WERTi - Easing Second Language Acquisition

Aleksandar Dimitrov

June 27, 2008

Contents

1 Introduction

Using tools and methods made available through new achievements in computational linguistics and related subjects like cognitive science to ease the process of second language acquisition has only recently gained focus in research projects. While adaptions of traditional data sources like dictionaries have been used for quite a while we have yet to discover new and effective ways of

WERTi tries to approach this probmlem from a more general perspective. Making use of the the momentum of the Internet, WERTi provides a platform for implementing linguistic analysis and subsequent input enhancement methods on user specified pages on the World Wide Web. Using Java Servlets technology for serving content and the UIMA framework for processing it in a dynamic and flexible session, we aim to provide a platform for linguistic processing of online content that can go beyond input enhancement.

2 The Development Process

In this section we will first explain the goals of the reimplementation and then display in what environment the project has been written and what technologies have been put to use.

2.1 Goals of the New Ipmlementation

While the original implementation was written in the programming language Python, our new design is rooted in Java technologies and makes use of several frameworks to ensure maximum scalability.

The new system was written with several aspects in mind:

- While the original system was restricted to a few hand picked web sites and in fact only supporting input from one particular news site[1], the new system should be able to support almost arbitrary input from sites in the World Wide Web. For this the system has to perform reliable and robust evaluation of the site content in order filter out text for later natural language processing tasks. This way we can provide the user with free choice over the target material

[1] This site was `http://www.reuters.com`

- Processing of site content should be generalized and made to be as flexible as possible in order to ensure maximum extensibility of the system. This also implied splitting up different parts of natural language processing tasks on the input into several interdependent steps. This way, the results of one of the *Analysis Engines* can always serve as the input to other engines.

- Ensure asynchronous client $\leftrightarrow$ server communication capabilities in order to allow evaluation of the user's performance on target texts. Together with an anticipated user account system this would provide us with the ability of measuring of a particular user's progress and provide them with automated feedback on their abilities.

- Overall we wanted to provide an easy to use, flexible and scalable web based platform for methods of second language acquisition assistance.

2.2 Concept & Design

The original design of WERTi has been developed at Ohio State University by Detmar Meurers and his research associates. There exist several papers about this original work.

At Tübingen University the concept has been extended to encompass a wider range of functionality and provide a more scalable solution. To achieve this several enterprise grade technologies have been put to use. Although this comes with certain drawbacks (such as an increase in the codebase by as much as more than 1000%), they provided us with a more flexible and robust architecture which should be able to cope with most demands to the system.

2.3 Design Process

The work on the Java implementation of WERTi happened mostly autonomous and was conducted by one person. Given free choice over the development environment and frameworks, a considerable part of the writing process was spent on evaluation and study of the technologies put to use. The only notable specifications set on the development environment was the use of UIMA, which in turn implied the use of the Java programming language.

Development happened in a very productive atmosphere with mostly weekly project meetings between the programmer and the project supervisor where core functionality and the design of the analysis process were discussed. During the course of the design process, the system was several times restructured and some parts of it have by now been rewritten two or even three times.

Work on the code has been mostly performed using standard UNIX command line tools for writing, testing and debugging code, although more advanced solutions like the Eclipse Java IDE were also used. They mostly provided easier organization of the work within the different frameworks[2]. All of the work has been tracked with a version control[3] system on which further work on the system will also depend.

3 The Architecture of the System

This section will explain the principles underlying WERTi's functionality by first looking at the data processing architecture and then showing how it is integrated into the user interface of the web application.

3.1 The UIMA Analysis Engines

All text extraction and natural language processing work is done inside the UIMA architecture. Modifications to the document's structure are clearly seperated from the process of annotating it. In this way we can ensure natural language analysis to be a consistent process on one and the same input.

The NLP routines only add annotations to the document, and they are not supposed to change its state in any other way. Enriching the content of the web site is then left to an outside module that processes only the annotations and does not look at the document text itself. To achieve this degree of encapsulation between the different tasks, the system has been split into three main parts, operating independently:

[2]Most importantly UIMA which comes with a number of useful plugins to easily devise analysis engine outlines.

[3]The author chose to use the *git* version control system, which has been and developed for and successfully used in the context of much greater codebases, such as the Linux kernel.

- HTML processing:

  During initial processing of the input text, which at this stage consists of the raw web site content retrieved by the server, HTML tag annotations are made to distinguish HTML tags as non-natural language text. Then another module finds "relevant" text within the text surrounded by tags. This lays ground to later linguistic analysis by setting the margins of which parts of the document it has to operate on.

- Linguistic processing:

  All linguistic tasks (tokenization, sentence boundary detection, part-of-speech tagging...) on the text-annotations from the previous processing step are performed in this module. This is also the most expensive step from a computational point of view. Optimizations to the code are most likely to yield visible results here.

- Post processing:

  Post processing analysis provides annotations on the document text with regard to the enhancement method the user inquired.

All steps operate solely on the CAS, UIMA's native document model. This is also the most strenous requirement on analysis engines to be integrated into WERTi. While external configuration may be read, there should be absolutely no side effects outisde the CAS - which is the only stateful entitity during linguistic processing.

The next sections will explain all subsequent analysis steps in further detail.

3.2 HTML Processing

The HTML processor method was designed to be primitive and efficient. While using a fully capable HTML parser was considered, we preferred to use a more simple approach. Full and formally correct HTML markup was deemed unnecessary and too time intensive. Furthermore, changing the implementation of an analysis step even this fundamental to further processing should be easy and without side-effects as long as the requirements to preconditions and postconditions are met.

Preconditions An input document retrieved in during earlier steps has been retrieved and it exists in memory as an instance of a singleton String[4] and is stored in the UIMA CAS. This is then passed to the `process(CAS)`-method of the `HTMLAnnotator`-class.

Postconditions The document contains annotations marking up the positions and spans of html tags in the document text. The names of tags[5] are also stored and a `isClosing`-flag is set, that denotes whether this tag is closing another sibling[6].

3.3 Finding Text to Process

The next step in the process is to find a way of denoting the text areas that will lay the basis for later linguistic processing. Originally this part of the analysis engine was far more productive than it is now. It has lost a lot of its functionality which has been taken over by the linguistic processing itself.

Currently, its main task is to eliminate whitespace and text between tag pairs considered irrelevant, mostly because they contain scripts and meta-information not actually rendered to text by the user's client.

This functionality could be further extended by providing hints and special caases for certain recommended web sites, such as Wikipedia or various news sites. However, since the source layout of most web pages is highly volatile, development focus has so far not turned to evaluation of this idea.

Preconditions The CAS contains full HTML tag annotations.

Postconditions The CAS contains a markup of all text that is going to be considered by the linguistic processing.

[4]As usual in UIMA. For large documents, UMIA provides ways of splitting up documents and processing the chunks independently. However, UIMA only considers documents well beyond one megabyte to be "big enough" to be split. The plain HTML most web pages serve rarely exceeds this mark.

[5]E.g. "p" for the tag `<p>` or "¡div¿" for the dag `div`.

[6]A preceding slash in the tag name closes the tag.

3.4 Linguistic Processing

Linguistic processing currently goes through 3 major steps: *Tokenization*, *Sentence Boundary Detection* and *Part of Speech Tagging*. This steps subsequently depend on each other.

3.4.1 Tokenization

Tokenization chunks the input from the previous processing steps into tokens of natural language. Different tokenizers may perform differently and consider different type of input spans to denote "tokens". This is especially important since later steps may depend on a particular type of tokenization rules[7]. Several types of Tokenizer have been implemented, with the current alternatives being an interface to the Stanford Tagger's tokenizer[8] and a simple example tokenizer, implemented locally for testing purposes, which seems to perform similar in terms of quality, but better in terms of performance and is the current default.

Precondition The CAS contains annotations that denote input data to the linguistic processing.

Postcondition The CAS contains a set of tokens that can lay the ground to all further linguistic analysis.

3.4.2 Sentence Boundary Detection

The sentence boundary detector implemented is currently not very advanced and simply matches several regular expressions and even single characters considered to end a sentence in all cases (`.`, `?` and `!`). This coud be improved by making more educated assumptions about the nature of the input tokens, but development has not focussed on these issues so far. While part of speech tagging does generally benefit from stringent denotations of sentences, the tagging process has so far been accurate enough and providing a correct method of sentence boundary detection could prove non-trivial to implement.

[7] Part of speech tagging is particularly sensivite to tokenization. Penn Tree Bank trained taggers usually require tokenization to happen according to the PTB tokenization guidelines. Currently, WERTi's tokenizers respect this guidelines, to provide a most general rule processin can rely upon

[8] Which in turn rips off something else

This could also be an entry point for future projects to improve the system's functionality, as accurate sentece boundary detection is of great importance to syntactic parsing.

Precondition Annotations in the CAS exist for all relevant input tokens in natural language.

Postcondition The CAS contains a markup of sentence boundaries. This markup only depends on a starting and an ending point within the documnet text, possibly spanning tags[9].

3.4.3 Part of Speech Tagging

Part of Speech tagging currently relies solely on external tools. Two taggers have so far been implemented: The *Penn Tree Bank Tagger* and the *UIMA Sandbox Tagger*. A Java *Interface* in the analysis package provides a common abstraction mechanism over the different taggers to be implemented. The `Tagger` interface declares processing methods as `synchronized`, so calls to the tagging routines are blocking. This ensures multiple clients running on the same server will not hinder the tagger in processing each call correctly[10].

Taggers are stored statically in server side context to ensure maximum performance as tagger instantiation is typically very costly. Most taggers are stateless during tagging, ensuring equal quality of results among calls.

Precondition The tagging process feeds on two types of annotations: `Token`s and `Sentence`s. It has access to the token annotations via calls to the sentence annotation's subiterator.

Postcondition All `Token` annotations the tagger found tags for now carry a "tag" field, indicating their part of speech tag.

[9]UIMA provides a `subiterator` method to construct an iterator over annotations of a particular type that are subsumed by another annotation of arbitrary type. It does not provide a general mechanism for implementing distributed annotations that would have multiple beginning and ending points.

[10]No tagger of those evaluated for useage provides concurrent processing of input strings since tagging is generally deemed to be an expensive step the machine should focus on.

3.4.4 Post Processing - Input Enhancement

This step depends on annotation results from the two preceeding modules; certain HTML markups are used in order to correctly organize all code later executed on client side.

3.5 The User Interface

At the time of writing the user interface to the WERTi platform has not been finished yet. As such, this section mostly provides a prespective on desired functionality, indicating partial results when they are already implemented.

3.5.1 The Interactive Web Interface

WERTi is now a web application, written in the Google Web Toolkit, which compiles native AJAX code from Java sources. The toolkit thus provided a possibility of increasing the consistnency of the platform's code by ensuring that it would be written using only one programming laguage. The user interface components rely on RPCs to interact with the server. RPC[11] provides an asynchronous method of interaction between client server side code, so the user can be informed about the progress of their request and also interact seamlessly with generated enhancements. While basic proof of concept for the user interface is already finished, more components will be implemented shortly.

- Users should be provided with an account system and components for evaluating their own progress in certain parts. For this, a basic database interface has been written, connected to the PostgreSQL engine. Calls to the database will be implemented in an asynchronous fashion, making use of non-blocking threading on the server side.

- Input enhancement on the retrieved document should make more use of its potential brought by the underlying framework. The chosen methods would allow for interactive suggestions and on line feedback, as well as the implementation of interesting new features possibly relying on client/server interaction. Partial translation of the document text or only dictionary lookups would be one such method.

[11] Remote Procedure Calls

The main reason for the user interface to be in a usable, but yet to be finished state is that development focus laid on making the back end reliable and stable enough to deal with user requests first. With a solid base provided, the user interface can now make use of the functionality described earlier in section 3.1.

4 Conclusion

Writing a system of the complexity of WERTi has proven to be a great and very enjoyable challange. Using corporate-grade environments, managing a large and constantly growing code base and deploying an interactive web application on web servers using very modern technologies have provided the author with great insights to the development of larger scale projects. While background in computational linguistics was neccessary in order to make the right choices in the design of the natural language processing tasks, working on the project also required flexibility terms of software engeneering and design, as well as knowledge of Internet technology and the principles underlying the World Wide Web.

The system itself has grown over the development process, which started on May 8th, 2008 and is currently continuing. Although it has made significant progress and achieved most of its original design goals, the next section will discuss some enhancements to the system required in order to bring full usability to WERTi.

4.1 Loose Ends

WERTi is by now in a mostly usable state. As such the programming project was a success, although there still remain some loos ends to be implemented. The following enumerates those and discusses possible takes on solving them.

- Provide Easier Access to Users

  The user interface is largely unwritten and lacks an interface to a web search engine such as Google or Yahoo Web Search, in order to retrieve content relevant to topics the user specifies.

  User accounts and relational databases with user data could be used to track a user's success and provide them with positive and negative feedback. E.g. showing in which context they regularly fail to give the

right preposition or determiner, or where they improved their score over time. RPC calls should provide sufficient client $\leftrightarrow$ server interaction potential.

- Provide a Greater Range of Features

  The analysis engines could also feature lemmatization, shallow parsing, (partial) translation and similar mechanisms for providing further methods of second language acquisition assistance.

Work on the system will continue in an open fashion and additional developers are encouraged to provide the author with their own ideas for implementation and code contributions to the system. This paper can serve as an entry point to understand the system at a more abstract level while the additional documentation integrated in the code base should provide developers with easy access to the system's inner workings.