NLTK Tutorial: Introduction to Natural Language Processing

Steven Bird
Ewan Klein
Edward Loper

Revision 1.66, 7 Apr 2005

Copyright © 2005

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

> The single and shortest definition of civilization may be the word *language*... Civilization, if it means something concrete, is the conscious but unprogrammed mechanism by which humans communicate. And through communication they live with each other, think, create, and act.
>
> —John Ralston Saul

1. The Language Challenge

Language is the chief manifestation of human intelligence. Through language we express basic needs and lofty aspirations, technical know-how and flights of fantasy. Ideas are shared over great separations of distance and time. The following samples from English illustrate the richness of language:

1.

   a. Overhead the day drives level and grey, hiding the sun by a flight of grey spears. (William Faulkner, *As I Lay Dying*, 1935)

   b. When using the toaster please ensure that the exhaust fan is turned on. (sign in dormitory kitchen)

   c. Amiodarone weakly inhibited CYP2C9, CYP2D6, and CYP3A4-mediated activities with Ki values of 45.1-271.6 μM (Medline)

   d. Iraqi Head Seeks Arms (spoof headline, `http://www.snopes.com/humor/nonsense/head97.htm`)

   e. The earnest prayer of a righteous man has great power and wonderful results. (James 5:16b)

f. Twas brillig, and the slithy toves did gyre and gimble in the wabe (Lewis Carroll, *Jabberwocky*, 1872)

g. There are two ways to do this, AFAIK :smile: (internet discussion archive)

Thanks to this richness, the study of language is part of many disciplines outside of linguistics, including translation, literary criticism, philosophy, anthropology and psychology. Many less obvious disciplines investigate language use, such as law, hermeneutics, forensics, telephony, pedagogy, archaeology, cryptanalysis and speech pathology. Each applies distinct methodologies to gather observations, develop theories and test hypotheses. Yet all serve to deepen our understanding of language and of the intellect which is manifested in language.

The importance of language to science and the arts is matched in significance by the cultural treasure that is inherent in language. Each of the world's ~7,000 human languages is rich in unique respects, in its oral histories and creation legends, down to its grammatical constructions and its very words and their nuances of meaning. Threatened remnant cultures have words to distinguish plant subspecies according to therapeutic uses which are unknown to science. Languages evolve over time as they come into contact with each other and they provide a unique window onto human pre-history. Technological change gives rise to new words like *weblog* and new morphemes like *e-* and *cyber-*. In many parts of the world, small linguistic variations from one town to the next add up to a completely different language in the space of a half-hour drive. For its breathtaking complexity and diversity, human language is as a colourful tapestry stretching through time and space.

Each new wave of computing technology has faced new challenges for language analysis. Early machine languages gave way to high-level programming languages which are automatically parsed and interpreted. Databases are interrogated using linguistic expressions like `SELECT age FROM employee`. Recently, computing devices have become ubiquitous and are often equipped with multimodal interfaces supporting text, speech, dialogue and pen gestures. One way or another, building new systems for natural linguistic interaction will require sophisticated language analysis.

Today, the greatest challenge for language analysis is presented by the explosion of text and multimedia content on the world-wide web. For many people, a large and growing fraction of work and leisure time is spent navigating and accessing this universe of information. *What tourist sites can I visit between Philadelphia and Pittsburgh on a limited budget? What do expert critics say about Canon digital cameras? What predictions about the steel market were made by credible commentators in the past week?* Answering such questions requires a combination of language processing tasks including information extraction, inference, and summarisation. The scale of such tasks often calls for high-performance computing.

As we have seen, *natural language processing*, or NLP, is important for scientific, economic, social, and cultural reasons. NLP is experiencing rapid growth as its theories and methods are deployed in a variety of new language technologies. For this reason it is important for a wide range of people to have a working knowledge of NLP. Within academia, this includes people in

areas from humanities computing and corpus linguistics through to computer science and artificial intelligence. Within industry, this includes people in human-computer interaction, business information analysis, and web software development. We hope that you, a member of this diverse audience reading these materials, will come to appreciate the workings of this rapidly growing field of NLP and will apply its techniques in the solution of real-world problems. The following chapters present a carefully-balanced selection of theoretical foundations and practical application, and equips readers to work with large datasets, to create robust models of linguistic phenomena, and to deploy them in working language technologies. By integrating all of this with the Natural Language Toolkit (NLTK), we hope this book opens up the exciting endeavour of practical natural language processing to a broader audience than ever before.

> **Note:** An important aspect of learning NLP using these materials is to experience both the challenge and — we hope — the satisfaction of creating software to process natural language. The accompanying software, NLTK, is available for free and runs on most operating systems including Linux/Unix, Mac OSX and Microsoft Windows. You can download NTLK from `nltk.sourceforge.net`, along with extensive documentation. We encourage you to install NLTK on your machine before reading beyond the end of this chapter.

2. A Brief History of Natural Language Processing

A long-standing challenge within computer science has been to build intelligent machines. The chief measure of machine intelligence has been a linguistic one, namely the *Turing Test*. Can a *dialogue system*, responding to a user's typed input with its own textual output, perform so naturally that users cannot distinguish it from a human interlocutor using the same interface? Today, there is substantial ongoing research and development in such areas as machine translation and spoken dialogue, and significant commercial systems are in widespread use. The following dialogue illustrates a typical application:

```
S: How may I help you?

U: When is Saving Private Ryan playing?

S: For what theater?

U: The Paramount theater.

S: Saving Private Ryan is not playing at the Paramount theater, but
   it's playing at the Madison theater at 3:00, 5:30, 8:00, and 10:30.
```

Today's commercial dialogue systems are strictly limited to narrowly-defined domains. We could not ask the above system to provide driving instructions or details of nearby restaurants unless the requisite information had already been stored and suitable question and answer sentences had been incorporated into the language processing system. Observe that the above system appears to understand the user's goals: the user asks when a movie is showing and the system correctly determines from this that the user wants to see the movie. This inference seems so obvious to humans that we usually do not even notice it has been made, yet a natural language system needs to be endowed with this capability in order to interact naturally. Without it, when asked `Do you know when Saving Private Ryan is playing`, a system might simply — and unhelpfully — respond with a cold `Yes`. While it appears that this dialogue system can perform simple inferences, such sophistication is only found in cutting edge research prototypes. Instead, the developers of commercial dialogue systems use contextual assumptions and simple business logic to ensure that the different ways in which a user might express requests or provide information are handled in a way that makes sense for the particular application. Thus, whether the user says `When is ...` , or `I want to know when ...` , or `Can you tell me when ...` , simple rules will always result in users being presented with screening times. This is sufficient for the system to provide a useful service.

Despite some recent advances, it is generally true that those natural language systems which have been fully deployed still cannot perform common-sense reasoning or draw on world knowledge. We can wait for these difficult artificial intelligence problems to be solved, but in the meantime it is necessary to live with some severe limitations on the reasoning and knowledge capabilities of natural language systems. Accordingly, right from the beginning, an important goal of NLP research has been to make progress on the holy grail of natural linguistic interaction *without* recourse to this unrestricted knowledge and reasoning capability. This is an old challenge, and so it is instructive to review the history of the field.

The very notion that natural language could be treated in a computational manner grew out of a research program, dating back to the early 1900s, to reconstruct mathematical reasoning using logic, most clearly manifested in the work by Frege, Russell, Wittgenstein, Tarski, Lambek and Carnap. This work led to the notion of language as a formal system amenable to automatic processing. Three later developments laid the foundation for natural language processing. The first was *formal language theory*. This defined a language as a set of strings accepted by a class of automata, such as context-free languages and pushdown automata, and provided the underpinnings for computational syntax.

The second development was *symbolic logic*. This provided a formal method for capturing selected aspects of natural language that are relevant for expressing logical proofs. A formal calculus in symbolic logic provides the syntax of a language, together with rules of inference and, possibly, rules of interpretation in a set-theoretic model; examples are propositional logic and first-order logic. Given such a calculus, with a well-defined syntax and semantics, it becomes possible to associate meanings with expressions of natural language by translating them into expressions of the formal calculus. For example, if we translate *John saw Mary* into a formula `saw(j,m)`, we (implicitly or explicitly) intepret the English verb *saw* as a binary relation, and *John* and *Mary* as

denoting individuals. More general statements like *All birds fly* require quantifiers, in this case ∀ meaning *for all*: ∀`x: bird(x)` ⟶ `fly(x)`. This use of logic provided the technical machinery to perform inferences that are an important part of language understanding. The third development was the *principle of compositionality*. This was the notion that the meaning of a complex expression is comprised of the meaning of its parts and their mode of combination. This principle provided a useful correspondence between syntax and semantics, namely that the meaning of a complex expression could be computed recursively. Given the representation of *It is not true that* $␣_p$ as `not(p)` and *John saw Mary* as `saw(j,m)`, we can compute the interpretation of *It is not true that John saw Mary* recursively using the above information to get `not(saw(j,m))`. Today, this approach is most clearly manifested in a family of grammar formalisms known as unification-based grammar, and NLP applications implemented in the Prolog programming language. This approach has been called *high-church* NLP, to highlight its attention to order and its ritualized methods for ensuring correctness.

A separate strand of development in the 1960s and 1970s eschewed the declarative/procedural distinction and the principle of compositionality. They only seemed to get in the way of building practical systems. For example, early question answering systems employed fixed pattern-matching templates such as: `How many` $␣_i$ `does` $␣_j$ `have?`, where slot `i` is a feature or service, and slot `j` is a person or place. Each template came with a predefined semantic function, such as `count(i,j)`. A user's question which matched the template would be mapped to the corresponding semantic function and then "executed" to obtain an answer, `k = count(i,j)`. This answer would be substituted into a new template: $␣_j$ `has` $␣_k$ $␣_i$. For example, the question *How many airports*$_i$ *does London*$_j$ *have?* can be mapped onto a template (as shown by the subscripts) and translated to an executable program. The result can be substituted into a new template and returned to the user: *London has five airports*. Finally, the subscripts are removed and the natural language answer is returned to the user.

This approach to NLP is known as *semantic grammar*. Such grammars are formalized like phrase-structure grammars, but their constituents are no longer grammatical categories like noun phrase, but semantic categories like *Airport* and *City*. These grammars work very well in limited domains, and are still widely used in spoken language systems. However, they suffer from brittleness, duplication of grammatical structure in different semantic categories, and lack of portability. This approach has been called *low-church* NLP, to highlight its readiness to adopt new methods regardless of their prestige, and a concomitant disregard for tradition.

In the preceding paragraphs we mentioned a distinction between high-church and low-church approaches to NLP. This distinction relates back to early metaphysical debates about *rationalism* versus *empiricism* and *realism* versus *idealism* that occurred in the Enlightenment period of Western philosophy. These debates took place against a backdrop of orthodox thinking in which the source of all knowledge was believed to be divine revelation. During this period of the seventeenth and eighteenth centuries, philosophers argued that human reason or sensory experience has priority over revelation. Descartes and Leibniz, amongst others, took the rationalist position, asserting that all truth has its origins in human thought, and in the existence of "innate ideas" implanted in our minds from birth. For example, they saw that the principles of Euclidean geometry were devel-

oped using human reason, and were not the result of supernatural revelation or sensory experience. In contrast, Locke and others took the empiricist view, that our primary source of knowledge is the experience of our faculties, and that human reason plays a secondary role in reflecting on that experience. Prototypical evidence for this position was Galileo's discovery — based on careful observation of the motion of the planets — that the solar system is heliocentric and not geocentric. In the context of linguistics, this debate leads to the following question: to what extent does human linguistic experience, versus our innate "language faculty", provide the basis for our knowledge of language? In NLP this matter surfaces as differences in the priority of corpus data versus linguistic introspection in the construction of computational models.

A further concern, enshrined in the debate between *realism* and *idealism*, was the metaphysical status of the constructs of a theory. Kant argued for a distinction between phenomena, the manifestations we can experience, and "things in themselves" which can never been known directly. A linguistic realist would take a theoretical construct like "noun phrase" to be real world entity that exists independently of human perception and reason, and which actually *causes* the observed linguistic phenomena. A linguistic idealist, on the other hand, would argue that noun phrases, along with more abstract constructs like semantic representations, are intrinsically unobservable, and simply play the role of useful fictions. The way linguists write about theories often betrays a realist position, while NLP practitioners occupy neutral territory or else lean towards the idealist position.

These issues are still alive today, and show up in the distinctions between symbolic vs statistical methods, deep vs shallow processing, binary vs gradient classifications, and scientific vs engineering goals. However, these contrasts are highly nuanced, and the debate is no longer as polarised as it once was. In fact, most of the discussions — and most of the advances even — involve a *balancing act* of the two extremes. For example, one intermediate position is to assume that humans are innately endowed with analogical and memory-based learning methods (weak rationalism), and use these methods to identify meaningful patterns in their sensory language experience (empiricism). For a more concrete illustration, consider the way in which statistics from large corpora may serve as evidence for binary choices in a symbolic grammar. For instance, dictionaries describe the words *absolutely* and *definitely* as nearly synonymous, yet their patterns of usage are quite distinct when combined with a following verb, as shown in Figure 1.

Figure 1. Absolutely vs Definitely (Liberman 2005, LanguageLog.org)

Google hits	adore	love	like	prefer
absolutely	289,000	905,000	16,200	644
definitely	1,460	51,000	158,000	62,600
ratio	198/1	18/1	1/10	1/97

Observe that *absolutely adore* is about 200 times as popular as *definitely adore*, while *absolutely prefer* is about 100 times rarer then *definitely prefer*. This information is used by statistical language models, but it also counts as evidence for a symbolic account of word combination in which *absolutely* can only modify extreme actions or attributes. This information could be represented as a binary-valued feature of certain lexical items. Thus, we see statistical data informing symbolic models. Now that this information is codified, it is available to be exploited as a contextual feature for a statistical language modelling, alongside many other rich sources of symbolic information, like hand-constructed parse trees and semantic representations. Now the circle is closed, and we see symbolic information informing statistical models.

This new rapprochement between high-church and low-church NLP is giving rise to many exciting new developments. We will touch on some of these in the ensuing pages. We too will perform this balancing act, employing approaches to NLP that integrate these historically-opposed philosophies and methodologies.

3. An NLP Application: Information Extraction

For many NLP applications, the priority is to extract meaning from written text. This must be done *robustly*; processing should not fail when unexpected or ill-formed input is received. Today, robust semantic interpretation is easiest when shallow processing methods are used. As we saw in the previous section, these methods severely limit (or even omit) syntactic analysis and restrict the complexity of the target semantic representations.

One well-known instance of shallow semantics is *Information Extraction* (IE). In contrast to full text understanding, IE systems only try to recognise a limited number of pre-specified semantic topics and use a constrained semantic representation. The initial phase of information extraction involves the detection of *named entities*, expressions that denote locations, people, companies, times, and monetary amounts. (In fact, these named entities are nothing other than the low-level categories of the semantic grammars we saw in the previous section, but under a new guise.) To illustrate, the following text contains several expressions which have been tagged as entities of types `time`, `company` and `location`:

```
The incident occurred <time>around 5.30pm</time> when a man walked into
<company>William Hill bookkeepers</company> on <location>Ware Road</location>,
and threatened a member of staff with a small black handgun.
```

The final output of an information extraction system, usually referred to as a *template*, consists of fixed number of slots that must be filled. More semantically, we can think of the template as denoting an event in which the participating entities play specific roles. Here is an example corresponding to the above text:

```
Crime event
     Time: around 5.30pm
```

```
Location: William Hill bookkeepers on Ware Road
Suspect: man
Weapon: small black handgun
```

Early work in IE demonstrated that hand-built rules could achieve good accuracy in the tasks defined by the DARPA Message Understanding Conferences (MUC). In general, highest accuracy has been achieved in identifying named entities, such as `William Hill bookkeepers`. However, it is harder to identify which slots the entities should fill; and harder still to accurately identify which event the entities are participating in. In recent years attention has shifted to making IE systems more portable to new domains by using some form of machine learning. These systems work moderately well on marked-up text (e.g. HTML-formatted text), but their performance on free text still needs improvement before they can be adopted more widely.

IE systems commonly use external knowledge sources in addition to the input text. For example, named entity recognizers often use name or location gazetteers, extensive catalogues of people and places such as the Getty Thesaurus of Geographic Names. In the bioinformatics domain it is common to draw on external knowledge bases for standard names of genes and proteins. A general purpose lexical ontology called WordNet is often used in IE systems. Sometimes formal domain ontologies are used as a knowledge source for IE, permitting systems to perform simple taxonomic inferences and resolve ambiguities in the classification of slot-fillers.

For example, an IE system for extracting basketball statistics from news reports on the web might have to interpret the sentence *Quincy played three games last weekend*, where the ambiguous subject *Quincy* might be either a player or a team. An ontology in which *Quincy* is identified as the name of a college basketball team would permit the ambiguity to be resolved. In other cases the template disambiguates the slot filler. Thus, in the basketball domain, *Washington* might refer to one of many possible players, teams or locations. If we know that the verb *won* requires an entity whose role in the taxonomy is a team, then we can resolve the ambiguity in the sentence *Washington won*.

4. The Architecture of linguistic and NLP systems

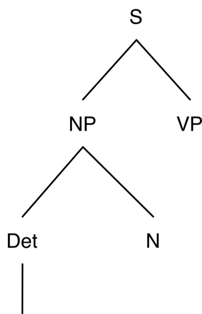
Within the approach to linguistic theory known as *generative grammar*, it is claimed that humans have distinct kinds of linguistic knowledge, organised into different modules: for example, knowledge of a language's sound structure (phonology), knowledge of word structure (morphology), knowledge of phrase structure (syntax), and knowledge of meaning (semantics). In a formal linguistic theory, each kind of linguistic knowledge is made explicit as different *module* of the theory, consisting of a collection of basic elements together with a way of combining them into complex structures. For example, a phonological module might provide a set of phonemes together with an operation for concatenating phonemes into phonological strings. Similarly, a syntactic module might provide labelled nodes as primitives together wih a mechanism for assembling them into trees. A set of linguistic primitives, together with some operators for defining complex elements, is often called *a level of representation*.

As well as defining modules, a generative grammar will prescribe how the modules interact. For example, well-formed phonological strings will provide the phonological content of words, and words will provide the terminal elements of syntax trees. Well-formed syntactic trees will be mapped to semantic representations, and contextual or pragmatic information will ground these semantic representations in some real-world situation.

As we indicated above, an important aspect of theories of generative grammar is that they are intended to model the linguistic knowledge of speakers and hearers; they are not intended to explain how humans actually process linguistic information. This is, in part, reflected in the claim that a generative grammer encodes the *competence* of an idealized native speaker, rather than the speaker's *performance*. A closely related distinction is to say that a generative grammar encodes *declarative* rather than *procedural* knowledge. As you might expect, computational linguistics has the crucial role of proposing procedural models of language. A central example is *parsing*, where we have to develop computational mechanisms which convert strings of words into structural representations such as syntax trees. Nevertheless, it is widely accepted that well-engineered computational models of language contain both declarative and procedural aspects. Thus, a full account of parsing will say how declarative knowledge in the form of a grammar and lexicon combines with procedural knowledge which determines how a syntactic analysis should be assigned to a given string of words. This procedural knowledge will be expressed as an *algorithm*: that is, an explicit recipe for mapping some input into an appropriate output in a finite number of steps.

A simple parsing algorithm for context-free gramars, for instance, looks first for a rule of the form $S \longrightarrow X_1 \ldots X_n$, and builds a partial tree structure. It then steps through the grammar rules one-by-one, looking for a rule of the form $X_1 \longrightarrow Y_1 \ldots Y_j$ which will expand the leftmost daughter introduced by the `S` rule, and further extends the partial tree. This process continues, for example by looking for a rule of the form $Y_1 \longrightarrow Z_1 \ldots Z_k$ and expanding the partial tree appropriately, until the leftmost node label in the partial tree is a lexical category; the parser then checks to see if the first word of the input can belong to the category. To illustrate, let's suppose that the first grammer rule chosen by the parser is `S` $\longrightarrow$ `NP VP` and the second rule chosen is `NP` $\longrightarrow$ `Det N`; then the partial tree will the one in Figure 2.

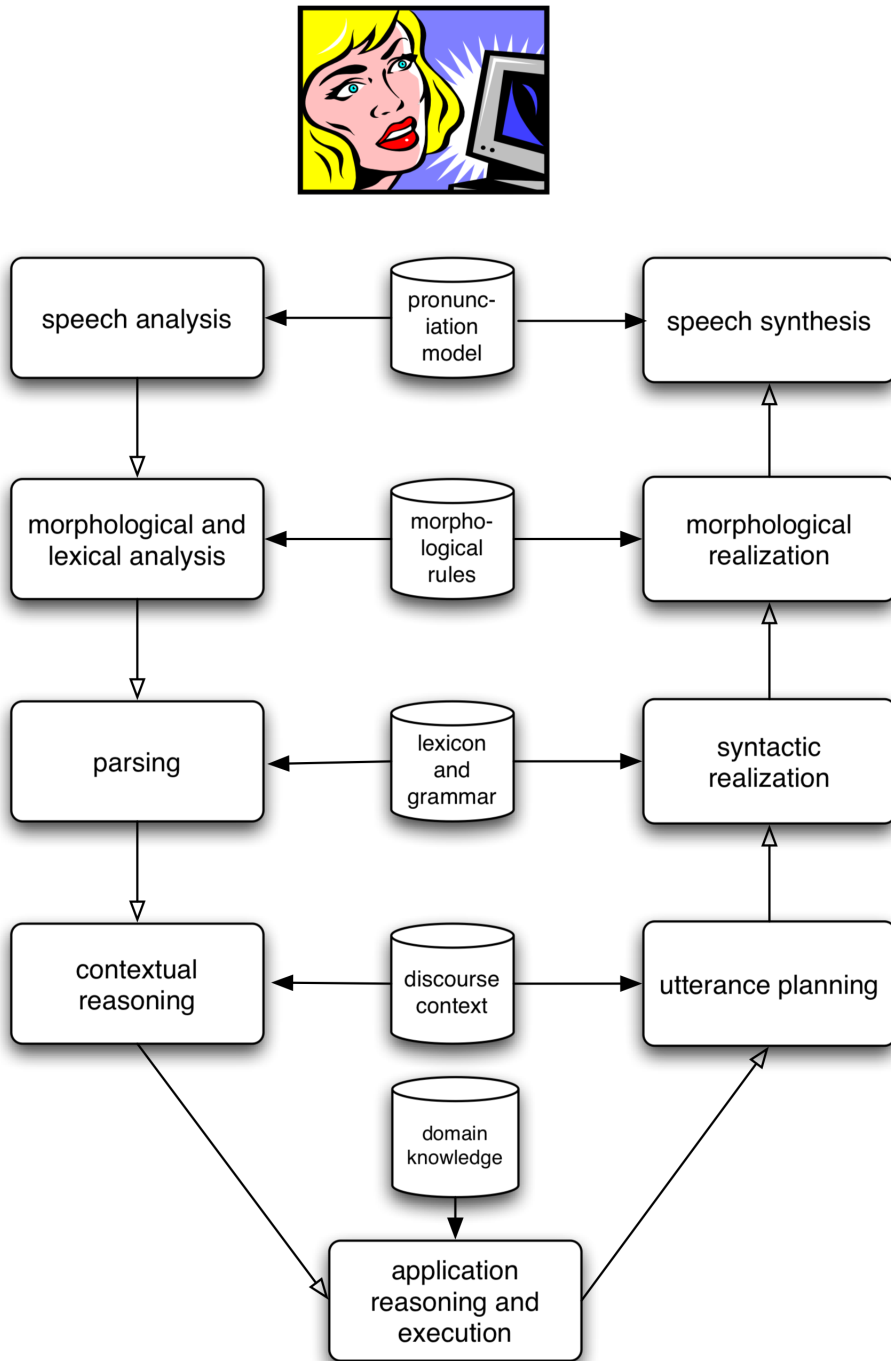
Figure 2. Partial Parse Tree



If we assume that the input string we are trying to parse is *the cat slept*, we will succeed in identifying *the* as a word which can belong to the category `Det`. In this case, the parser goes on to the next node of the tree, `N`, and next input word, *cat*. However, if we had built the same partial tree with an input string *did the cat sleep*, the parse would fail at this point, since *did* is not of category `Det`. The parser would throw away the structure built so far and look for an alternative way of going from the `S` node down to a leftmost lexical category (e.g., using a rule `S` $\longrightarrow$ `V NP VP`). The important point for now is not the details of this or other parsing algorithms; we discuss this topic much more fully in the chapter on parsing. Rather, we just want to illustrate the idea that an algorithm can be broken down into a fixed number of steps which produce a definite result at the end.

In Figure 3 we further illustrate some of these points in the context of a *spoken dialogue system*, such as our earlier example of an application that offers the user information about movies currently on show.

Figure 3. Architecture of Spoken Dialogue System



Down the lefthand side of the diagram we have shown a pipeline of some representative speech understanding *component*s. These map from speech input via syntactic parsing to some kind of meaning representation. Up the righthand side is an inverse pipeline of components for concept-to-speech generation. These components constitute the procedural aspect of the system's natural language processing. In the central column of the diagram are some representative declaratives aspects: the repositories of language-related information which are called upon by the processing components.

In addition to embodying the declarative/procedural distinction, the diagram also illustrates that linguistically motivated ways of modularizing linguistic knowledge are often reflected in computational systems. That is, the various components are organized so that the data which they exchange corresponds roughly to different levels of representation. For example, the output of the speech analysis component will contain sequences of phonological representations of words, and the output of the parser will be a semantic representation. Of course the parallel is not precise, in part because it is often a matter of practical expedience where to place the boundaries between different processing components. For example, we can assume that within the parsing component of Figure 3 there is a level of syntactic representation, although we have chosen not to expose this at the level of the system diagram. Despite such idiosyncracies, most NLP systems break down their work into a series of discrete steps. In the process of natural language understanding, these steps go from more concrete levels to more abstract ones, while in natural language production, the direction is reversed.

5. Learning NLP with the Natural Language Toolkit

The *Natural Language Toolkit (NLTK)* was originally created as part of a computational linguistics course in the Department of Computer and Information Science at the University of Pennsylvania in 2001. Since then it has been developed and expanded with the help of dozens of contributors. It has now been adopted in courses in dozens of universities, and serves as the basis of many research projects. In this section we will discuss some of the benefits of learning (and teaching) NLP using NLTK.

NLP is often taught within the confines of a single-semester course, either at advanced undergraduate level, or at postgraduate level. Unfortunately, it turns out to be rather difficult to cover both the theoretical and practical sides of the subject in such a short span of time. Some courses focus on theory to the exclusion of practical exercises, and deprive students of the challenge and excitement of writing programs to automatically process natural language. Other courses are simply designed to teach programming for linguists, and do not get past the mechanics of programming to cover significant NLP. NLTK was developed to address this very problem, making it feasible to cover a substantial amount of theory and practice within a single-semester course.

A significant fraction of any NLP course is made up of fundamental data structures and algorithms. These are usually taught with the help of formal notations and complex diagrams. Large trees and

charts are copied onto the board and edited in tedious slow motion, or laboriously prepared for presentation slides. A more effective method is to use live demonstrations in which those diagrams are generated and updated automatically. NLTK provides interactive graphical user interfaces, making it possible to view program state and to study program execution step-by-step. Most NLTK components have a demonstration mode, and will perform an interesting task without requiring any special input from the user. It is even possible to make minor modifications to programs in response to "what if" questions. In this way, students learn the mechanics of NLP quickly, gain deeper insights into the data structures and algorithms, and acquire new problem-solving skills.

NLTK supports assignments of varying difficulty and scope. In the simplest assignments, students experiment with existing components to perform a wide variety of NLP tasks. This may involve no programming at all, in the case of the existing demonstrations, or simply changing a line or two of program code. As students become more familiar with the toolkit they can be asked to modify existing components or to create complete systems out of existing components. NLTK also provides students with a flexible framework for advanced projects, such as developing a multi-component system, by integrating and extending NLTK components, and adding on entirely new components. Here NLTK helps by providing standard implementations of all the basic data structures and algorithms, interfaces to standard corpora, substantial corpus samples, and a flexible and extensible architecture. Thus, as we have seen, NLTK offers a fresh approach to NLP pedagogy, in which theoretical content is tightly integrated with application.

6. The Python Programming Language

NLTK is written in the *Python* language, a simple yet powerful scripting language with excellent functionality for processing linguistic data. Python can be downloaded for free from `www.python.org`. Here is a five-line Python program which takes text input and prints all the words ending in `ing`:

```
import sys                             # load the system library
for line in sys.stdin.readlines():     # for each line of input
    for word in line.split():          # for each word in the line
        if word.endswith('ing'):       # does the word end in 'ing'?
            print word                 # if so, print the word
```

This program illustrates some of the main features of Python. First, whitespace is used to *nest* lines of code, thus the line starting with `if` falls inside the scope of the previous line starting with `for`, so the `ing` test is performed for each word. Second, Python is *object-oriented*; each variable is an entity which has certain defined attributes and methods. For example, `line` is more than a sequence of characters. It is a string object that has a method (or operation) called `split` that we can use to break a line into its words. To apply a method to an object, we give the object name, followed by a period, followed by the method name. Third, methods have *arguments* expressed inside parentheses. For instance, `split` had no argument because we were splitting the string wherever there was white space. To split a string into sentences delimited by a period, we could

write `split('.')`. Finally, and most importantly, Python is highly readable, so much so that it is fairly easy to guess what the above program does even if you have never written a program before.

This readability of Python is striking in comparison to other languages which have been used for NLP, such as *Perl*. Here is a Perl program which prints words ending in `ing`:

```
while (<>) {                              # for each line of input
    foreach my $word (split) {            # for each word in a line
        if ($word =~ /ing$/) {            # does the word end in 'ing'?
            print "$word\n";              # if so, print the word
        }
    }
}
```

Like Python, Perl is a scripting language. However, it is not a real object-oriented language, and its syntax is obscure. For instance, it is difficult to guess what kind of entities are represented by: `<>`, `$`, `my`, and `split`. We agree that "it is quite easy in Perl to write programs that simply look like raving gibberish, even to experienced Perl programmers" (Hammond 2003:47). Having used Perl ourselves in research and teaching since the 1980s, we have found that Perl programs of any size are inordinately difficult to maintain and re-use. Therefore we believe Perl is not an optimal choice of programming language for linguists or for language processing. Several other languages are used for NLP, including Prolog, Java, LISP and C. In the appendix we have provided translations of our five-line Python program into these and other languages, and invite you to compare them for readability.

We chose Python as the implementation language for NLTK because it has a shallow learning curve, its syntax and semantics are transparent, and it has good string-handling functionality. As a scripting language, Python facilitates interactive exploration. As an object-oriented language, Python permits data and methods to be encapsulated and re-used easily. Python comes with an extensive standard library, including components for graphical programming, numerical processing, and web data processing.

NLTK defines a basic infrastructure that can be used to build NLP programs in Python. It provides:

- Basic classes for representing data relevant to natural language processing.
- Standard interfaces for performing tasks, such as tokenization, tagging, and parsing.
- Standard implementations for each task, which can be combined to solve complex problems.
- Extensive documentation, including tutorials and reference documentation.

7. The Design of NLTK

NLTK was designed with six requirements in mind. First, NLTK is *easy to use*. The primary

purpose of the toolkit is to allow students to concentrate on building natural language processing systems. The more time students must spend learning to use the toolkit, the less useful it is. Second, we have made a significant effort to ensure that all the data structures and interfaces are *consistent*, making it easy to carry out a variety of tasks using a uniform framework. Third, the toolkit is *extensible*, easily accommodating new components, whether those components replicate or extend the toolkit's existing functionality. Moreover, the toolkit is organized so that it is usually obvious where extensions would fit into the toolkit's infrastructure. Fourth, the toolkit is designed to be *simple*, providing an intuitive and appealing framework along with substantial building blocks, for students to gain a practical knowledge of NLP without having to write mountains of code. Fifth, the toolkit is *modular*, so that the interaction between different components of the toolkit is minimized, and uses simple, well-defined interfaces. In particular, it should be possible to complete individual projects using small parts of the toolkit, without needing to understand how they interact with the rest of the toolkit. This allows students to learn how to use the toolkit incrementally throughout a course. Modularity also makes it easier to change and extend the toolkit. Finally, the toolkit is *well documented*, including nomenclature, data structures, and implementations.

Contrasting with these requirements are three non-requirements. First, while the toolkit provides a wide range of functions, it is not intended to be encyclopedic. There should be a wide variety of ways in which students can extend the toolkit. Second, while the toolkit should be efficient enough that students can use their NLP systems to perform meaningful tasks, it does not need to be highly optimized for runtime performance. Such optimizations often involve more complex algorithms, and sometimes require the use of C or C++, making the toolkit harder to install. Third, we have avoided clever programming tricks, since clear implementations are far preferable to ingenious yet indecipherable ones.

NLTK is organized into a collection of task-specific components. Each module is a combination of data structures for representing a particular kind of information such as trees, and implementations of standard algorithms involving those structures such as parsers. This approach is a standard feature of *object-oriented design*, in which components encapsulate both the resources and methods needed to accomplish a particular task.

The most fundamental NLTK components are for identifying and manipulating individual words of text. These include: `tokenizer`, for breaking up strings of characters into word tokens; `tokenreader`, for reading tokens from different kinds of corpora; `stemmer`, for stripping affixes from tokens, useful in some text retrieval applications; and `tagger`, for adding part-of-speech tags, including regular-expression taggers, n-gram taggers and Brill taggers.

The second kind of module is for creating and manipulating structured linguistic information. These components include: `tree`, for representing and processing parse trees; `featurestructure`, for building and unifying nested feature structures (or attribute-value matrices); `cfg`, for specifying free grammars; and `parser`, for creating parse trees over input text, including chart parsers, chunk parsers and probabilistic parsers.

Several utility components are provided to facilitate processing and visualization. These include:

`draw`, to visualize NLP structures and processes; `probability`, to count and collate events, and perform statistical estimation; and `corpus`, to access tagged linguistic corpora.

Finally, several advanced components are provided, mostly demonstrating NLP applications of machine learning techniques. These include: `clusterer`, for discovering groups of similar items within a large collection, including k-means and expectation maximisation; `classifier`, for categorising text into different types, including naive Bayes and maximum entropy; and `hmm`, for Hidden Markov Models, useful for a variety of sequence classification tasks.

A further group of components is not part of NLTK proper. These are a wide selection of third-party contributions, often developed as student projects at various institutions where NLTK is used, and distributed in a separate package called *NLTK Contrib*. Several of these student contributions, such as the Brill tagger and the HMM module, have now been incorporated into NLTK. Although these components are not maintained, they may serve as a useful starting point for future student projects.

In addition to software and documentation, NLTK provides substantial *corpus* samples, listed in Figure 4. Many of these can be accessed using the `corpus` module, avoiding the need to write specialized file parsing code before you can do NLP tasks.

Figure 4. Corpora and Corpus Samples Distributed with NLTK

Corpus	Compiler	Contents	Example Application
20 Newsgroups (selection)	Ken Lang and Jason Rennie	3 newsgroups, 4000 posts, 780k words	text classification
Brown Corpus	Nelson Francis and Henry Kucera	15 genres, 1.15M words, tagged	training and testing taggers, text classification, language modelling
CoNLL 2000 Chunking Data	Erik Tjong Kim Sang	270k words, tagged and chunked	training and testing chunk parsers
Genesis Corpus	Misc web sources	6 texts, 200k words, 6 languages	text classification
Project Gutenberg (selection)	Michael Hart, Gregory Newby, et al	14 texts, 1.7M words	text classification, language modelling
NIST 1999 Information Extraction (selection)	John Garofolo	63k words, newswire and named-entity SGML markup	training and testing named-entity recognizers
Levin Verb Index	Beth Levin	3k verbs with Levin classes	parser development

Lexicon Corpus		Words, tags and frequencies from Brown Corpus and WSJ	general purpose
Names Corpus	Mark Kantrowitz and Bill Ross	8k male and female names	text classification
PP Attachment Corpus	Adwait Ratnaparkhi	28k prepositional phrases, tagged as noun or verb modifiers	parser development
Roget's Thesaurus	Project Gutenberg	200k words, formatted text	word-sense disambiguation
SEMCOR	Vasile Rus and Rada Mihalcea	880k words, part-of-speech and sense tagged	word-sense disambiguation
SENSEVAL 2 Corpus	Ted Pedersen	600k words, part-of-speech and sense tagged	word-sense disambiguation
Stopwords Corpus	Martin Porter et al	2,400 stopwords for 11 languages	text retrieval
Penn Treebank (selection)	LDC	40k words, tagged and parsed	parser development
Wordnet 1.7	George Miller and Christiane Fellbaum	180k words in a semantic network	word-sense disambiguation, natural language understanding
Wordlist Corpus	OpenOffice.org et al	960k words and 20k affixes for 8 languages	

8. Further Reading

The Association for Computational Linguistics (ACL) is the peak professional body in NLP. Its journal and conference proceedings, approximately 10,000 articles, are available online with a full-text search interface, via `http://www.aclweb.org/anthology/`.

Several NLP systems have online interfaces that you might like to experiment with, e.g.:

- WordNet: `http://wordnet.princeton.edu/`
- Translation: `http://world.altavista.com/`

- ChatterBots: `http://www.loebner.net/Prizef/loebner-prize.html`
- Question Answering: `http://www.answerbus.com/`
- Summarisation: `http://tangra.si.umich.edu/clair/md/demo.cgi`

Useful websites with substantial information about NLP: `http://www.hltcentral.org/`, `http://www.lt-world.org/`, `http://www.aclweb.org/`, `http://www.elsnet.org/`. The ACL website contains an overview of computational linguistics, including copies of introductory chapters from recent textbooks, at `http://www.aclweb.org/archive/what.html`.

Acknowledgements: The dialogue example is taken from Bob Carpenter and Jennifer Chu-Carroll's ACL-99 Tutorial on Spoken Dialogue Systems; The terms high-church and low-church were used by Mark Liberman in a seminar at the University of Pennsylvania; the following people kindly provided program samples: Tim Baldwin, Trevor Cohn, Rod Farmer, Edward Ivanovic, Olivia March, and Lars Yencken.

8.1. NLP Textbooks and Surveys

This section will give a brief overview of other NLP textbooks, and field-wide surveys *(to be written).*

Recent textbooks: Manning & Schutze, Jurafsky & Martin. Older textbooks: Allen (1995), Charniak (1993), Grishman. Prolog-based: [Covington94], [GazdarMellish89], Pereira & Shieber. Mathematical foundations: Partee et al.

Recent field-wide surveys: Mitkov, Dale et al

Bibliography

[Covington94] Michael A. Covington, 1994, *Natural Language Processing for Prolog Programmers*, Prentice-Hall.

[GazdarMellish89] Gerald Gazdar, Chris Mellish, 1989, *Natural Language Processing in Prolog: An Introduction to Computational Linguistics*, Addison-Wesley.

[Dale00] Edited by Robert Dale, Edited by Hermann Moisl, Edited by Harold Somers, 2000, *Handbook of Natural Language Processing*, Marcel Dekker.

[Mitkov03] Edited by Ruslan Mitkov, 2003, *The Oxford Handbook of Computational Linguistics*, Oxford University Press.

9. Appendix: NLP in other Programming Languages

Earlier we explained the thinking that lay behind our choice of the Python programming language. We showed a simple Python program that reads in text and prints the words that end with `ing`. In this appendix we provide equivalent programs in other languages, so that readers can gain a sense of the appeal of Python.

Prolog is a logic programming language which has been popular for developing natural language parsers and feature-based grammars, given the inbuilt support for search and the *unification* operation which combines two feature structures into one. Unfortunately Prolog is not easy to use for string processing or input/output, as the following program code demonstrates:

```
main :-
    current_input(InputStream),
    read_stream_to_codes(InputStream, Codes),
    codesToWords(Codes, Words),
    maplist(string_to_list, Words, Strings),
    filter(endsWithIng, Strings, MatchingStrings),
    writeMany(MatchingStrings),
    halt.

codesToWords([], []).
codesToWords([Head | Tail], Words) :-
    ( char_type(Head, space) ->
        codesToWords(Tail, Words)
    ;
        getWord([Head | Tail], Word, Rest),
        codesToWords(Rest, Words0),
        Words = [Word | Words0]
    ).

getWord([], [], []).
getWord([Head | Tail], Word, Rest) :-
    (
        ( char_type(Head, space) ; char_type(Head, punct) )
    ->  Word = [], Tail = Rest
    ;   getWord(Tail, Word0, Rest), Word = [Head | Word0]
    ).

filter(Predicate, List0, List) :-
    ( List0 = [] -> List = []
    ;   List0 = [Head | Tail],
        ( apply(Predicate, [Head]) ->
            filter(Predicate, Tail, List1),
            List = [Head | List1]
        ;   filter(Predicate, Tail, List)
        )
```

```
    ).

endsWithIng(String) :- sub_string(String, _Start, _Len, 0, 'ing').

writeMany([]).
writeMany([Head | Tail]) :- write(Head), nl, writeMany(Tail).
```

Java is an object-oriented language incorporating native support for the internet, that was originally designed to permit the same executable program to be run on most computer platforms. Java is quickly replacing COBOL as the standard language for business enterprise software.

```
import java.io.*;
public class IngWords {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new
    InputStreamReader(
                 System.in));
        String line = in.readLine();
        while (line != null) {
            for (String word : line.split(" ")) {
                if (word.endsWith("ing"))
                    System.out.println(word);
            }
            line = in.readLine();
        }
    }
}
```

The C programming language is a highly-efficient low-level language that is popular for operating system software and for teaching the fundamentals of computer science.

```
#include <sys/types.h>
#include <regex.h>
#include <stdio.h>
#define BUFFER_SIZE 1024

int main(int argc, char **argv) {
     regex_t space_pat, ing_pat;
     char buffer[BUFFER_SIZE];
     regcomp(&space_pat, "[, \t\n]+", REG_EXTENDED);
     regcomp(&ing_pat, "ing$", REG_EXTENDED | REG_ICASE);

     while (fgets(buffer, BUFFER_SIZE, stdin) != NULL) {
         char *start = buffer;
         regmatch_t space_match;
         while (regexec(&space_pat, start, 1, &space_match, 0) == 0) {
             if (space_match.rm_so > 0) {
```

```
                regmatch_t ing_match;
                start[space_match.rm_so] = '\0';
                if (regexec(&ing_pat, start, 1, &ing_match, 0) == 0)
                    printf("%s\n", start);
            }
            start += space_match.rm_eo;
        }
    }
    regfree(&space_pat);
    regfree(&ing_pat);

    return 0;
}
```

LISP is a so-called functional programming language, in which all objects are lists, and all operations are performed by (nested) functions of the form `(function arg1 arg2 ...)`. Many of the earliest NLP systems were implemented in LISP.

```
(defpackage "REGEXP-TEST" (:use "LISP" "REGEXP"))
(in-package "REGEXP-TEST")

(defun has-suffix (string suffix)
  "Open a file and look for words ending in _ing."
  (with-open-file (f string)
     (with-loop-split (s f " ")
        (mapcar #'(lambda (x) (has_suffix suffix x)) s))))

(defun has_suffix (suffix string)
  (let* ((suffix_len (length suffix))
   (string_len (length string))
    (base_len (- string_len suffix_len)))
    (if (string-equal suffix string :start1 0 :end1 NIL :start2 base_len :end2 NIL)
        (print string))))

(has-suffix "test.txt" "ing")
```

Haskell is another functional programming language which permits a much more compact solution of our simple task.

```
module Main
  where main = interact (unlines.(filter ing).(map (filter isAlpha)).words)
    where ing = (=="gni").(take 3).reverse
```

Index

generative grammar, 8
a level of representation, 8
algorithm, 9
competence, 9
corpus, 16
declarative, 9
dialogue system, 3
empiricism, 5
formal language theory, 4
idealism, 5
Information Extraction, 7
named entities, 7
natural language processing, 2
Natural Language Toolkit (NLTK), 12
object-oriented design, 15
parsing, 9
performance, 9
Perl, 14
principle of compositionality, 5
procedural, 9
Python, 13
rationalism, 5
realism, 5
semantic grammar, 5
spoken dialogue system, 10
symbolic logic, 4
template, 7
Turing Test, 3