Chapter 1

INTRODUCTION

Teachers of introductory courses on computational linguistics are often faced with the challenge of setting up a practical programming component for student assignments and projects. This is a difficult task because different computational linguistics domains require a variety of different data structures and functions, and because a diverse range of topics may need to be included in the syllabus. A widespread practice is to employ multiple programming languages, where each language provides native data structures and functions that are a good fit for the task at hand. For example, a course might use Prolong for parsing, Perl for corpus processing, and a finite-state toolkit for morphological analysis. By relying on the built-in features of various languages, the teacher avoids having to develop a lot of software infrastructure.

An unfortunate consequence is that a significant part of such courses must be devoted to teaching programming languages. Further, many interesting projects span a variety of domains, and would require that multiple languages be bridged. For example, a student project that involved syntactic parsing of corpus data from a morphologically rich language might involve all three of the languages mentioned above: Perl for string processing a finite state toolkit for morphological analysis and Prolong for parsing. It is clear that these considerable overheads and shortcomings warrant a fresh approach.

Apart from the practical component, computational linguistics courses may also depend on software for in-class demonstrations. This con- text calls for highly interactive graphical user interfaces, making it possible to view program state (e.g. the chart of a chart parser), observe program execution step-by-step (e.g. execution of a finite-state machine), and even make minor modifications to programs in response to “what if” questions from the class. Because of these difficulties it is common to avoid live demonstrations, and keep classes for theoretical presentations only. Apart from being dull, this approach leaves students to solve important practical problems on their own, or to deal with them less efficiently in office hours. describe NLTK, the Natural Language Toolkit, which we have developed in conjunction with a course we have taught at the University of Pennsylvania. The Natural Language Toolkit is available under an open source license from http://nltk.sf.net/. NLTK runs on all platforms supported by Python, including Windows, OS X, Linux, and Unix.

Chapter 2

LITERATURE SURVEY

[1] Natural Language Processing using NLTK and WordNet

NLP

Natural language processing (NLP) is a field of computer science, artificial intelligence, and computational linguistics concerned with the interactions between computers and human (natural) languages [3].The goal of natural language processing is to allow that kind of interaction so that non-programmers can obtain useful information from computing systems. Natural language processing also includes the ability to draw insights from data contained in emails, videos, and other unstructured material. The various aspects of NLP include Parsing, Machine Translation, Language Modelling, Machine Learning, Semantic Analysis etc. In this paper we only focus on semantic analysis aspect of NLP using NLTK.

NLTK

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning.NLTK includes graphical demonstrations and sample data. NLTK is intended to support research and teaching in NLP or closely related areas, including empirical linguistics, cognitive science, artificial intelligence, information retrieval, and machine learning.We discuss how we can perform semantic analysis in NLP using NLTK as a platform for different corpora. Adequate representation of natural language semantics requires access to vast amounts of common sense and domain-specific world knowledge. We focus our efforts on using WordNet as a preferred corpora for using NLTK.

WORDNET

[5]Because meaningful sentences are composed of meaningful words, any system that hopes to process natural languages as people do must have information about words and their meanings. This information is traditionally provided through dictionaries, and machine-readable dictionaries are now widely available. But dictionary entries evolved for the convenience of human readers, not for machines. WordNet provides a more effective combination of traditional lexicographic information and modern computing. WordNet is

an online lexical database designed for use under program control. English nouns, verbs, adjectives, and adverbs are organized into sets of synonyms, each representing a lexicalized concept. Semantic relations link the synonym sets. Using NLTK and WordNet, we can form semantic relations and perform semantic analysis on texts, strings and documents.

PYTHON

Python is a dynamic object-oriented programming language. It offers strong support for integrating with other technologies, higher programmer productivity throughout the development life cycle, and is particularly well suited for large or complex projects with changing requirements. Python has a very shallow learning curve and an excellent online learning resource with support of innumerable libraries.

[2] NLTK: The Natural Language Toolkit (2009)

The Natural Language Toolkit (NLTK) was developed in conjunction with a computational linguistics course at the University of Pennsylvania in 2001 (Loper and Bird, 2002). It was designed with three pedagogical applications in mind: assignments, demonstrations, and projects.

Assignments. NLTK supports assignments of varying difficulty and scope. In the simplest assign- ments, students experiment with existing compo- nents to perform a wide variety of NLP tasks. As students become more familiar with the toolkit, they can be asked to modify existing components, or to create complete systems out of existing compo- nents.

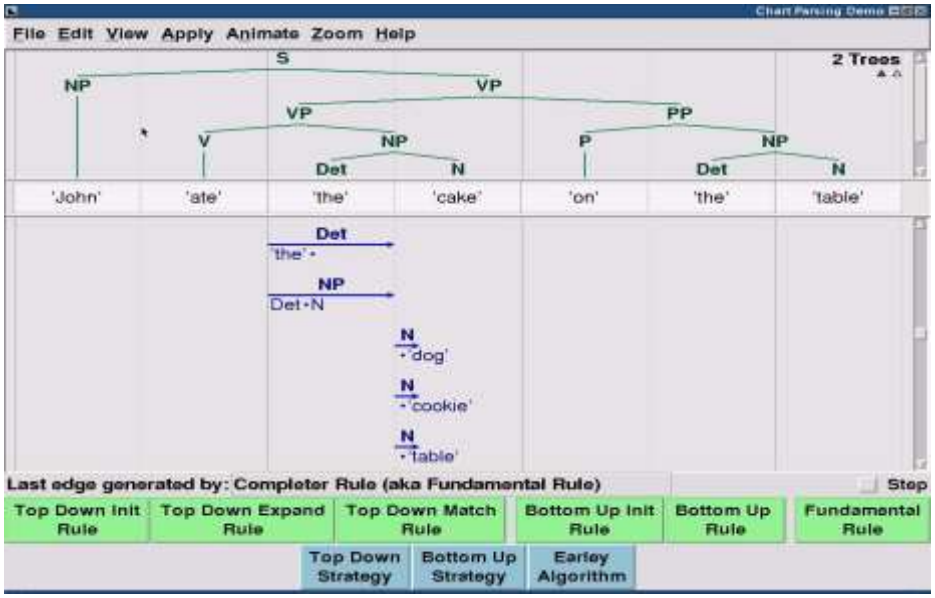
Demonstrations. NLTK's interactive graphical demonstrations have proven to be very useful for students learning NLP concepts. The demonstrations give a step-by-step execution of important algorithms, displaying the current state of key data structures. A screenshot of the chart parsing demonstration is shown in Figure 1.

Projects. NLTK provides students with a flexible framework for advanced projects. Typical projects might involve implementing a new algorithm, developing a new component, or implementing a new task.

We chose Python because it has a shallow learning curve, its syntax and semantics are transparent, and it has good string-handling functionality. As an interpreted 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 tools for graphical programming and numerical processing. The recently added

generator syntax makes it easy to create interactive implementations of algorithms (Loper, 2004; Rossum, 2003a; Rossum, 2003b).

Figure 2.1: model working



[3] Using NLTK for educational and scientific purposes

Natural language processing (NLP) is an extremely active field of research, which attracts a lot of students every year. It gives the possibility to study human language from the applied side, not just theoretically, and to try to solve some of the tasks considering human language. However, the most important reason for choosing NLP as an area of study, is the variety of very interesting problems with no general solutions. For example, the original problem of machine translation still remains one of the hardest to solve, even after twenty years of discussing and active research. Python and the Natural Language Toolkit (NLTK) allow any programmer, even a beginner, to get acquainted with NLP tasks easily without spending too much time on studying or gathering resources. The aim of this paper is to provide valuable proof and examples, which show how necessary the NLTK is for the course of Computational Linguistics at the university and for researchers in the field of natural language processing.

[4] NLTK: The Natural Language Toolkit (2002)

Teachers of introductory courses on computational linguistics are often faced with the challenge of setting up a practical programming component for student assignments and

projects. This is a difficult task because different computational linguistics domains require a variety of different data structures and functions, and because a diverse range of topics may need to be included in the syllabus. A widespread practice is to employ multiple programming languages, where each language provides native data structures and functions that are a good fit for the task at hand. For example, a course might use Prolog for parsing, Perl for corpus processing, and a finite-state toolkit for morphological analysis. By relying on the built-in features of various languages, the teacher avoids having to develop a lot of software Infrastructure. An unfortunate consequence is that a significant part of such courses must be devoted to teaching programming languages. Further, many interesting projects span a variety of domains, and would require that multiple languages be bridged. For example, a student project that involved syntactic parsing of corpus data from a morphologically rich language might involve all three of the languages mentioned above: Perl for string processing; a finite state toolkit for morphological analysis; and Prolog for parsing. It is clear that these considerable overheads and shortcomings warrant a fresh approach. Apart from the practical component, computational linguistics courses may also depend on software for in-class demonstrations. This context calls for highly interactive graphical user interfaces, making it possible to view program state (e.g. the chart of a chart parser), observe program execution step-by-step (e.g. execution of a finite-state machine), and even make minor modifications to programs in response to "what if" questions from the class. Because of these difficulties it is common to avoid live demonstrations, and keep classes for theoretical presentations only. Apart from being dull, this approach leaves students to solve important practical problems on their own, or to deal with them less efficiently in office hours. In this paper we introduce a new approach to the above challenges, a streamlined and flexible way of organizing the practical component of an introductory computational linguistics course. We describe NLTK, the Natural Language Toolkit, which we have developed in conjunction with a course we have taught at the University of Pennsylvania. The Natural Language Toolkit is available under an open source license from http://nltk.sf.net/. NLTK runs on all platforms supported by Python, including Windows, OS X, Linux, and Unix.

Chapter 3

SCOPE AND SIGNIFICANCE

3.1 NATIONAL SIGNIFICANCE

Natural Language Processing is often taught within the confines of a single-semester course at advanced undergraduate level or postgraduate level. Many instructors have found that it is 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 language. Other courses are simply designed to teach programming for linguists, and do not manage to cover any significant NLP content. NLTK was originally developed to address this problem, making it feasible to cover a substantial amount of theory and practice within a single-semester course, even if students have no prior programming experience.

3.2 INTERNATIONAL SIGNIFICANCE

A significant fraction of any NLP syllabus deals with algorithms and data structures. On their own these can be rather dry, but NLTK brings them to life with the help of interactive graphical user interfaces that make it possible to view algorithms step-by-step. Most NLTK components include a demonstration that performs an interesting task without requiring any special input from the user. This approach leaves students to solve important practical problems on their own, or to deal with them less efficiently in office hours.

3.3 SCOPE

Simplicity: To provide an intuitive framework along with substantial building blocks, giving users a practical knowledge of NLP without getting bogged down in the tedious house-keeping usually associated with processing annotated language data

Consistency: To provide a uniform framework with consistent interfaces and data structures, and easily-guessable method names

Extensibility: To provide a structure into which new software modules can be easily accommodated, including alternative implementations and competing approaches to the same task

Modularity: To provide components that can be used independently without needing to understand the rest of the toolkit.

Chapter 4

METHODOLOGY

The Natural Language Toolkit is a collection of program modules, data sets, tutorials and exercises, covering symbolic and statistical natural language processing. NLTK is written in Python and distributed under the GPL open source license.

NLTK is implemented as a large collection of minimally interdependent modules, organized into a shallow hierarchy [7]. A set of core modules defines basic data types that are used throughout the toolkit. The remaining modules are task modules, each devoted to an individual natural language processing task. For example, the nltk.parser module encompasses to the task of parsing, or deriving the syntactic structure of a sentence; and the nltk.tokenizer module is devoted to the task of tokenizing, or dividing a text into its constituent parts.

4.1 NLTK CORPORA

NLTK incorporates several useful text corpora that are used widely for NLP. Some of them are as follows:

Brown Corpus: The Brown Corpus of Standard American English is the first general English corpus that could be used in computational linguistic processing tasks. This corpus consists of one million words of American English texts printed in 1961. For the corpus to represent as general a sample of the English language as possible, 15 different genres were sampled such as Fiction, News and Religious text. Subsequently, a POS-tagged version of the corpus was also created with substantial manual effort.

Gutenberg Corpus: The Gutenberg Corpus is a collection of 14 texts chosen from Project Gutenberg - the largest online collection of free e-books. The corpus contains a total of 1.7 million words.

Stopwords Corpus: Apart from regular content words, there is another class of words called stop words that perform important grammatical functions but are unlikely to be interesting by themselves, such as prepositions, complementizers and determiners. NLTK comes bundled with the Stopwords Corpus - a list of 2400 stop words across 11 different languages (including English). Apart from these corpora which are shipped with NLTK we can use intelligent sources of data like WordNet or Wikipedia.

4.2 MODULES OF NLTK

Parsing Modules

The parser module defines a high-level interface for creating trees that represent the structures of texts [7]. The chunkparser module defines a sub-interface for parsers that identify non overlapping linguistic groups (such as base noun phrases) in unrestricted text.Four modules provide implementations for these abstract interfaces.

The srparser module implements a simple shift-reduce parser. The chartparser module defines a flexible parser that uses a chart to record hypotheses about syntactic constituents. The pcfgparser module provides a variety of different parsers for probabilistic grammars. And the rechunkparser module defines a transformational regular- expression based implementation of the chunk parser interface.

Tagging Modules

The tagger module defines a standard interface for extending each token of a text with additive information, such as its part of speech or its WordNet synset tag. It also provides several different implementations for this interface.

Finite State Automata

The fsa module provides an interface for creating automata from regular expressions.

Type Checking

Debugging time is an important factor in the toolkit's ease of use. To reduce the amount of time students must spend debugging their code, a type checking module is provided, which can be used to ensure that functions are given valid arguments. However, when efficiency is an issue, type checking can be disabled which causes no performance penalty.

Language processing task	NLTK modules	Functionality
Accessing corpora	nltk.corpus	standardized interfaces to corpora and lexicons
String processing	nltk.tokenize, nltk.stem	tokenizers, sentence tokenizers, stemmers
Collocation discovery	nltk.collocations	t-test, chi-squared, point-wise mutual information
Part-of-speech tagging	nltk.tag	n-gram, backoff, Brill, HMM, TnT
Classification	nltk.classify, nltk.cluster	decision tree, maximum entropy, naive Bayes, EM, k-means
Chunking	nltk.chunk	regular expression, n-gram, named-entity
Parsing	nltk.parse	chart, feature-based, unification, probabilistic, dependency
Semantic interpretation	nltk.sem, nltk.inference	lambda calculus, first-order logic, model checking
Evaluation metrics	nltk.metrics	precision, recall, agreement coefficients
Probability and estimation	nltk.probability	frequency distributions, smoothed probability distributions
Applications	nltk.app, nltk.chat	graphical concordancer, parsers, WordNet browser, chatbots
Linguistic fieldwork	nltk.toolbox	manipulate data in SIL Toolbox format

Figure 4.1 Language processing tasks and corresponding nltk modules with examples of functionality

4.3 MODULES

The toolkit is implemented as a collection of independent modules, each of which defines a specific data structure or task. A set of core modules defines basic data types and processing systems that are used throughout the toolkit. The token module provides basic classes for processing individual elements of text, such as words or sentences. The tree module defines data structures for representing tree structures over text, such as syntax trees and morphological trees. The probability module implements classes that encode frequency

distributions and probability distributions, including a variety of statistical smoothing techniques. The remaining modules define data structures and interfaces for performing specific NLP tasks. This list of modules will grow over time, as we add new tasks and algorithms to the toolkit.

Parsing Modules

The parser module defines a high-level inter- face for producing trees that represent the structures of texts. The chunkparser module defines a sub-interface for parsers that identify non-overlapping linguistic groups (such as base noun phrases) in unrestricted text Four modules provide implementations for these abstract interfaces. The srparser module implements a simple shift-reduce parser. The chartparser module defines a flexible parser that uses a chart to record hypotheses about syntactic constituents. The pcfgparser module provides a variety of different parsers for probabilistic grammars. And the rechunkparser module defines a transformational regular-expression based implementation of the chunk parser interface.

Tagging Modules

The tagger module defines a standard interface for augmenting each token of a text with supple- mentary information, such as its part of speech or its WordNet synset tag; and provides several different implementations for this interface.

Finite State Automata

The fsa module defines a data type for encod- ing finite state automata; and an interface for creating automata from regular expressions.

Type Checking

Debugging time is an important factor in the toolkit's ease of use. To reduce the amount of time students must spend debugging their code, we provide a type checking module, which can be used to ensure that functions are given valid arguments. The type checking module is used by all of the basic data types and processing classes. Since type checking is done explicitly, it can slow the toolkit down. However, when efficiency is an issue, type checking can be easily turned off; and with type checking is disabled, there is no performance penalty.

Visualization

Visualization modules define graphical interfaces for viewing and manipulating data structures, and graphical tools for experimenting with NLP tasks. The draw.tree module

provides a simple graphical inter- face for displaying tree structures. The draw.tree edit module provides an interface for building and modifying tree structures. An unfortunate consequence is that a significant part of such courses must be devoted to teaching programming languages. Further, many interesting projects span a variety of domains, and would require that multiple languages be bridged. For example, a student project that involved syntactic parsing of corpus data from a morphologically rich language might involve all three of the languages mentioned above: Perl for string processing a finite state toolkit for morphological analysis and Prolong for parsing. It is clear that these considerable overheads and shortcomings warrant a fresh approach.

Apart from the practical component, computational linguistics courses may also depend on software for in-class demonstrations. This con- text calls for highly interactive graphical user interfaces, making it possible to view program state (e.g. the chart of a chart parser), observe program execution step-by-step (e.g. execution of a finite-state machine), and even make minor modifications to programs in response to "what if" questions from the class. Because of these difficulties it is common to avoid live demonstrations, and keep classes for theoretical presentations only. The draw plot graph module can be used to graph mathematical functions. The draw fsa module provides a graphical tool for displaying and simulating finite state automata. The draw chart module provides an interactive graphical tool for experimenting with chart parsers. The visualization modules provide interfaces for interaction and experimentation; they do not directly implement NLP data structures or tasks. Simplicity of implementation is therefore less of an issue for the visualization modules than it is for the rest of the toolkit.

Text Classification

The classifier module defines a standard interface for classifying texts into categories. This interface is currently implemented by two modules. The classifier.naivebayes module defines a text classifier based on the Naive Bayes assumption. The classifier.maxent module defines the maximum entropy model for text classification, and implements two algorithms for training the model: Generalized Iterative Scaling and Improved Iterative Scaling. The classifier feature module provides a standard encoding for the information that is used to make decisions for a particular classification task. This standard encoding allows students to experiment with the differences between different text classification algorithms, using identical feature sets. The classifier feature selection module defines a standard interface for choosing which features are

relevant for a particular classification task. Good feature selection can significantly improve classification performance.

Working with NLTK.

The Natural Language Toolkit (NLTK) is a mature open source platform for building Python programs to work with human language data Figure 1 shows the five-step processing pipeline. NLTK provides the basic pieces to accomplish those steps, each one with different options and degrees of freedom. Starting with an unstructured body of words (i.e., raw text), we want to obtain sentences (the first step of abstraction on top of simple words) and have access to each word independently (without
loosing its context or relative positioning to its sentence). This process is known as *tokenization* and it is complicated by the possibility of a single word being associated with multiple token types. Consider, for example, the sentence: “These prerequisites are known as (computer) system requirements and are often used as a guideline as opposed to an absolute rule.” The abbreviated script of Python code is as follows

```
text = "These prerequisites are known as (computer)
system requirements and are often used as a
guideline as opposed to an absolute rule."
tokens = nltk.word_tokenize(my_string)
print tokens
=>
['These', 'prerequisites', 'are', 'known', 'as',
'(', 'computer', ')', 'system', 'requirements',
'and', 'are', 'often', 'used', 'as', 'a',
'guideline', 'as',' opposed', 'to', 'an',
'absolute', 'rule','.']
```

The result of this script is an array that contains all the text’s tokens, each token being a word or a punctuation character. After we have obtained an array with each token (i.e., word) from the original text, we may want to normalize these tokens. This means: (1) Converting all letters to lower case, (2) Making all plural words singular ones, (3) Removing *ing* endings from verbs, (4) Making all verbs be in present tense, and (5) Other similar actions to remove meaningless differences between words. In NLP jargon, the latter is known as *stemming*, in reference to a

process that strips off affixes and leaves you with a stem [6]. NLTK provides us with higher level *stemmers* that incorporate complex rules to deal with the difficult problem of stemming. The Porter stemmer that uses the algorithm presented in [7], the Lancaster stemmer, based on [8], or the built in lemmatizer – Stemming is also known as *lemmatization*, referencing the search of the *lemma* of which one is looking an inflected form [6] – found in WordNet. Wordnet is an open lexical database of English maintained by Princeton University [9]. The latter is considerably slower than all the other ones, since it has to look for the potential stem into its database for each token.

The next step is to identify what role each word plays on the sentence: a noun, a verb, an adjective, a pronoun,
preposition, conjunction, numeral, article and interjection [10].This process is known as *part of speech tagging*, or simply *POS tagging* [11]. On top of POS tagging we can identify the *entities*. We can think of these *entities* as "multiple word nouns" or objects that are present in the text. NLTK provides an interface for tagging each token in a sentence with supplementary information such as its part of speech. Several taggers are included, but an *off-the-shelf* one is available, based on the Penn Treebank tagset [12]. The following listing shows how simple is to perform a basic part of speech tagging.

```
my_string = "When I work as a senior systems
engineer, I truly enjoy my work."
tokens = nltk.word_tokenize(my_string)
print tokens
tagged_tokens = nltk.pos_tag(tokens)
print tagged_tokens
=>
[('When', 'WRB'), ('I', 'PRP'), ('work', 'VBP'),
('as', 'RB'), ('a', 'DT'), ('senior', 'JJ'),
('systems', 'NNS'), ('engineer', 'NN'), (',', ','),
('I', 'PRP'), ('truly', 'RB'), ('enjoy', 'VBP'),
('my', 'PRP$'), ('work', 'NN'), ('.', '.')]
```

The first thing to notice from the output is that the tags are two or three letter codes. Each one represent a lexical category or part of speech. For instance, WRB stands for *Wh-adverb*, including *how*, *where*, *why*, etc. PRP stands for *Personal pronoun*; *RB* for *Adverb*; *JJ* for

Adjective, *VBP* for *Present verb tense*, and so forth [13]. These categories are more detailed than presented in [10], but they can all be traced back to those ten major categories. It is important to note the the possibility of one-to-many relationships between a word and the tags that are possible. For our test example, the word *work* is first classified as a verb, and then at the end of the sentence, is classified as a noun, as expected. Moreover, we found two nouns (i.e. objects), so we can affirm that the text is saying something about *systems*, *an engineer* and *a work*. But we know more than that. We are not only referring to *an engineer*, but to a *systems engineer*, and not only a *systems engineer*, but a *senior systems engineer*. This is our *entity* and we need to *recognize* it from the text (thus the sectionname). In order to do this, we need to somehow tag groups of words that represent an entity (e.g., sets of nouns that appear in succession: *('systems', 'NNS'), ('engineer', 'NN')*). NLTK offers regular expression processing support for identifying groups of tokens, specifically noun phrases, in the text. The rules for the parser are specified defining *grammars*, including patterns, known as *chunking*, or excluding patterns, known as *chinking*. As a case in point, Figures 2 and 3 show the tree structures that are generated when chunking and chinking are applied to our test sentence.

Chapter 5

RESULTS

5.1 CALCULATING SEMANTIC SIMILARITY AND RELATEDNESS

Our approach to calculate similarity is as follows:

1. Remove the stopwords from both the sentences using a database for stopwords in WordNet.
2. Tokenize the sentences without stopwords
3. Compare each word of 1st sentence with the database of given words in 2nd sentence from WordNet.
4. Each comparison returns us a score of similarity.
5. Average out the score for the whole sentence.
6. Python code for doing the above is given below:

```
stop = stopwords.words('english')
goodwords= [i for i in sentences.split() if i not in stop]
goodwords1= [i for i in target_sentence.split() if i not in
stop]
m=0
n=0
l=[]
fl=[]
for m,p in enumerate(goodwords):
for n,q in enumerate (goodwords1):
xx = wn.synsets(p)
y = wn.synsets(q)[0]
del l[:]
for x in xx:
if (x.wup_similarity(y))==None:
l.append(0)
else:
l.append(x.wup_similarity(y))
try:
```

```
fl.append(max(l))
except:
fl.append(0)
score=sum(fl)/len(fl)
```

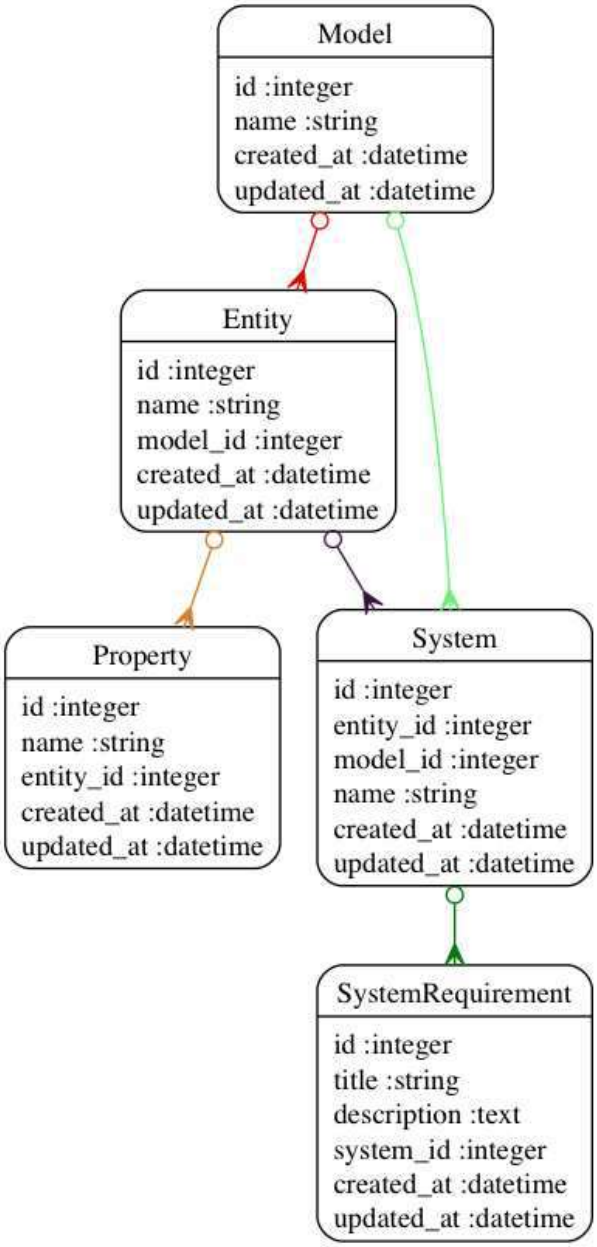


Figure 5.1: Diagram of the application models.

We have exercised our ideas in a prototype application, step-by-step development of a simplified aircraft ontology model and a couple of associated textual requirements. The software system requires two inputs: (1) An ontology model that defines what we are designing, and (2) A system defined by its requirements. We manage a flattened (i.e., tabular) version of a simplified aircraft ontology. Figure 5 shows the aircraft model we are going to use.

Figure 5.2 Simplified ontology model for an aircraft.

This simple ontology suggests usage of a hierarchical model structure, with aircraft properties also being represented by their own specialized ontology models. Second, it makes sense to include a property in the model even if its value isn't set. Naturally, this lacks valuable information, but it does give us the knowledge that that particular property is part of the model, so we can check for its presence. The next step is to create a system model and link it to the ontology. We propose a one-to-one association relationship between the system and an ontology, with more complex relationships handled through hierarchical structures in ontologies. This assumption simplifies development because when we are creating a system we only need to refer to one ontology model and one entity. The design of the system is specified through *textual system requirements*. To enter them we need a system, a title and a description. Figure 6 shows, for example, all the system Requirements for the system *UMDBus 787*. Notice that each requirement has a title and a description, and it belongs to a specific system. The prototype software has views (details not provided here) to highlight connectivity relationships between the requirements, system model (in this case, a simplified model of a UMDBus 787), and various aircraft ontolology models. The analysis and validation actions match the system's properties taken from its ontology model against information provided in the requirements.

Chapter 6

CONCLUSION AND FUTURE ENHANCEMENT

NLTK provides a simple, extensible, uniform framework for assignments, projects, and class demonstrations. It is well documented, easy to learn, and simple to use. We hope that NLTK will allow computational linguistics classes to include more hands-on experience with using and building NLP components and systems.

NLTK is unique in its combination of three factors. First, it was deliberately designed as courseware and gives pedagogical goals primary status. Second, its target audience consists of both linguists and computer scientists, and it is accessible and challenging at many levels of prior computational skill. Finally, it is based on an object-oriented scripting language supporting rapid prototyping and literate programming. We plan to continue extending the breadth of materials covered by the toolkit. We are currently working on NLTK modules for Hidden Markov Models, language modeling, and tree adjoining grammars. We also plan to increase the number of algorithms implemented by some existing modules, such as the text classification module. Finding suitable corpora is a prerequisite for many student assignments and projects. We are therefore putting together a collection of corpora containing data appropriate for every module defined by the toolkit.NLTK is an open source project, and we welcome any contributions. Readers who are inter- ested in contributing to NLTK, or who have suggestions for improvements, are encouraged to contact the authors.

REFERENCES

[1] **Natural Language Processing using NLTK and WordNet**, Alabhya Farkiya et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 6 (6) , 2015, 5465-5469 www.ijcsit.com

[2] **NLTK: The Natural Language Toolkit(2011)**, Steven Bird , University of Melbourne, Edward Loper, University of Pennsylvania sb@csse.unimelb.edu.au http://arXiv.org/abs/ cs/0205028.

[3] **Using NLTK for educational and scientific purposes**, Mykhailo Lobur, Andriy Romanyuk, Mariana Romanyshyn, CADSM'2011, 23-25 February, 2011, Polyana-Svalyava (Zakarpattya), UKRAINE

[4] **NLTK: The Natural Language Toolkit,** Edward Loper and Steven Bird Department of Computer and Information Science University of Pennsylvania, Philadelphia, PA 19104-6389, USA

[5] **NLTK BOOK**: https://www.nltk.org/book/