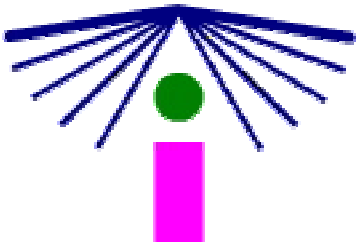

Text Analysis International Inc.

INTEGRATED DEVELOPMENT ENVIRONMENTS FOR NATURAL LANGUAGE PROCESSING

Text Analysis International, Inc.
1669-2 Hollenbeck Ave. #501
Sunnyvale, California 94087

October 2001

INTRODUCTION

From the time of the first Russian-to-English translation projects of the 1950s, a tremendous amount of research and development has been expended to automate the analysis and understanding of unstructured or "free" text.

Children universally become fluent in their native language within the first few years of life, leading us to expect that language should be easy to process using computers. However, language processing has proved to be an extremely complex field of endeavor. Natural Language Processing (NLP) is the branch of Artificial Intelligence (AI) devoted to this area, along with its sister discipline, Computational Linguistics (CL). Knowledge Representation (KR) theories also play an important role in the use of linguistic, conceptual, and domain knowledge required for properly automating language processing.

The past ten years have seen rapid and substantial growth in the commercialization of NLP, along with the widespread assessment that NLP is currently a critical technology for business. Two trends are largely responsible for this: the ascent of Natural Language Engineering (NLE) and statistical NLP. Both arose in response to the brittleness of previous approaches based on formal theories and crudely handcrafted systems.

NLE emerged almost overnight, propelled by successes such as the FASTUS system [REF SRI, Hobbs et al, 1993] at the fourth DARPA Message Understanding Conference (MUC-4). Here, an internationally respected research group that had spent decades on a formal approach to language processing reversed course and implemented an "unprincipled" *information extraction* (IE) system that achieved among the highest scores in competitive blind testing.

Statistical NLP approaches have yielded strong commercial successes in recent years, primarily in text categorization applications. The popularity of weak methods such as Bayesian statistics, machine learning, and neural networks is due to the ease of their implementation and the fact that developers don't have to "write the rules." However, systems based solely on such shallow methods are limited in terms of accuracy, extensibility, and applicability to NLP tasks.

What is the next required step in NLP? To quickly build a fast, accurate, extensible, and maintainable NLP application, one requires excellent tools that integrate multiple methods and strategies. Currently, the field primarily employs discrete, ad hoc tools and components, with no overall framework for building a complete text analysis system.

WHERE ARE THE TOOLS?

1.1 Requirements for an IDE

In every well-established area of computer software, tools have sprung up to organize, accelerate, and standardize the development of applications. Prime examples are computer programming applications, which are now universally developed via integrated development environments (IDEs) such as Microsoft Visual C++ (c). IDEs offer universally recognized features such as:

- A graphical user interface (GUI) that enables users to conveniently manage the task at hand
- A complete environment for building an application
- Helpful views of the application and components under construction
- Cross-references to provide instant feedback about changes to the application
- Prebuilt framework, components, and tools that work together seamlessly
- Open architecture for integrating multiple methods and external components
- Debugging facilities to step through the running of the application and identify errors
- Integrated documentation to help the developer answer technical questions quickly

The IDE used in building an application doubles as a platform for maintaining and enhancing the application. The modular structure of an application built with an IDE enables developers other than the creators of the application to understand, maintain, and enhance it. In addition, non-expert users of the application should be able to maintain and enhance it, to a reasonable extent, using the IDE. The benefits of IDEs include

- Accelerated development of applications
- Properly engineered, modular, and organized applications
- Applications with reusable components
- Highly extensible and maintainable applications
- Bottom line: reduced resources needed to build and maintain applications

1.2 NLP Tools

While many excellent tools have been built for NLP, precious few merit the label "IDE."

A variety of tool kits and software development kits (SDKs) exist, all claiming to substantially aid in the construction of NLP applications. In some of these offerings, the tool kit is little more than a bag of tricks that cannot easily be combined. Each program in the kit may perform its task well and quickly, but developers must cobble the programs together into a patchwork system of diverse components and data representations, and must provide their own programmatic "glue", somehow adding hand-built components so that the application can actually perform the tasks required of it.

Typical tool kits may include programs for:

- Tokenizing
- Morphological analysis
- Spelling correction
- Part-of-speech tagging
- Entity recognition or simple extraction (names, titles, locations, dates, quantities)
- Constituent recognition (e.g., noun phrase recognition)

It is often the case that these components are given as "black boxes." That is, they cannot be read, modified, or enhanced, but must be run as they are given.

More integrated tool kits are also available, in which the component programs can run in *multiple passes*. For example, the part-of-speech tagger (that identifies nouns, verbs, and so on) may use the tokenizer's output, and so on. In this way, the programs can be cascaded so as to work in concert.

But in this integrated tool kit, at least some of the components are still black boxes. Furthermore, to place additional passes between the given programs requires programmers to build ad hoc passes that conform to the output of one program and the input of the other.

Many critical components are noticeably absent from available tool kits, including: parsing, semantic analysis, and discourse analysis. All of these present inordinate difficulties for most tool vendors: they don't have an adequate solution and typically offer no framework or support for crafting a solution.

Even with the integrated tool kit, the output of the final cascaded program typically bears no resemblance to the output required of the application. If developers need an excellent part-of-speech tagger, perhaps they can use one that is provided, but then they must complete the application – somehow adding the most difficult components, such as parsing, semantic, and discourse analysis -- with no assistance from the tool kit. And the only way to enhance the provided tagger is to wait for an upgrade from the vendor.

1.3 FASTUS

One of the attractions of FASTUS, mentioned earlier, is that it supports the construction of information extraction applications. FASTUS uses a set of cascaded components that do the complete job. The multiple passes of FASTUS are as follows (depending on which historical version is being referenced):

Name Recognizer
Parser
Combiner
Domain Phase

In essence, FASTUS looks very much like the integrated tool kits described earlier. But it supports the functionality needed for an end-to-end *information extraction* system and it provides modifiable components – users may add rules and patterns to the system.

While FASTUS may have been one of the better performers at the MUC conferences [REFERENCE], its rating in the information extraction task at MUC-6 (one of the later MUC conferences) of 44% recall and 61% precision leaves substantial room for improvement.

To quote one of the FASTUS papers [FASTUS: Extracting Information from Natural Language Texts, by Jerry Hobbs et al.]:

> "While FASTUS is an elegant achievement, the whole host of linguistic problems that were bypassed are still out there, and will have to be addressed eventually for more complex tasks, and to achieve higher performance on simple tasks."

1.4 NLP Frameworks

To support the creation of accurate and complete NLP systems, we require much more than a medley of components, no matter how good they are individually. We also require a more flexible arrangement than a hard-wired ordering of passes and algorithms, even when those

passes can be modified by the user. In short, we require a generalized framework for managing passes, algorithms, and their associated rules, data structures, and knowledge.

Given such a flexible arrangement, one can organize an NLP system in the best possible way for any given application. Furthermore, the ability to insert passes into an existing set of passes enables a system to grow in a flexible and modular fashion. For example, some passes can be devoted entirely to syntax, others to lexical processing, others to domain specific tasks, and so on. Each pass may consist of a simple process such as segmenting into lines, or a complex subsystem such as a recursive grammar for handling lists.

Also, by conditionally executing passes, a single NLP system may identify, process and unify a wide variety of input formats such as RTF, PDF, and HTML, in addition to plain text.

Consider three enticing scenarios for a flexible multi-pass framework:

High confidence processing. Imagine an NLP system that performs a complete set of processes, but only when the independent confidence of each action exceeds 99%. That is, it performs lexical, syntactic, semantic, and discourse processing only where actions can be performed with the highest accuracy. While not many actions may be actually executed, those that are performed will be extremely accurate. Furthermore, matched constructs will serve as *context* for subsequent actions. That is, after one complete cycle, lower the confidence threshold to 98% and repeat the processing cycle, then lower to 97% and so on, until some minimal threshold is reached. Each cycle works in more constrained contexts that are characterized with high confidence. In this way, a very accurate overall processing of text can occur.

Simulating a recursive grammar. While recursive grammar approaches are typically plagued by problems of multiple interpretations and ambiguity, imagine a parser that constructs only a single parse tree. Each step in a multi-pass NLP system can assess and control the modifications to a parse tree, allowing implementation of a "best first" stepwise approach to simulating the work of a recursive grammar.

Characterization. In addition to passes that modify a parse tree, one can intersperse any number of passes whose sole purpose is to characterize the text at various points, in order to gather evidence and preferences for subsequent processing tasks. For example, one can gather various types of evidence about a period and its context to ascertain its likelihood of being an end of sentence. Segmenting at ambiguous ends of sentence can be deferred until enough evidence has been accumulated for or against.

THE SHALLOW ONES

A host of companies in the market place claim that hand-building NLP systems is not needed. This may be so in the case of shallow applications such as categorization, summarization, and attribute extraction, where high accuracy is not as important as getting a system up quickly. In some simple applications, high accuracy has also been achieved, but as the tasks become more complex, the accuracy falls off rapidly.

The primary technologies used for shallow processing are statistical NLP (e.g., Bayesian methods), probabilistic methods (e.g., Hidden Markov Models), neural networks (NN), and machine learning (ML).

These are excellent technologies that enable systems to be constructed automatically or by simpler set ups than handcrafting an NLP system. They should be part of a complete framework for NLP. However, these methods by themselves cannot support high precision and recall in many applications, and they cannot by themselves support deep, accurate, and complete analysis and understanding of text.

The most advanced types of systems built exclusively with these methods are on the order of "name recognition" or "entity recognition." That is, if a text has items to be identified individually, then statistical methods may be able to support an accurate and complete solution. Thus, for example, a shallow system can be built to accurately find the pieces of a job description, including the job title, contact information, years of experience, and other discrete, non-repeating data types.

But for texts that mix multiple events or repeating data items, such as a Wall Street Journal article or employment resume, statistical methods will in general not suffice to accurately extract and synthesize business event records or contact, employment, and education records, respectively. The more complex the domain of discourse and the more structured the desired information, the more NLP and customization will be required, relative to shallow automated methods.

Implementing shallow methods is relatively cut-and-dried, and therefore competition is fierce among companies that critically depend on them for their technology and products. While substantial hype emanates from many such companies, customer experience is often at odds with the exaggerated claims.

Categorization is an interesting application to examine, since it is widely implemented by text analysis vendors. To automatically distinguish sports and business articles accurately, for example, is feasible by all the shallow methods. However, as finer-grained or overlapping categories are specified, e.g., indoor versus outdoor sports, accuracy of automatically generated categorizers will quickly drop off. Similarly, processing texts that entangle multiple categories of interest will typically yield unsatisfactory results by these methods. Categories that share substantial numbers of keywords present difficult obstacles to these methods.

One qualitative difference between these technologies and NLP for many applications is: with shallow methods, a person still has to read every text of interest, while deeper processing can convert text to a database or other structured representation, so that people need not read the voluminous text. Instead, they can query a database to immediately get precise answers to queries such as "list all the acquisitions in the last quarter valued at between $10M and $50M."

Another grand claim of shallow methods is that they are "language independent." While this is true and is a positive feature in that the methods can be applied quickly, it also means that these methods can never suffice to *understand* language. They only work by gathering information about frequency, proximity, and other statistics relating to patterns of letters.

While shallow methods may eventually be used to accurately perform complex language analysis tasks, the state of the art is years away from that point. Statistical NLP is still in its infancy and we cannot rely on it exclusively for commercial strength solutions to complex language processing tasks.

It also seems to be the case that the more fine-grained the learning required of a system, the more manually prepared data is required. It is not at all clear that the preparation of such data will be less resource-intensive than manually engineering an NLP system.

INTEGRATED DEVELOPMENT ENVIRONMENTS

The purpose of an IDE is to accelerate the development of an accurate, fast, robust, and maintainable application.

To be sure, there has been a substantial effort to develop IDEs for NLP in recent years. Even going back to the 1980s, Carnegie Group's LanguageCraft was a notable early attempt to commercialize an IDE. But knowledge of NLP then was not what it is today and hardware was not as fast and capable, so LanguageCraft and some other early efforts did not succeed commercially.

Several IDEs under development today are found in the universities and research organizations. VisualText is by far the most ambitious and groundbreaking commercial IDE developed to date.

The reason we have not seen more effort in this area is that the shallow methods alluded to earlier have been the "hot topic" of the 1990s. Why build and use an IDE when you can automate the processing of text? As we have seen, the shallow methods are very useful, but are insufficient for engineering a complete and accurate NLP system for a task of substantial complexity.

For most complex tasks, a combination of strategies is required. The ideal IDE will integrate shallow methods with methods based on tokens, patterns, grammars, keywords, logic, and finite state automata.

It will integrate knowledge bases, expert systems, experts, planners, and other artificial intelligence systems with the overall framework.

VisualText is unique in that it incorporates a programming language to help "glue" the various methods and systems, as well as to specify actions to undertake in the most general way possible. For example, an entire program can accompany the matching of a single pattern in NLP++.

We stress that IDEs in no way conflict with shallow methods. IDEs should serve as a framework for incorporating and integrating shallow and deep methods for text analysis. They should further serve to support ever-greater degrees of automation in the construction of text analyzers.

Similarly, IDEs do not conflict or compete with particular paradigms for NLP, such as HPSG (Head Driven Phrase Structure Grammar) and other popular approaches. IDEs constitute a "meta" approach that incorporates all possible methods.

It has been claimed that handcrafting analyzers requires inordinate amount of time, effort, and bottom line cost. This is no doubt true when inexperienced developers attempt build text analysis solutions from scratch. The result in these cases is typically a brittle, under-performing, and inextensible analyzer that can only be maintained by its developers.

In the last decade, the indications are strong that NLP has found a place in the market. Commercial-grade NLP systems have been built and are being built in such areas as medical coding, email analysis, legal analysis, resume analysis, chat management, and more. But the downsides are still maintainability, cost to build, expertise, and resources.

NLP developers are not generally aware that IDEs for text analysis exist, but we expect that situation to change substantially in the coming years, with the commercial release of IDEs. The time is now. With several academic IDEs gaining substantial popularity, the window of opportunity for IDEs for NLP is expanding quickly. With the growing popularity of multi-pass frameworks such as FASTUS, further inroads into the marketplace are being established. We expect VisualText to play a major role in transforming the development and maintenance of NLP systems.

IDEs support, organize, and dramatically accelerate the construction of accurate, fast, extensible, and maintainable text analyzers. Furthermore, because multi-pass analyzers can be constructed in modular fashion, each new analyzer built can reuse substantial parts of previously built analyzers.

1.5 A Practical Development Scheme

Typically the most important measure of performance for an analyzer is *recall*, which is a measure of *completeness* and coverage of the analyzer. That is, if it accurately extracts all the required information and only the required information, then it has high recall.

A second measure is *precision*. If every output that an analyzer produces is correct, even though it may not extract everything it needs to, then the analyzer has high precision.

When one starts building an analyzer, it's recall and precision start at 0%. But the minute we output an extracted datum, precision can shoot up to 100% if that datum is always correct.

Therefore, an obvious and very practical methodology strives to keep precision as high as possible, while successively increasing the coverage of the analyzer. It turns out that, as the analyzer extracts more information, or does more "work", mistaken outputs will necessarily increase. Therefore it is axiomatic that as an analyzer is developed, *precision will decrease as recall is increased*. However, by carefully crafting the analyzer and looking to shore up precision every time coverage is increased, the decrease in precision can be minimized relative to the increase in recall.

Another important guideline for analyzer building is to have the analyzer assess its own confidence in the decisions it makes, actions it takes, and outputs that it produces. In this way, the user of the analyzer can control the precision and recall tradeoff by specifying a confidence threshold that all outputs must exceed.

1.6 Knowledge

To achieve the highest levels of accuracy in automatically processing text, the analyzer must in general *understand* the text much as a human being does. Overlapping NLP is a field of endeavor called natural language understanding (NLU) that emphasizes this aspect of processing text.

To understand a text, or to attempt to simulate human understanding of a text, many types of knowledge must be brought to bear, including at least:

- Linguistic – knowledge of language, its use, and its processing
- Conceptual – general knowledge of the world, or commonsense knowledge

- Domain – specific knowledge about the specialized domains of discourse touched on in a text.

Some text analysis technologies emphasize knowledge over the practical aspects of processing texts. For example, CYC has expended tremendous resources to compile a comprehensive commonsense knowledge base.

Our view is that knowledge is a tool to be used and balanced with other tools. It is easy to be swamped with too much knowledge. For example, a dictionary entry for the word "take" may list 50 distinct senses. But for a text analysis system to attempt to choose among these senses is ludicrous in most cases.

One successful method for building text analyzers stresses "just in time" knowledge. That is, knowledge is incorporated by the developers only where it proves essential to make progress. This approach works very well for text analyzers in restricted domains (e.g., business and medical).

While broad coverage analyzers exist for applications such as categorization, the full utilization of large-scale dictionaries has not yet been accomplished in practice.

Another problem with knowledge is that it hasn't been standardized. Every dictionary and ontology has its own biases, and disagreements about standardization abound.

IDEs must be flexible enough to enable the incorporation of diverse and sometimes competing knowledge representation schemes. IDEs may also serve as a platform for moving to practical and standardized representations of knowledge.

1.7 Beyond Text Analysis

IDEs for NLP can be designed along two lines. The first is as a completely specialized environment for NLP and not much else. The second is as a general development environment augmented with specializations for NLP.

Because the second design is a superset of the first, it is clearly preferable. Basically, it allows the IDE to serve additional needs of developers without their having to switch between a programming environment and an NLP environment.

To support such a capability, the IDE should provide a programming language that not only addresses the needs of NLP but also of general programming.

NLP++ is the first and only programming language, to our knowledge, that addresses both NLP and general programming needs. Though it is an evolving programming language, even in its current state it can serve to "glue" components and extend the processing capability of an NLP system in ways that no other development framework can match.

Because NLP is extremely complex and difficult, applying NLP-grade technology to simpler problems such as pattern matching and low-level document processing should be "overkill." It should be easy to develop applications such as

- Compilers for programming languages and other formal languages
- Error checking and correcting compilers

- General pattern matching
- General file processing
- Format conversion
- HTML generation (as performed by Perl and other CGI languages)

NLP++ supports these types of applications very well. Indeed, NLP++ itself is parsed and compiled by an analyzer written in NLP++. A bootstrapping analyzer for a simple subset of NLP++ is built into the VisualText analyzer engine. This parses the full NLP++ specification, which then parses the user-defined analyzer.

1.8 NLP Programming Language

What is a programming language for NLP, and what should it look like? What is lacking in Prolog, Lisp, C++, Java, and other programming languages, when building NLP systems?

An NLP programming language should empower developers to engineer a complete text analyzer. It should support

- Processing at all levels, including tokenization, morphological, lexical, syntactic, semantic, and discourse
- Constructing all needed data structures at each level of processing
- Deploying all the methods needed for text analysis, including pattern, grammar, keyword, and statistical.
- Executing pattern matching mechanisms for all methods
- Attaching arbitrary actions to matched patterns
- Elaborating the heuristics that augment and control text analysis

An NLP programming language should, at a minimum, support both patterns and actions associated with matching those patterns. It should also provide the "glue" we need to build a practical application, so that we don't need to cobble together ad hoc components of an overall analyzer.

One early system still in use for combining grammars and general programming languages is YACC (Yet Another Compiler Compiler). Bison is a popular copycat version for Gnu/Linux systems. YACC is attractive in that, when a rule matches, one can specify arbitrary actions in a general programming language. However, YACC is best suited for processing formal grammars and languages, rather than natural languages. Some extensions to YACC address ambiguous languages, leading to somewhat better handling of natural languages, but these enhancements are not sufficient to support a complete NLP system. Furthermore, YACC is limited to a single recursive grammar executed by a single processing algorithm.

Natural language requires a more flexible approach. As we've described earlier, the multi-pass methodology has proved ideal for building analyzers. In this way, each pass can employ its own processing algorithm rule set, knowledge, and actions. An NLP programming language should encompass the multi-pass methodology and all the objects and processes needed in each pass of such a methodology.

The multiple passes of such a methodology must communicate with each other. The output of one pass should serve as the input to the next. A *parse tree* is universally recognized as an ideal data structure for holding at least a subset of such information. Further, the multiple passes should be able to post to and utilize information from a shared knowledge base. An NLP programming language should facilitate the management of parse trees and knowledge bases.

A reasoning engine should also be supported. Ideal might be to integrate an expert system with the multi-pass framework and the knowledge base. At least, the NLP programming language should support arbitrary heuristics and algorithms, in the absence of an expert system.

Declarative and procedural knowledge should be segregated as far as possible, with the understanding that rules and their associated actions must be bound in some manner.

To truly support the construction of a complete NLP system, the programming language must incorporate many of the features of a general programming language such as C++ or Java. On top of these, it must provide an additional layer or libraries that support the specialized needs of NLP.

In these regards, NLP++ is by far the most advanced programming language for NLP. NLP++ is an evolving general programming language that "talks" parse tree and knowledge base, as well as supporting a spectrum of actions tailored to rule and pattern matching. Parse tree nodes and knowledge base objects are built-in NLP++ data types, enabling simple programmatic manipulation of these objects. NLP++ also enables dynamically decorating the parse tree and knowledge base with semantic representations.

A key feature of NLP++ is that its multi-pass framework communicates via a *single* parse tree. In this way, combinatorial explosion is not an option. The text analyzer must be engineered so that the best possible parse tree is elaborated in stages. Further, constraining the analyzer to deal with a single parse tree helps guide the development of a fast and efficient text analyzer.

Unlike recursive grammar approaches, the multi-pass methodology enables the elaboration of a parse tree that represents the best understanding of a text. It is not required to reduce every sentence to a "sentence node." Where the coverage is sufficient, such a node will be produced. But useful information and patterns may be found in sentences that are malformed, ungrammatical, terse, or otherwise not covered by the text analyzer's "grammar." Hence, the multi-pass approach also yields robust text analyzers that can operate on messy input (such as text from speech and from OCR).

1.9 Learning and Training

We have described the recent popularity of statistical methods for text processing, including Bayesian, probabilistic, Hidden Markov Models, neural network, and machine learning.

An IDE should support the use of statistical methods where appropriate and deeper NLP methods where they are best suited as well. In a multi-pass analyzer, a sequence of passes can be dedicated to automated training by statistical methods. Once training is complete, those passes can be deactivated or skipped so that the analyzer can be run against "real" input texts.

Various VisualText applications have utilized statistical methods. For example, a generic analyzer framework has been built that incorporates statistical part-of-speech tagging of text.

A related method that VisualText supports is an automated rule generation (RUG) facility. Developers and users may highlight text samples and categorize them (i.e., *annotate* text), letting RUG automatically generalize and merge the patterns represented by the samples, automatically

creating accurate rules. Various statistical methods require large bodies of annotated text. RUG is a powerful methodology for several reasons:

- Developers control rule generalization, merging, and generation
- Accurate rules are generated from relatively few samples
- High quality rules result from integrating NLP passes with RUG passes
- Generated rules are *self maintaining*. They adapt to changes in the analyzer

We are aware of no capability comparable to RUG, other than the general similarity with statistical methods that utilize annotated (i.e., tagged) texts. The VisualText IDE exploits multiple methods to create novel advances such as RUG.

1.10 Existing IDEs

PRODUCT	ORGANIZATION	DESCRIPTION
Alembic Workbench	MITRE	Engineering environment for tagged texts
GATE	University of Sheffield	General architecture for text engineering, including information extraction
INTEX	LADL	For developing linguistic finite state machines with large-coverage dictionaries and grammars
LexGram	University of Stuttgart	Grammar development tool for categorial grammars
PAGE	DFKI	NLP core engine for development of grammatical and lexical resources
Pinocchio	ITC-Irst	Development environment with rich GUI for developing and running information extraction applications
VisualText	Text Analysis International	Comprehensive IDE supporting deep NLP, with integrated NLP++ language and knowledge base management

The table above lists the major GUI environments for NLP available today. Some are the work of academic, rather than commercial, entities. VisualText is far and away the most advanced IDE for natural language processing ever developed and commercialized. By virtue of the NLP++ programming language, it is unique in its ability to support the development of a complete text analyzer entirely within a GUI environment.