum possible line every tokenizer should pass.

The Simple Tokenizer checks the potential sentence terminator and checks

	True Words Recall	Test Words Precision	F-measure
Academia Sinica	0.933	0.919	0.926
City University	0.934	0.934	0.934
Peking University	0.923	0.933	0.928
Microsoft Research	0.951	0.952	0.951

Table 4.5: The scores assigned to our tokenizer by the official scoring script of the Second International Chinese Word Segmentation Bakeoff.

CzEng - Czech	Segmentation			
	Acc.	Prec.	Rec.	F-m.
Absolute Baseline	80.08%	72.72%	99.06%	83.87%
Simple Tokenizer	93.67%	92.38%	95.79%	94.06%
Groomed Tokenizer	95.93%	95.26%	96.90%	96.07%

Table 4.6: The sentence boundary disambiguation performance of the various methods for tokenizing Czech on the CzEng sample.

CzEng - Czech	Tokenization			
	Acc.	Prec.	Rec.	F-m.
Absolute Baseline	99.29%	99.29%	100.00%	99.64%
Simple Tokenizer	99.26%	99.35%	99.92%	99.63%
Groomed Tokenizer	99.36%	99.39%	99.97%	99.68%

Table 4.7: The token boundary disambiguation performance of the various methods for tokenizing Czech on the CzEng sample.

whether the following word starts with an upper-case letter. It represents the often too simple approach to tokenization.

The English-only Satz-like [21] system uses only part of speech data about the surrounding tokens to predict a boundary.

Finally, the Groomed Tokenizer is the tokenization scheme used in the original reference implementation, which has been supplied with lists of abbreviations and lots of useful regular expressions.

All systems were tested both on a sample of data from CzEng and, in the case of the English tests, also on the Brown corpus. All datasets were divided into equally large development, heldout and testing sets to be used as in Section 4.1.1. As for the part of speech data of the Satz-like system, lexicons for each part of speech were extracted from the training section of the Brown corpus for the Brown corpus exercise and from the entire Brown corpus for the CzEng exercise. The results of the trials can be seen in Tables 4.6, 4.7, 4.8, 4.9, 4.10 and 4.11.

CzEng - English	Segmentation			
	Acc.	Prec.	Rec.	F-m.
Absolute Baseline	81.27%	67.50%	99.91%	80.57%
Simple Tokenizer	95.21%	91.38%	96.81%	94.01%
Satz-like System	94.87%	92.42%	94.57%	93.48%
Groomed Tokenizer	97.08%	95.66%	96.90%	96.27%

Table 4.8: The sentence boundary disambiguation performance of the various methods for tokenizing English on the CzEng sample.

CzEng - English	Tokenization			
	Acc.	Prec.	Rec.	F-m.
Absolute Baseline	95.31%	95.31	100.00%	97.60%
Simple Tokenizer	95.27%	95.31%	99.95%	97.58%
Satz-like System	96.84%	96.79%	100.00%	98.37%
Groomed Tokenizer	95.99%	95.99%	99.98%	97.94%

Table 4.9: The token boundary disambiguation performance of the various methods for tokenizing English on the CzEng sample.

Brown	Segmentation			
	Acc.	Prec.	Rec.	F-m.
Absolute Baseline	78.49%	62.83%	99.61%	77.06%
Simple Tokenizer	96.47%	93.26%	97.30%	95.24%
Satz-like System	99.31%	99.58%	98.52%	99.05%
Groomed Tokenizer	99.31%	99.30%	98.80%	99.05%

Table 4.10: The sentence boundary disambiguation performance of the various methods for tokenizing English on the Brown corpus.

Brown	Tokenization			
	Acc.	Prec.	Rec.	F-m.
Absolute Baseline	82.71%	85.16%	88.74%	86.91%
Simple Tokenizer	93.63%	94.12%	96.16%	95.13%
Satz-like System	99.64%	99.62%	99.82%	99.72%
Groomed Tokenizer	99.73%	99.72%	99.86%	99.79%

Table 4.11: The token boundary disambiguation performance of the various methods for tokenizing English on the Brown corpus.

While text from the CzEng dataset proves to be more difficult to segment than text from the Brown dataset for all but the Baseline tokenizer, the Satz-like system's segmentation performance suffers the most. This was not unexpected as the Satz-like tokenizer relies on a lexicon of part of speech tags extracted from parts of the Brown corpus. When the tokenizer was evaluated on the Brown corpus, the lexicon was induced from the training and heldout datasets. This gave the tokenizer's lexicon a 99.41% coverage on the training dataset and a 99.44% coverage on the heldout dataset (the coverage is not 100% as some of the words containing dashes or apostrophes were broken into separate rough tokens); the coverage on the testing dataset was 95.75%. On the other hand, when the tokenizer was evaluated on the CzEng dataset, the coverage on the training, heldout and testing datasets was 95.80%, 95.95% and 95.66% respectively. Since the Satz-like tokenizer relies only on part of speech data, this decrease in the part of speech lexicon's coverage can be severely detrimental to the tokenizer's performance.

The Simple tokenizer demonstrates a pretty high recall on sentence boundary detection. This can be attributed to the fact that its decisions are governed only by the potential sentence boundary and the case of the following word. Since mostly every sentence will start with a capital letter, we can expect the Simple tokenizer to notice most of them. The Simple tokenizer can however be easily misled by multi-part abbreviations and initials in names (e.g. "U.S.A.", "M. Smith"). This explains why its precision is noticeably lower than its recall.

The Groomed tokenizer delivers a good performance on all the examined datasets, which goes to show that time spent developing a tokenization scheme can indeed pay off.

4.2 The Speed of the System

The tokenizer processes roughly 15000–60000 decision points per second, which amounts to 60000–250000 words or 300000–1300000 bytes in the case of the Brown corpus. The performance varies greatly depending on the chosen tokenization scheme and so in this section, we will do the performance analysis for each of the four tokenization schemes presented in Section 4.1.

4.2.1 Parallel Processing

One of the most important aspects of the tokenizer which drove the design was parallel processing. In Chapter 3, we have seen how it encouraged us to divide the tokenizer's duties to several autonomous subsystems. This design enabled us

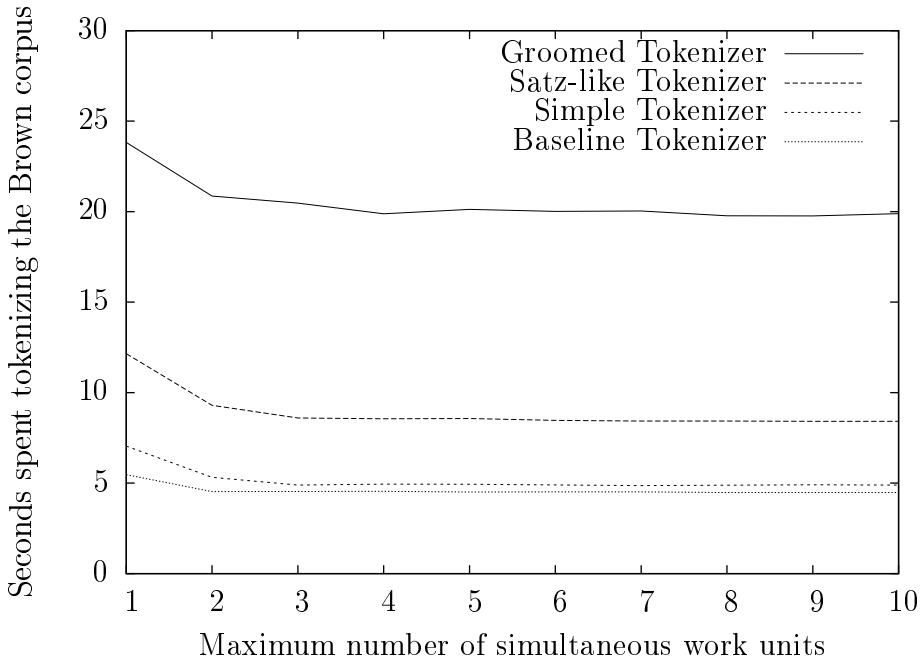


Figure 4.1: The effect of maximum simultaneous work units on the performance of the tokenizer. The plotted spent time is a median of 10 trials.

to perform all the tasks in the pipeline in parallel using the **pipeline** class from the Threading Building Blocks library [3]. To measure the impact this design choice made on performance, we ran the tokenizer on the entire Brown corpus while restricting the maximum number of pipeline stages allowed to run at the same time. The results are plotted in Figure 4.1. The Baseline and Groomed tokenizers speed up by 20%–24%, while the Simple and Satz-like tokenizers gain a speedup of 44%.

To investigate the reason why the different tokenization schemes gain a different speedup and where we should optimize further to improve the processing time, we measure the workload of the different pipeline stages. We restrict the maximum number of simultaneous work units in the pipeline to 1, to ensure that only one can use the CPU at a time. In each of the stages we measured the total time spent processing the stream of data. The averaged results can be seen in Table 4.12.

From the data, we can see that the workload is more balanced in the Simple and Satz-like tokenizers, while in the Baseline and Groomed tokenizers, the RoughTokenizer, resp. the Classifier, spend more time than all of the other stages combined. This means that when using the Baseline or Groomed tokenizer, one thread will be working in the RoughTokenizer, resp. the Classifier, leaving the other threads very little work to do, which leads to only a small speedup from the original scenario with one thread.

	Baseline	Simple	Satz-like	Groomed
RoughTokenizer	3.06	3.06	3.09	3.18
FeatureExtractor	0.28	0.67	2.49	5.13
Classifier	1.11	2.19	5.12	14.22
OutputFormatter	0.72	0.70	0.71	0.71

Table 4.12: Time (in seconds) spent in the various pipeline stages when tokenizing the Brown corpus. In order to measure these values, the pipeline has been set up to run only one stage at a time. The tabled time is an average of 10 trials.

	Baseline	Simple	Satz-like	Groomed
Width of context	1	2	7	17
Number of user-defined properties	0	1	37	32
Number of possible features per decision	6	13	311	673
Average number of features per decision	1.53	3.98	21.04	75.38

Table 4.13: The factors which define the computational complexity of the Classifier stage. The average number of features per decision was measured on the Brown corpus.

We can also see that the complexity of rough tokenization and output processing is the same with all the tokenization schemes, which was to be expected as there are next to no differences in these stages between the contesting tokenizers. The FeatureExtractor's workload scales with the number of regular expression properties and list properties as expected (the data also shows that the multiple list properties used for the part of speech lexicon in the Satz-like tokenizer are faster to check than the individual regular expressions used in the Groomed tokenizer).

The most important fact we can glean from the results, however, is that in the more complex tokenizers, the Classifier is the bottleneck. The Classifier is the subsystem responsible for checking the context surrounding each decision point, producing a list of strings describing the features of the rough tokens in the context and consulting the maximum entropy model for a disambiguation.

The distinguishing factors which define the computational complexity of the Classifier are listed in Table 4.13. The Classifier iterates over the rough tokens in the context. Each rough token is checked for the mandatory properties (whitespace between tokens, presence of decision points) and strings representing the features are created. The rough tokens at specific offsets are also checked for the user-defined properties and strings describing these features are generated as well. This vector of features is then deciphered by the maximum entropy toolkit. Each feature name is mapped to a factor and the factors are added up for each individual outcome. The outcome with the highest probability is then selected.

	Baseline	Simple	Satz-like	Groomed
Initialization	0.002	0.03	0.30	0.11
Processing	4.473	4.86	8.12	19.59
Total	4.475	4.88	8.42	19.70

Table 4.14: Time (in seconds) spent tokenizing the Brown corpus using the 4 tokenization schemes presented. Initialization is the time spent before the pipeline is run. The tabled time is an average of 10 trials.

The amount of work needed to handle the built-in mandatory probabilities is linear with respect to the width of the context.

The rest of the time is spent checking for the user-defined properties and generating feature strings (a string containing the offset, name and value of a feature). Assuming the names of user-defined properties are bound by some constant, the worst case time spent doing this is linear to the product of the context's width and the number of user-defined properties. However, at some of the offsets in the context, some of the properties might not be requested by the user or might simply not hold for the rough token in question. We let the tokenizers log the decision points and the features describing them and measured how many feature strings per decision are actually generated and processed by the maximum entropy library (the values are listed in Table 4.13). This factor is most indicative of the workload of the Classifier.

As the Classifier has been identified as a bottleneck of the pipeline, any attempts at optimizing the performance of the tokenizer should be performed there. The amount of time spent in the maximum entropy library is only 15%–23% of the entire time spent in the Classifier. Improving the string manipulation and feature representation thus seem to be sensible places to look at. In the case of the Groomed tokenizer, more speed could be gained by culling the number of features or narrowing the context.

The bottleneck issue might also be worked around by dividing the input and processing multiple segments at the same time. The Classifier would still be the bottleneck, but several instances of the pipeline (and therefore the Classifier) would run at the same time (e.g. on different paragraphs).

4.2.2 Initialization Costs

A necessary part of processing data with the tokenizer is the execution and initialization of the tokenizer itself. We were interested in how long the initialization takes in comparison to the processing of input. We measured the time spent in both these stages and listed our measurements in Table 4.14.

	Baseline	Simple	Satz-like	Groomed
Size of data	530	36500	269000	35500

Table 4.15: Volumes of data (in bytes) which take the same time to process using a given tokenization scheme as it takes to initialize the tokenization scheme.

It can be seen that when processing large quantities of data, the initialization costs are negligible. However, it is quite probable that the tokenizer will be used to process smaller files. For example, the entire Brown corpus has 6MB of data, but it is distributed as a set of files about 11KB small. To express the cost of initialization in more useful terms, we found the volume of data that the tokenizer can process within the amount of time spent to initialize it (Table 4.15). The data shows that when using the Simple or the Groomed tokenizer, it would take four times as long to process the Brown corpus if we were to initialize the tokenizer before processing each file. When using the Satz-like tokenizer or any other tokenization scheme based on large lexicons, the initialization costs are even bigger.

The expected cost in initialization time is mitigated by the ability to run the tokenizer on batches of files. The tokenizer can look for files to be processed in lists of file paths stored in files or passed through the standard input. The results are written to files whose paths are found by applying a user-specified regular expression replacement string on the original files' paths.

All the tokenization schemes presented in this chapter were trained and tested using this way of execution. If large volumes of small files are to be processed using the tokenizer, these batch facilities are essential as they make the daunting cost of initialization marginal (as in Table 4.14).

The file lists interface also presents another opportunity for further parallelization. If the input is already spread into small files, the files in the file lists could be processed simultaneously by different instances of the pipeline. Something like this is also quite easy to implement on the user's side, where the user can divide the files to be processed into a number of file lists appropriate for the number of CPUs and amount of memory available to him and execute the tokenizer on each of the file lists simultaneously.

Conclusion

We have presented a data-driven system for tokenizing and segmenting text. We have demonstrated the system's versatility by combining methods based on different techniques such as morphological dictionaries, regular expressions and exception lists. The system proved its universal applicability in being able to act both as a sentence boundary disambiguator for languages such as English and Czech and as a word segmenter for languages which do not use whitespace such as Chinese. We have also pointed to the fact that the program relies only on multi-platform programs and libraries. While it has not been tested on Windows or MacOS yet, care was taken at every step to ensure it would be a smooth transition (ICU can be used instead of libiconv for character code conversion, CMake is used for building, OS-specific matters are accessed via Boost only...).

We measured the accuracy, precision, recall and F-measure of the token and sentence boundary disambiguation. The tests were executed with several very different tokenization schemes and on several datasets in multiple languages. We also measured and analyzed the tokenizer's speed and identified the bottleneck which should serve as an avenue for further optimization.

The natural next step would be to invent and experiment with new ways and features for tokenizing and segmenting text. The system offers fast feedback on the accuracy of the user's tokenization schemes and is helpful in pointing out positions in the text which are yet to be covered by rules for inserting decision points. Another possible elaboration might be to change the maximum entropy training back-end to the Toolkit for Advanced Discriminative Modelling or some other alternative.

A. User Documentation

```
trtok - a fast and trainable tokenizer for natural languages
------------------------------------------------------------

  Trtok is a very universal performance-oriented tokenizer for processing
natural languages. It reads text and tries to correctly detect sentence
boundaries and divide the text into tokens.

  Trtok does not implement any specific heuristic to perform these tasks,
instead it lets the user define rules for potential joining and splitting of
words into tokens and sentences. The final decision whether to split or join
words and whether to break sentences is left to a conditional probabilistic
model which is trained from user-supplied annotated data. The way the trainer
understands the data can be extensively customized by the user who can define
his own features and specify which features are significant for what tokens.


1) Tokenization schemes
-----------------------

  The user might want to use trtok for processing more than 1 language or for
  processing 1 language in many ways. These different ways of tokenization are
  described by "tokenization schemes". Their definitions reside in the
  "schemes" subdirectory of the installation directory. Every folder inside
  "schemes" defines a single tokenization scheme by way of various
  configuration files.

  Tokenization schemes may be nested to represent a sort of scheme inheritance
  where a scheme inherits all the configuration files of its ancestors unless
  it redefines them by having a configuration file of the same name.

  a) Rough tokenization rules

    Files with a .split file extension must contain pairs of whitespace
    delimited regular expressions. When tokenizing the input, every position in
    every word where the prefix leading to the position matches the first
    expression and the suffix following it matches the second expression is
    marked as a potential token split and the word is split into two temporary
    rough tokens.

    Files with a .join extension have the same syntax, but they describe two
    spans of text which may be potentially joined into a single token if found
    separated by whitespace.

    Files with a .begin and a .end extension list characters before which, resp.
    after which, a potential sentence break is to be marked (and if this
    potential sentence break occurs within a word, the word is split into two
    rough tokens). If the newline character is to be a potential sentence
    beginning or ending character, an empty line is expected in either a .begin
    file or a .end file.

    The grammar of the regular expressions in .split and .join files is the one
    used by Quex and described in detail at
    http://quex.sourceforge.net/doc/html/usage/patterns/context-free.html.

    The .split and .join files may contain comments which are lines that begin
    with the # symbol.
```

b) User-defined properties

Files with a .rep extension contain a single regular expression from the family of expressions allowed in PCRE (see pcre.org). A rough token is marked as having this property if it can be matched to the regular expression.

Files with a .listp extension define properties using lists of token types. If a rough token's text is exactly the same as a line from a .listp file, then that rough token is marked as having the property defined by that .listp file.

c) Feature selection

Every tokenization scheme must have a file named "features". For each rough token in the vicinity of the potential split/join/sentence break, it specifies which features are important for the decision.

A typical line starts by declaring a set of interesting offsets (0 is the rough token preceding the decision point, -1 the one before it, +1 the one after it, etc...). These offsets are separated by commas and intervals can be used for convenience (e.g. -4,-2..+2,5 selects -4,-2,-1,0,1,2,5).

After the offsets comes a colon and a comma separated list of properties. The property names are the filenames of their definitions without the extensions and they are limited to the common identifier character set [a-zA-Z0-9_]. The line is closed with a terminating semicolon.

Apart from these simple features, it is possible to ask for combined features which bundle the value of different properties of tokens at different offsets into a single feature value. These are defined on their own line and are enclosed in parentheses. Inside the parentheses is a "^" separated list of offset:property pairs. If a combined feature takes properties from a single token only, the parenthesized expression can appear on the right-hand side of a typical line instead of a simple property name and the offsets within its definition are omitted.

Apart from the user-defined properties from the .rep and .listp files, the tokenizer defines the non-binary property "%length" whose value is the length of the rough tokenizer and the meta-property "%Word" which generates a property for each rough token type.

Example:

```
-2..+2: %Word;
-5..5: uppercase, abbreviation, (starts_with_number ^ ends_with_period);
(0:fullstop ^ 1:initial)
```

d) Maxent training parameters

More control over the process of training the probabilistic model can be had by manipulating the "maxent.params" file. This file is an INI-style configuration file which lets the user set the following parameters, which get passed directly to the training toolkit.

```
event_cutoff=<int>                 All training events which occur less
      times than event_cutoff are ignored. Default 1.
```

```
      n_iterations=<int>                How many iterations at most will the
            iterative method use. Default 15.

      method_name=lbfgs|gis             Which of the two methods L-BFGS or GIS
            is to be used. L-BFGS is recommended. Default lbfgs.

      smoothing_coefficient=<double>    Sigma, the coefficient in Gaussian
            smoothing. Default 0 (no smoothing).

      convergence_tolerance=<double>    The model is regarded as convergent
            when the relative difference between the log-likelihood of the
            succeeding models is < convergence_tolerance. Default 1e-05.

      save_as_binary=false|true         Whether to save the file in a binary
            format which is faster to load and smaller if Maxent was compiled
            with zlib support. Default false.

  e) File lists and filename replacement regular expressions

    Files [prepare|train|heldout|tokenize|evaluate].[fl|fnre] are for
    convenience only and are described later.


2) Running the tokenizer
------------------------

  a) Different ways of selecting input

    The first argument passed to the tokenizer selects its mode, which can be
    either "prepare", "train", "tokenize" or "evaluate". The second argument is
    a path relative to the directory "schemes" which selects the tokenization
    scheme to be used. The rest of the arguments are input files and options.

    Input files can be specified explicitly on the command line. More files can
    be given using the -l (--file-list) option which takes a path to a file and
    adds every line of it as another input file.

    When running in prepare mode or tokenize mode, an output file for each file
    has to be specified and when running in train mode or evaluate mode, a file
    with the annotated version has to be specified. These secondary files are
    selected by taking the input file's path and transforming it using a regular
    expression/replacement string. The filename regular expression/replacement
    string is specified using the -r (--filename-regexp) option. The strings
    look like replacement commands in sed, where the first character can be any
    ASCII character and that character separates the regular expression from
    the replacement string and also terminates the entire string. Unlike sed,
    this special character cannot be present anywhere else in the string (no
    escaping). The breed of regular expressions used here is the one supported
    by PCRE, the replacement strings contain the placeholders \0, \1... for the
    entire matched string, first captured sequence...

      Example:

        trtok train en/simple/brown -l data/brown/train.fl -r "|raw|txt|"

    If no input file or file lists were given, a default file list named
    <mode_name>.fl, which is part of the tokenization scheme, is used. If no
    filename regular expression/replacement string is given, the one in the
```

file named <mode_name>.fnre from the tokenization scheme is used. In both cases <mode_name> is expanded to either "prepare", "train", "tokenize" or "evaluate" depending on the current mode.

If no input file or file lists were given and there are no default file lists defined by the tokenization scheme, then the tokenizer processes the standard input and writes to the standard output. This is, however, only possible for the "prepare" and "tokenize" modes. The standard input/output combo can also be explicitly selected by specifying the input file "-" on the command line.

b) Different modes of execution

In "prepare" mode, the tokenizer reads the input, splits it into rough tokens and then outputs it with all possible splits and sentence breaks performed. This format might be handy for manual annotators who then only have to join together parts of tokens and sentences.

In "train" mode, the tokenizer reads both the input and its annotated version. It uses the annotated data to get pairs of questions (values of features in a given context surrounding a decision point) and answers (whether the decision point is to become a joining of tokens, a splitting of tokens or a sentence break). These pairs are then used to train the probabilistic model and store it in a file under the "build" directory.

In "tokenize" mode, the tokenizer relies on the presence of an already trained model and uses it to classify every decision point in the input file and output the tokenized and segmented text.

In "evaluate" mode, the tokenizer reads both the input and its annotation as in "train" mode, but now it also queries the trained model for an opinion and compares it with the one found in the annotated data. The tokenizer outputs a log of every context and both the predicted and correct outcomes for later analysis. The "analyze" script provided with trtok will let you read this output and determine the accuracy of your system.

c) Different options

If you launch trtok with no command line arguments, you will get a summary of all the supported command line options and their meaning. These include options for setting the encoding of the input and output files, options for controlling the output (preserving the original tokenization, segmentation or paragraph division), the preprocessing of input (if entities are to be expanded for the duration of the tokenization and if they are to be kept expanded in the output; if XML should be hidden from tokenization), options for logging the contexts and outcomes to a third file and others.

Bibliography

[1] *CMake - Cross Platform Make.*
URL `http://cmake.org/`

[2] *GNU Libtool - The GNU Portable Library Tool.*
URL `http://www.gnu.org/software/libtool/`

[3] *Intel® Threading Building Blocks 3.0 for Open Source.*
URL `http://threadingbuildingblocks.org/`

[4] *LaTeX WikiBook / Formatting.*
URL `http://en.wikibooks.org/wiki/LaTeX/Formatting`

[5] *The OpenMP API.*
URL `http://openmp.org/wp/`

[6] *PCRE - Perl Compatible Regular Expressions.*
URL `http://www.pcre.org/`

[7] *re2 - an efficient, principled regular expression library.*
URL `http://code.google.com/p/re2/`

[8] Berger, A.L., Pietra, V.J.D., and Pietra, S.A.D. *A maximum entropy approach to natural language processing.* Computational linguistics, 22(1):39–71, 1996.

[9] Bird, S., Loper, E., and Klein, E. *Natural Language Toolkit.*
URL `http://www.nltk.org/`

[10] Bojar, O. *Trainable Tokenizer v.0.1.*
URL `http://ufal.mff.cuni.cz/euromatrixplus/downloads.html`

[11] Dawes, B., Williams, A., Prus, V., Henney, K., Järvi, J., Dimov, P., and Abrahams, D. et al. *Boost C++ Libraries (filesystem, thread, program_ options, lexical_ cast, ref).*
URL `http://www.boost.org/`

[12] Emerson, T. *Second International Chinese Word Segmentation Bakeoff*, 2005.
URL `http://www.sighan.org/bakeoff2005/`

[13] Grefenstette, G. and Tapanainen, P. What is a Word, what is a Sentence?: Problems of Tokenisation. Citeseer, 1994.

[14] Kiss, T. and Strunk, J. *Unsupervised multilingual sentence boundary detection.* Computational Linguistics, 32(4):485–525, 2006.

[15] Klyueva, N. and Bojar, O. *UMC 0.1: Czech-Russian-English Multilingual Corpus.* In Proc. of International Conference Corpus Linguistics, pp. 188–195. 2008.
URL `http://ufal.mff.cuni.cz/euromatrixplus/downloads.html`

[16] Le, Zhang. *Maximum entropy modeling toolkit for Python and C++.* 2004.
URL `http://homepages.inf.ed.ac.uk/lzhang10/maxent_toolkit.html`

[17] Low, J.K., Ng, H.T., and Guo, W. *A maximum entropy approach to Chinese word segmentation.* In Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing, vol. 1612164. Jeju Island, Korea, 2005.

[18] Malouf, R. *A comparison of algorithms for maximum entropy parameter estimation.* In proceedings of the 6th conference on Natural language learning-Volume 20, pp. 1–7. Association for Computational Linguistics, 2002.

[19] Malouf, R., Baldridge, J., and Osborne, M. *The Toolkit for Advanced Discriminative Modeling*, 2005.
URL `http://tadm.sourceforge.net/`

[20] Nasonov, A. *"Any interest in io-stream like pipe?" An implementation of of pipestreams*, 2003.
URL `http://lists.boost.org/Archives/boost/2003/08/51289.php`

[21] Palmer, D.D. and Hearst, M.A. *Adaptive multilingual sentence boundary disambiguation.* Computational Linguistics, 23(2):241–267, 1997.

[22] Rafferty, A., Kleeman, A., Finkel, J., and Manning, C. *Stanford Classifier*, 2007.
URL `http://nlp.stanford.edu/downloads/classifier.shtml`

[23] Ratnaparkhi, A. *A simple introduction to maximum entropy models for natural language processing.* IRCS Technical Reports Series, p. 81, 1997.

[24] Reynar, J.C. and Ratnaparkhi, A. *A maximum entropy approach to identifying sentence boundaries.* In Proceedings of the fifth conference on Applied natural language processing, pp. 16–19. Association for Computational Linguistics, 1997.

[25] Riley, M.D. *Some applications of tree-based modelling to speech and language.* In Proceedings of the workshop on Speech and Natural Language, pp. 339–352. Association for Computational Linguistics, 1989.

[26] Schäfer, F.-R. *Quex - Fast Universal Lexical Analyzer Generator.* URL `http://quex.sourceforge.net/`

List of Tables