Blueprint for a High Performance NLP Infrastructure

James R. Curran
School of Informatics
University of Edinburgh
2 Buccleuch Place, Edinburgh. EH8 9LW
jamesc@cogsci.ed.ac.uk

Abstract

Natural Language Processing (NLP) system developers face a number of new challenges. Interest is increasing for real-world systems that use NLP tools and techniques. The quantity of text now available for training and processing is increasing dramatically. Also, the range of languages and tasks being researched continues to grow rapidly. Thus it is an ideal time to consider the development of new experimental frameworks. We describe the requirements, initial design and exploratory implementation of a high performance NLP infrastructure.

1 Introduction

Practical interest in NLP has grown dramatically in recent years. Accuracy on fundamental tasks, such as part of speech (POS) tagging, named entity recognition, and broad-coverage parsing continues to increase. We can now construct systems that address complex real-world problems such as information extraction and question answering. At the same time, progress in speech recognition and text-to-speech technology has made complete spoken dialogue systems feasible. Developing these complex NLP systems involves composing many different NLP tools. Unfortunately, this is difficult because many implementations have not been designed as components and only recently has input/output standardisation been considered. Finally, these tools can be difficult to customise and tune for a particular task.

NLP is experiencing an explosion in the quantity of electronic text available. Some of this new data will be manually annotated. For example, 10 million words of the American National Corpus (Ide et al., 2002) will have manually corrected POS tags, a tenfold increase over the Penn Treebank (Marcus et al., 1993), currently used for training POS taggers. This will require more efficient learning algorithms and implementations.

However, the greatest increase is in the amount of raw text available to be processed, e.g. the English Gigaword Corpus (Linguistic Data Consortium, 2003). Recent work (Banko and Brill, 2001; Curran and Moens, 2002) has suggested that some tasks will benefit from using significantly more data. Also, many potential applications of NLP will involve processing very large text databases. For instance, biomedical text-mining involves extracting information from the vast body of biological and medical literature; and search engines may eventually apply NLP techniques to the whole web. Other potential applications must process text online or in real-time. For example, Google currently answers 250 million queries per day, thus processing time must be minimised. Clearly, efficient NLP components will need to be developed. At the same time, state-of-the-art performance will be needed for these systems to be of practical use.

Finally, NLP is growing in terms of the number of tasks, methods and languages being researched. Although many problems share algorithms and data structures there is a tendency to reinvent the wheel.

Software engineering research on Generative Programming (Czarnecki and Eisenecker, 2000) attempts to solve these problems by focusing on the development of configurable elementary components and knowledge to combine these components into complete systems. Our infrastructure for NLP will provide *high performance*[1] components inspired by Generative Programming principles.

This paper reviews existing NLP systems and discusses the requirements for an NLP infrastructure. We then describe our overall design and exploratory implementation. We conclude with a discussion of programming interfaces for the infrastructure including a script language and GUI interfaces, and web services for distributed NLP system development. We seek feedback on the overall design and implementation of our proposed infrastructure and to promote discussion about software engineering best practice in NLP.

[1] We use high performance to refer to both state of the art performance and high runtime effi ciency.

2 Existing Systems

There are a number of generalised NLP systems in the literature. Many provide graphical user interfaces (GUI) for manual annotation (e.g. General Architecture for Text Engineering (GATE) (Cunningham et al., 1997) and the Alembic Workbench (Day et al., 1997)) as well as NLP tools and resources that can be manipulated from the GUI. For instance, GATE currently provides a POS tagger, named entity recogniser and gazetteer and ontology editors (Cunningham et al., 2002). GATE goes beyond earlier systems by using a component-based infrastructure (Cunningham, 2000) which the GUI is built on top of. This allows components to be highly configurable and simplifies the addition of new components to the system.

A number of stand-alone tools have also been developed. For example, the suite of LT tools (Mikheev et al., 1999; Grover et al., 2000) perform tokenization, tagging and chunking on XML marked-up text directly. These tools also store their configuration state, e.g. the transduction rules used in LT CHUNK, in XML configuration files. This gives a greater flexibility but the tradeoff is that these tools can run very slowly. Other tools have been designed around particular techniques, such as finite state machines (Karttunen et al., 1997; Mohri et al., 1998). However, the source code for these tools is not freely available, so they cannot be extended.

Efficiency has not been a focus for NLP research in general. However, it will be increasingly important as techniques become more complex and corpus sizes grow. An example of this is the estimation of maximum entropy models, from simple iterative estimation algorithms used by Ratnaparkhi (1998) that converge very slowly, to complex techniques from the optimisation literature that converge much more rapidly (Malouf, 2002). Other attempts to address efficiency include the fast Transformation Based Learning (TBL) Toolkit (Ngai and Florian, 2001) which dramatically speeds up training TBL systems, and the translation of TBL rules into finite state machines for very fast tagging (Roche and Schabes, 1997). The TNT POS tagger (Brants, 2000) has also been designed to train and run very quickly, tagging between 30,000 and 60,000 words per second.

The Weka package (Witten and Frank, 1999) provides a common framework for several existing machine learning methods including decision trees and support vector machines. This library has been very popular because it allows researchers to experiment with different methods without having to modify code or reformat data.

Finally, the Natural Language Toolkit (NLTK) is a package of NLP components implemented in Python (Loper and Bird, 2002). Python scripting is extremely simple to learn, read and write, and so using the existing components and designing new components is simple.

3 Performance Requirements

As discussed earlier, there are two main requirements of the system that are covered by “high performance”: speed and state of the art accuracy. Efficiency is required both in training and processing. Efficient training is required because the amount of data available for training will increase significantly. Also, advanced methods often require many training iterations, for example active learning (Dagan and Engelson, 1995) and co-training (Blum and Mitchell, 1998). Processing text needs to be extremely efficient since many new applications will require very large quantities of text to be processed or many smaller quantities of text to be processed very quickly.

State of the art accuracy is also important, particularly on complex systems since the error is accumulated from each component in the system. There is a speed/accuracy tradeoff that is rarely addressed in the literature. For instance, reducing the beam search width used for tagging can increase the speed without significantly reducing accuracy. Finally, the most accurate systems are often very computationally intensive so a tradeoff may need to be made here. For example, the state of the art POS tagger is an ensemble of individual taggers (van Halteren et al., 2001), each of which must process the text separately. Sophisticated modelling may also give improved accuracy at the cost of training and processing time.

The space efficiency of the components is important since complex NLP systems will require many different NLP components to be executing at the same time. Also, language processors many eventually be implemented for relatively low-specification devices such as PDAs. This means that special attention will need to be paid to the data-structures used in the component implementation. The infrastructure should allow most data to be stored on disk (as a configuration option since we must tradeoff speed for space). Accuracy, speed and compactness are the main execution goals. These goals are achieved by implementing the infrastructure in C/C++, and profiling and optimising the algorithms and data-structures used.

4 Design Requirements

The remaining requirements relate to the overall and component level design of the system. Following the Generative Programming paradigm, the individual components of the system must be elementary and highly configurable. This ensures minimal redundancy between components and makes them easier to understand, implement, test and debug. It also ensures components are maximally composable and extensible. This is particularly important in NLP because of the high redundancy across tasks and approaches.

Machine learning methods should be interchangeable: Transformation-based learning (TBL) (Brill, 1993) and

Memory-based learning (MBL) (Daelemans et al., 2002) have been applied to many different problems, so a single interchangeable component should be used to represent each method. We will base these components on the design of Weka (Witten and Frank, 1999).

Representations should be reusable: for example, named entity classification can be considered as a sequence tagging task or a bag-of-words text classification task. The same beam-search sequence tagging component should be able to be used for POS tagging, chunking and named entity classification. Feature extraction components should be reusable since many NLP components share features, for instance, most sequence taggers use the previously assigned tags. We will use an object-oriented hierarchy of methods, representations and features to allow components to be easily interchanged. This hierarchy will be developed by analysing the range of methods, representations and features in the literature.

High levels of configurability are also very important. Firstly, without high levels of configurability, new systems are not easy to construct by composing existing components, so reinventing the wheel becomes inevitable. Secondly, different languages and tasks show a very wide variation in the methods, representations, and features that are most successful. For instance, a truly multilingual tagger should be able to tag a sequence from left to right or right to left. Finally, this flexibility will allow for research into new tasks and languages to be undertaken with minimal coding.

Ease of use is a very important criteria for an infrastructure and high quality documentation and examples are necessary to make sense of the vast array of components in the system. Preconfigured standard components (e.g. an English POS tagger) will be supplied with the infrastructure. More importantly, a Python scripting language interface and a graphical user interface will be built on top of the infrastructure. This will allow components to be configured and composed without expertise in C++. The user interface will generate code to produce standalone components in C++ or Python. Since the Python components will not need to be compiled, they can be distributed immediately.

One common difficulty with working on text is the range of file formats and encodings that text can be stored in. The infrastructure will provide components to read/write files in many of these formats including HTML files, text files of varying standard formats, email folders, Postscript, Portable Document Format, Rich Text Format and Microsoft Word files. The infrastructure will also read XML and SGML marked-up files, with and without DTDs and XML Schemas, and provide an XPath/XSLT query interface to select particular subtrees for processing. All of these reading/writing components will use existing open source software. It will also eventually provide components to manipulate groups of files: such as iterate through directories, crawl web pages, get files from ftp, extract files from zip and tar archives. The system will provide full support to standard character sets (e.g. Unicode) and encodings (e.g. UTF-8 and UTF-16).

Finally, the infrastructure will provide standard implementations, feature sets and configuration options which means that if the configuration of the components is published, it will be possible for anyone to reproduce published results. This is important because there are many small design decisions that can contribute to the accuracy of a system that are not typically reported in the literature.

5 Components Groups

When completed the infrastructure will provide highly configurable components grouped into these broad areas:

file processing reading from directories, archives, compressed files, sockets, HTTP and newsgroups;

text processing reading/writing marked-up corpora, HTML, emails, standard document formats and text file formats used to represent annotated corpora.

lexical processing tokenization, word segmentation and morphological analysis;

feature extraction extracting lexical and annotation features from the current context in sequences, bag of words from segments of text

data-structures and algorithms efficient lexical representations, lexicons, tagsets and statistics; Viterbi, beam-search and n-best sequence taggers, parsing algorithms;

machine learning methods statistical models: Naïve Bayes, Maximum Entropy, Conditional Random Fields; and other methods: Decision Trees and Lists, TBL and MBL;

resources APIs to WordNet (Fellbaum, 1998), Google and other lexical resources such as gazetteers, ontologies and machine readable dictionaries;

existing tools integrating existing open source components and providing interfaces to existing tools that are only distributed as executables.

6 Implementation

The infrastructure will be implemented in C/C++. Templates will be used heavily to provide generality without significantly impacting on efficiency. However, because templates are a static facility we will also provide dynamic versions (using inheritance), which will be slower but accessible from scripting languages and user interfaces. To provide the required configurability in the static version of the code we will use policy templates (Alexandrescu, 2001), and for the dynamic version we will use configuration classes.

A key aspect of increasing the efficiency of the system will be using a common text and annotation representation throughout the infrastructure. This means that we do not need to save data to disk, and load it back into memory between each step in the process, which will provide a significant performance increase. Further, we can use techniques for making string matching and other text processing very fast such as making only one copy of each lexical item or annotation in memory. We can also load a lexicon into memory that is shared between all of the components, reducing the memory use.

The implementation has been inspired by experience in extracting information from very large corpora (Curran and Moens, 2002) and performing experiments on maximum entropy sequence tagging (Curran and Clark, 2003; Clark et al., 2003). We have already implemented a POS tagger, chunker, CCG supertagger and named entity recogniser using the infrastructure. These tools currently train in less than 10 minutes on the standard training materials and tag faster than TNT, the fastest existing POS tagger. These tools use a highly optimised GIS implementation and provide sophisticated Gaussian smoothing (Chen and Rosenfeld, 1999). We expect even faster training times when we move to conjugate gradient methods.

The next step of the process will be to add different statistical models and machine learning methods. We first plan to add a simple Naïve Bayes model to the system. This will allow us to factor out the maximum entropy specific parts of the system and produce a general component for statistical modelling. We will then implement other machine learning methods and tasks.

7 Interfaces

Although C++ is extremely efficient, it is not suitable for rapidly gluing components together to form new tools. To overcome this problem we have implemented an interface to the infrastructure in the Python scripting language. Python has a number of advantages over other options, such as Java and Perl. Python is very easy to learn, read and write, and allows commands to be entered interactively into the interpreter, making it ideal for experimentation. It has already been used to implement a framework for teaching NLP (Loper and Bird, 2002).

Using the Boost.Python C++ library (Abrahams, 2003), it is possible to reflect most of the components directly into Python with a minimal amount of coding. The Boost.Python library also allows the C++ code to access new classes written in Python that are derived from the C++ classes. This means that new and extended components can be written in Python (although they will be considerably slower). The Python interface allows the components to be dynamically composed, configured and extended in any operating system environment without the need for a compiler. Finally, since Python can produce stand-alone executables directly, it will be possible to create distributable code that does not require the entire infrastructure or Python interpreter to be installed.

The basic Python reflection has already been implemented and used for large scale experiments with POS tagging, using pyMPI (a message passing interface library for Python) to coordinate experiments across a cluster of over 100 machines (Curran and Clark, 2003; Clark et al., 2003). An example of using the Python tagger interface is shown in Figure 1.

On top of the Python interface we plan to implement a GUI interface for composing and configuring components. This will be implemented in wxPython which is a platform independent GUI library that uses the native windowing environment under Windows, MacOS and most versions of Unix. The wxPython interface will generate C++ and Python code that composes and configures the components. Using the infrastructure, Python and wxPython it will be possible to generate new GUI applications that use NLP technology.

Because C++ compilers are now fairly standards compliant, and Python and wxPython are available for most architectures, the infrastructure will be highly portable. Further, we eventually plan to implement interfaces to other languages (in particular Java using the Java Native Interface (JNI) and Perl using the XS interface).

8 Web services

The final interface we intend to implement is a collection of *web services* for NLP. A web service provides a remote procedure that can be called using XML based encodings (XMLRPC or SOAP) of function names, arguments and results transmitted via internet protocols such as HTTP. Systems can automatically discover and communicate with web services that provide the functionality they require by querying databases of standardised descriptions of services with WSDL and UDDI. This standardisation of remote procedures is very exciting from a software engineering viewpoint since it allows systems to be totally distributed. There have already been several attempts to develop distributed NLP systems for dialogue systems (Bayer et al., 2001) and speech recognition (Hacioglu and Pellom, 2003). Web services will allow components developed by different researchers in different locations to be composed to build larger systems.

Because web services are of great commercial interest they are already being supported strongly by many programming languages. For instance, web services can be accessed with very little code in Java, Python, Perl, C, C++ and Prolog. This allows us to provide NLP services to many systems that we could not otherwise support using a single interface definition. Since the service arguments and results are primarily text and XML, the web service interface will be reasonably efficient for small

```
% python
Python 2.2.1 (#1, Sep 30 2002, 20:13:03)
[GCC 2.96 20000731 (Red Hat Linux 7.3 2.96-110)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import nlp.tagger
>>> op = nlp.tagger.Options('models/pos/options')
>>> print op
nklasses = 46
...
alpha = 1.65
>>> tagger = nlp.tagger.Tagger(op)
>>> tags = tagger.tag(['The', 'cat', 'sat', 'on', 'the', 'mat', '.'])
>>> print tags
['DT', 'NN', 'VBD', 'IN', 'DT', 'NN', '.']
>>> tagger.tag('infile', 'outfile')
>>>
```

Figure 1: Calling the POS tagger interactively from the Python interpreter

quantities of text (e.g. a single document). The second advantage they have is that there is no startup costs when tagger loads up, which means local copies of the web service could be run to reduce tagging latency. Finally, web services will allow developers of resources such as gazetteers to provide the most up to date resources each time their functionality is required.

We are currently in the process of implementing a POS tagging web service using the gSOAP library, which will translate our C infrastructure binding into web service wrapper code and produce the necessary XML service description files.

9 Conclusion

The Generative Programming approach to NLP infrastructure development will allow tools such as sentence boundary detectors, POS taggers, chunkers and named entity recognisers to be rapidly composed from many elemental components. For instance, implementing an efficient version of the MXPOST POS tagger (Ratnaparkhi, 1996) will simply involve composing and configuring the appropriate text file reading component, with the sequential tagging component, the collection of feature extraction components and the maximum entropy model component.

The individual components will provide state of the art accuracy and be highly optimised for both time and space efficiency. A key design feature of this infrastructure is that components share a common representation for text and annotations so there is no time spent reading/writing formatted data (e.g. XML) between stages.

To make the composition and configuration process easier we have implemented a Python scripting interface, which means that anyone can construct efficient new tools, without the need for much programming experience or a compiler. The development of a graphical user interface on top of the infrastructure will further ease the development cycle.

Acknowledgements

We would like to thank Stephen Clark, Tara Murphy, and the anonymous reviewers for their comments on drafts of this paper. This research is supported by a Commonwealth scholarship and a Sydney University Travelling scholarship.

References

David Abrahams. 2003. Boost.Python C++ library. http://www.boost.ory (viewed 23/3/2003).

Andrei Alexandrescu. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ In-Depth Series. Addison-Wesley, New York.

Michele Banko and Eric Brill. 2001. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th annual meeting of the Association for Computational Linguistics*, pages 26–33, Toulouse, France, 9–11 July.

Samuel Bayer, Christine Doran, and Bryan George. 2001. Dialogue interaction with the DARPA Communicator Infrastructure: The development of useful software. In J. Allan, editor, *Proceedings of HLT 2001, First International Conference on Human Language Technology Research*, pages 114–116, San Diego, CA, USA. Morgan Kaufmann.

Avrim Blum and Tom Mitchell. 1998. Combining labeled and unlabeled data with co-training. In *Proceedings of the 11th Annual Conference on Computational Learning Theory*, pages 92–100, Madisson, WI, USA.

Thorsten Brants. 2000. TnT - a statistical part-of-speech tagger. In *Proceedings of the 6th Conference on Applied Natural Language Processing*, pages 224–231, Seattle, WA, USA, 29 April – 4 May.

Eric Brill. 1993. *A Corpus-Based Appreach to Language Learning*. Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

Stanley Chen and Ronald Rosenfeld. 1999. A Gaussian prior for smoothing maximum entropy models. Technical report, Carnegie Mellon University.

Stephen Clark, James R. Curran, and Miles Osborne. 2003. Bootstrapping POS-taggers using unlabelled data. In *Proceedings of the 7th Conference on Natural Language Learning*, Edmonton, Canada, 31 May – 1 June. (to appear).

Hamish Cunningham, Yorick Wilks, and Robert J. Gaizauskas. 1997. GATE – a general architecture for text engineering. In *Proceedings of the 16th International Conference on Computational Linguistics*, pages 1057–1060, Copenhagen, Denmark, 5–9 August.

Hamish Cunningham, Diana Maynard, C. Ursu K. Bontcheva, V. Tablan, and M. Dimitrov. 2002. Developing language processing components with GATE. Technical report, University of Sheffi eld, Sheffi eld, UK.

Hamish Cunningham. 2000. *Software Architecture for Language Engineering*. Ph.D. thesis, University of Sheffi eld.

James R. Curran and Stephen Clark. 2003. Investigating GIS and smoothing for maximum entropy taggers. In *Proceedings of the 11th Meeting of the European Chapter of the Association for Computational Lingustics*, pages 91–98, Budapest, Hungary, 12–17 April.

James R. Curran and Marc Moens. 2002. Scaling context space. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Philadelphia, PA, USA, 7–12 July.

Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

Walter Daelemans, Jakub Zavrel, Ko van der Sloot, and Antal van den Bosch. 2002. TiMBL: Tilburg Memory-Based Learner reference guide. Technical Report ILK 02-10, Induction of Linguistic Knowledge. Tilburg University.

Ido Dagan and Sean P. Engelson. 1995. Committee-based sampling for training probabilistic classifi ers. In *Proceedings of the International Conference on Machine Learning*, pages 150–157, Tahoe City, CA, USA, 9–12 July.

David Day, John Aberdeen, Lynette Hirschman, Robyn Kozierok, Patricia Robinson, and Marc Vilain. 1997. Mixed-initiative development of language processing systems. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, pages 384–355, Washington, DC, USA, 31 March – 3 April.

Cristiane Fellbaum, editor. 1998. *Wordnet: an electronic lexical database*. The MIT Press, Cambridge, MA USA.

Claire Grover, Colin Matheson, Andrei Mikheev, and Marc Moens. 2000. LT TTT - a fexible tokenisation tool. In *Proceedings of Second International Language Resources and Evaluation Conference*, pages 1147–1154, Athens, Greece, 31 May – 2 June.

Kadri Hacioglu and Bryan Pellom. 2003. A distributed architecture for robust automatic speech recognition. In *Proceedings of Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Hong Kong, China, 6–10 April.

Nancy Ide, Randi Reppen, and Keith Suderman. 2002. The american national corpus: More than the web can provide. In *Proceedings of the Third Language Resources and Evaluation Conference*, pages 839–844, Las Palmas, Canary Islands, Spain.

Lauri Karttunen, Tamás Gaál, and André Kempe. 1997. Xerox Finite-State Tool. Technical report, Xerox Research Centre Europe Grenoble, Meylan, France.

Linguistic Data Consortium. 2003. English Gigaword Corpus. catalogue number LDC2003T05.

Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the Workshop on Effective Tools and Methodologies for Teaching NLP and Computational Linguistics*, pages 63–70, Philadelphia, PA, 7 July.

Robert Malouf. 2002. A comparison of algorithms for maximum entropy parameter estimation. In *Proceedings of the 6th Conference on Natural Language Learning*, pages 49–55, Taipei, Taiwan, 31 August – 1 September.

Mitchell Marcus, Beatrice Santorini, and Mary Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Andrei Mikheev, Claire Grover, and Marc Moens. 1999. Xml tools and architecture for named entity recognition. *Journal of Markup Languages: Theory and Practice 1*, 3:89–113.

Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. 1998. A rational design for a weighted fi nite-state transducer library. *Lecture Notes in Computer Science*, 1436.

Grace Ngai and Radu Florian. 2001. Transformation-based learning in the fast lane. In *Proceedings of the Second Meeting of the North American Chapter of the Association for Computational Linguistics*, pages 40–47, Pittsburgh, PA, USA, 2–7 June.

Adwait Ratnaparkhi. 1996. A maximum entropy part-of-speech tagger. In *Proceedings of the EMNLP Conference*, pages 133–142, Philadelphia, PA, USA.

Adwait Ratnaparkhi. 1998. *Maximum Entropy Models for Natural Language Ambiguity Resolution*. Ph.D. thesis, University of Pennsylvania.

Emmanuel Roche and Yves Schabes. 1997. Deterministic part-of-speech tagging with fi nite-state transducers. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, chapter 7. The MIT Press.

Hans van Halteren, Jakub Zavrel, and Walter Daelemans. 2001. Improving accuracy in wordclass tagging through combination of machine learning systems. *Computational Linguistics*, 27(2):199–229.

Ian H. Witten and Eibe Frank. 1999. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers.