Natural Language Processing of Textual Requirements

Andres Arellano
Government of Chile,
Santiago, Chile
Email: andres.arellano@gmail.com

Edward Carney
Lockheed Martin,
College Park, MD 20742, USA
Email: edward.carney@lmco.com

Mark A. Austin
Department of Civil Engineering,
University of Maryland,
College Park, MD 20742, USA
Email: austin@isr.umd.edu

***Abstract*—Natural language processing (NLP) is the application of automated parsing and machine learning techniques to analyze standard text. Applications of NLP to requirements engineering include extraction of ontologies from a requirements specification, and use of NLP to verify the consistency and/or completion of a requirements specification. This work-in-progress paper describes a new approach to the interpretation, organization and management of textual requirements through the use of application-specific ontologies and natural language processing. We also design and exercise a prototype software tool that implements the new framework on a simplified model of an aircraft.**

Keywords-Systems Engineering; Ontologies; Natural Language Processing; Requirements; Rule Checking.

I. Introduction

Model-based systems engineering development is an approach to systems-level development in which the focus and primary artifacts of development are models, as opposed to documents. As engineering systems become increasingly complex the need for automation arises [1]. A key element of required capability is an ability to identify and manage requirements during the early phases of the system design process, where errors are cheapest and easiest to correct. While engineers are looking for semi-formal and formal models to work with, the reality remains that many large-scale projects begin with hundreds – sometimes thousands – of pages of textual requirements, which may be inadequate because they are incomplete, under specified, or perhaps ambiguous. State-of-the art practice involves the manual translation of text into a semi-formal format (suitable for representation in a requirements database). A second key problem is one of completeness. For projects defined by hundreds/thousands of textual requirements, how do we know a system description is complete and consistent? The motivating tenet of our research is that supporting tools that make use of computer processing could significantly help software engineers to validate the completeness of system requirements. Given a set of textual descriptions of system requirements, we could analyze them making use of natural language processing tools, extracting the objects or properties that are referenced within the requirements. Then, we could match these properties against a defined ontology model corresponding to the domain of this particular requirement. This would throw alerts in case of lacking requirements for some properties.

II. Project Objectives

Significant work has been done to apply natural language processing (NLP) to the domain of requirements engineering [2] [3] [4]. Applications range from using NLP to extract ontologies from a requirements specification, to using NLP to verify the consistency and/or completion of a requirements specification. This work-in-progress paper outlines a framework for using NLP to assist in the requirements decomposition process. Our research objectives are to use modern language processing tools to scan and tag a set of requirements, and offer support to systems engineers in their task of defining and maintaining a comprehensive, valid and accurate body of requirements. Section III describes two aspects of our work in progress: (1) Working with NLTK, and (2) Integration of NLP with ontologies and textual requirements. A simple aircraft application is presented in Section IV. Section V covers the conclusions and directions for future work.

III. Work in Progress

Topic 1. Working with NLTK. The Natural Language Toolkit (NLTK) is a mature open source platform for building Python programs to work with human language data [5].

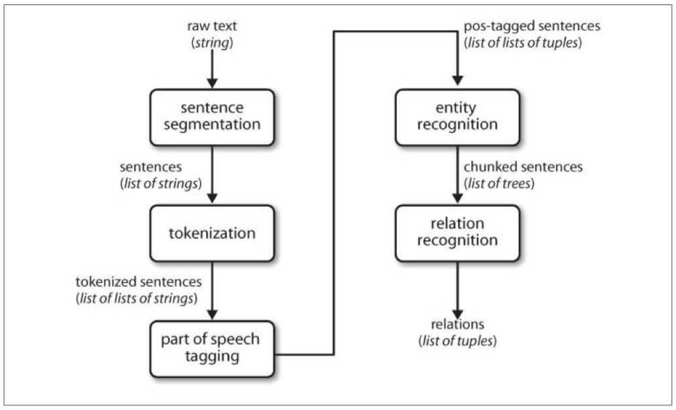


Figure 1. Information extraction system pipeline architecture.

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

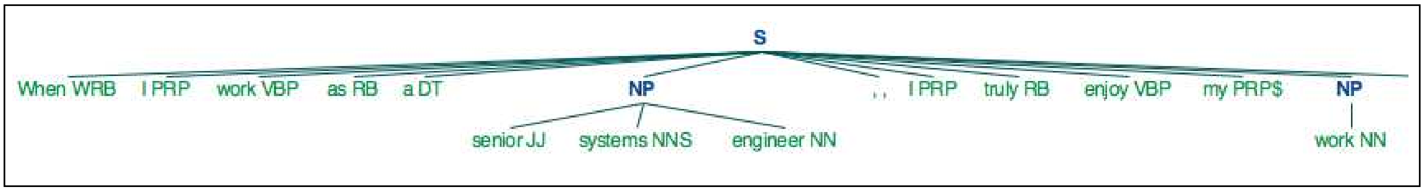


Figure 2. Output from building a chunking grammar.

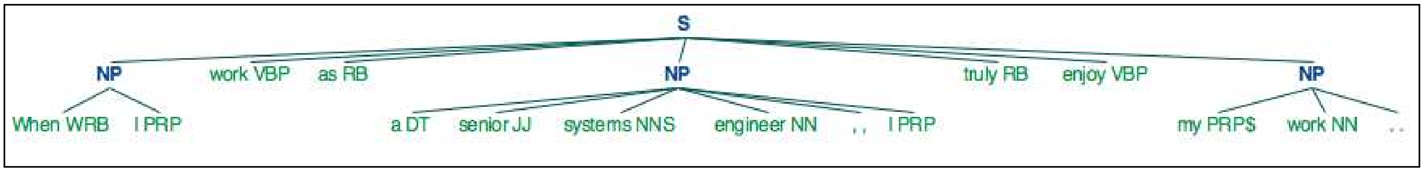


Figure 3. Output from the example on chinking.

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 section

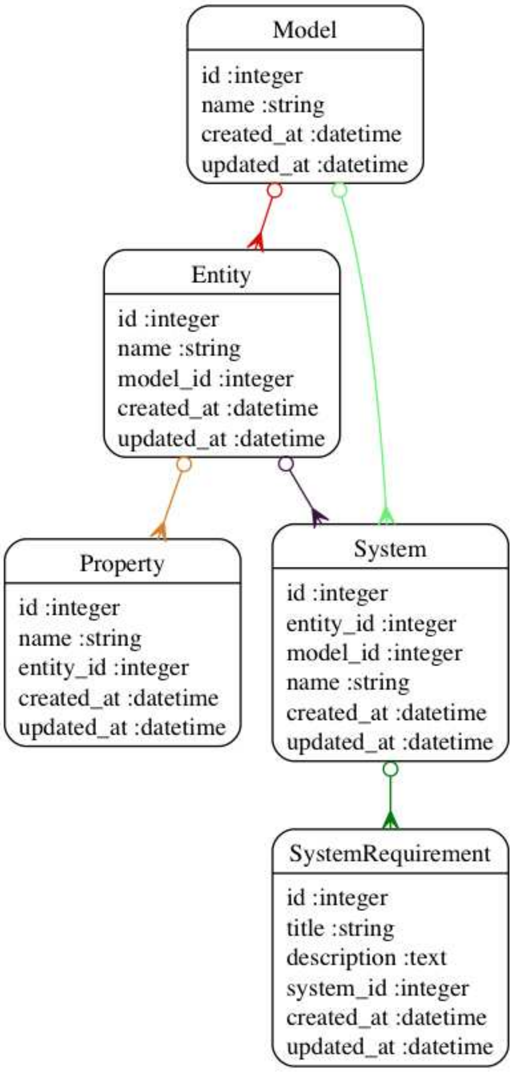


Figure 4. UML diagram of the application models.

name). 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.

Topic 2. Integration of NLP with Ontologies and Textual Requirements. In order to provide a platform for the integration of natural language processing, ontologies and systems requirements, and to give form to our project, we built *TextReq Validation*, a web based software that serves as a proof of concept for our objectives. The software stores ontology models in a relational database (i.e., tables), as well as a system with its requirements. It can do a basic analysis on these requirements and match them against the model's properties, showing which ones are covered and which ones are not. The software has two main components: The web application that provides the user interfaces, handles the business logic, and manages the storage of models and systems. This component was built using Ruby on Rails (RoR), a framework to create web applications following the Model View Controller pattern [14]. The views and layouts are supported by the front-end framework Bootstrap [15]; These scripts are written using Python. Figure 4 is a UML diagram showing all the models. The *models* corresponding to the MVC architecture of the web application, reveal the simple design used to represent an Ontology and a System. The first one consists of a Model – named after an Ontology Model, and not because it is a MVC model – that has many Entities. The Entities, in turn, have many Properties. The latter is even simpler, consisting of only a *System* that has many *System Requirements*. Most of the business logic resides in the models. Notice, in particular, system-level interpretation of results from the natural language processing.

IV. Simple Aircraft Application

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.

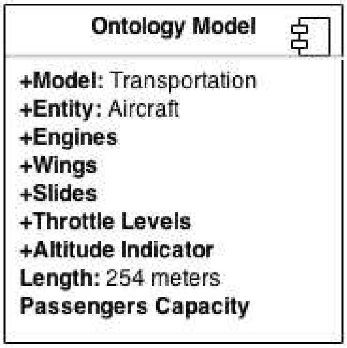
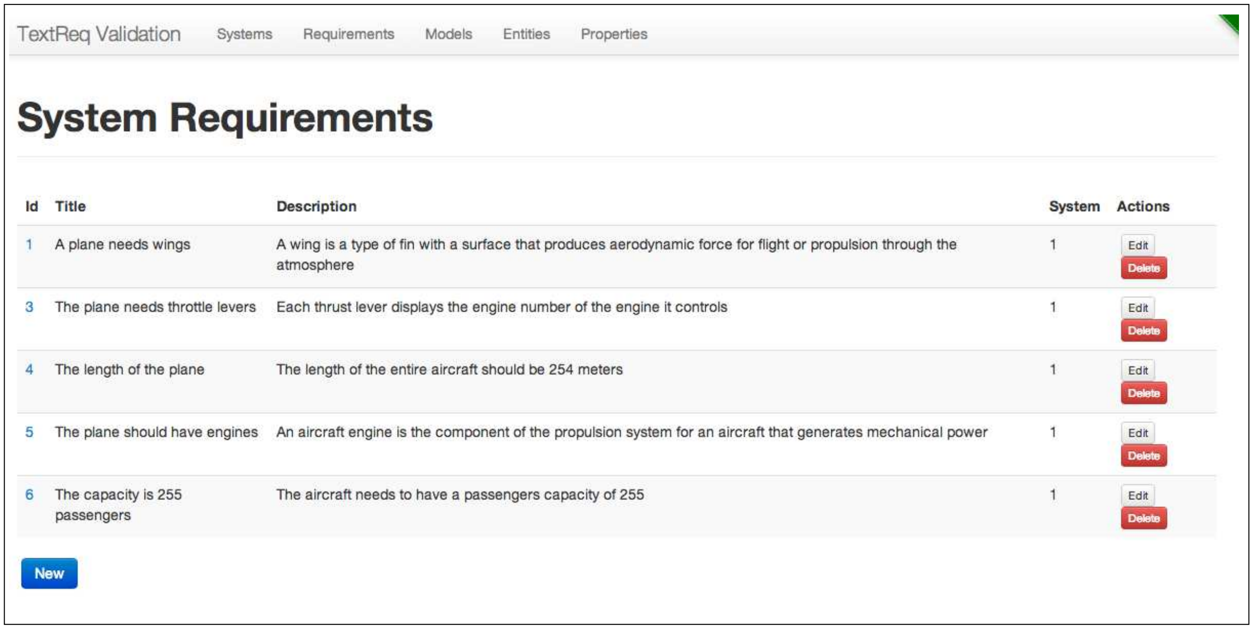


Figure 5. Simplified ontology model for an aircraft.

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



System Requirements

Id	Title	Description	System	Actions
1	A plane needs wings	A wing is a type of fin with a surface that produces aerodynamic force for flight or propulsion through the atmosphere	1	Edit Delete
3	The plane needs throttle levers	Each thrust lever displays the engine number of the engine it controls	1	Edit Delete
4	The length of the plane	The length of the entire aircraft should be 254 meters	1	Edit Delete
5	The plane should have engines	An aircraft engine is the component of the propulsion system for an aircraft that generates mechanical power	1	Edit Delete
6	The capacity is 255 passengers	The aircraft needs to have a passengers capacity of 255	1	Edit Delete

Figure 6. Panel showing all the requirements for the system *UMDBus 787*.

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. The output from these actions is shown in Figures 7 and 8, respectively.

V. Conclusions and Future Work

When a system is prescribed by a large number of (non formal) textual requirements, the combination of previously defined ontology models and natural language processing techniques can play an important role in validating and verifying a system design. Future work will include formal analysis on the attributes of each property coupled with use of NLP to extract ontology information from a set of requirements. Rigorous automatic domain ontology extraction requires a deep understanding of input text, and so it is fair to say that these techniques are still relatively immature. A second opportunity is the use of NLP techniques in conjunction with a repository of acceptable "template sentence structures" for writing requirements [16]. Finally, there is a strong need for techniques that use the different levels of detail in the requirements specification, and bring ontology models from different domains to validate that the requirements belongs to the supposed domain. This challenge belongs to the NLP area of *classification*.

References

[1] M.A. Austin, and J.S. Baras, "An Introduction to Information-Centric Systems Engineering". Toulouse, France: Tutorial F06, INCOSE, June 2004.

[2] V. Ambriola and V. Gervasi, "Processing Natural Language Requirements," in Proceedings 12th IEEE International Conference Automated Software Engineering. IEEE Comput. Soc, 1997, pp. 36–45. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=632822

[3] C. Rolland and C. Proix, "A Natural Language Approach for Requirements Engineering," in Advanced Information Systems Engineering. Springer, 1992, pp. 257–277.

[4] K. Ryan, "The Role of Natural Language in Requirements Engineering," in [1993] Proceedings of the IEEE International Symposium on Requirements Engineering. IEEE Comput. Soc. Press, 1993, pp. 240–242. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=324852

[5] NLTK Project, "Natural Language Toolkit NLTK 3.0 documentation." [Online]. Available: http://www.nltk.org/

[6] C. Manning and H. Schuetze, Foundations of Statistical Natural Language Processing. The MIT Press, 2012. [Online]. Available: http://www.amazon.com/Foundations-Statistical-Natural-Language-Processing-ebook/dp/B007L7LUKO

[7] M. Porter, "An Algorithm for Suffix Stripping," Program: electronic library and information systems, vol. 14, no. 3, Dec. 1980, pp. 130–137. [Online]. Available: http://www.emeraldinsight.com/journals.htm?issn=0033-0337&volume=14&issue=3&articleid=1670983&show=html

[8] C. D. Paice, "Another Stemmer," ACM SIGIR Forum, vol. 24, no. 3, Nov. 1990, pp. 56–61. [Online]. Available: http://dl.acm.org/citation.cfm?id=101306.101310

[9] Princeton University, "About WordNet - WordNet - About WordNet." [Online]. Available: http://wordnet.princeton.edu/

[10] M. Haspelmath, "Word Classes and Parts of Speech," 2001. [Online]. Available: http://philpapers.org/rec/HASWCA

[11] S. Bird, E. Klein, and E. Loper, Natural Language Processing with Python. O'Reilly Media, Inc., 2009.

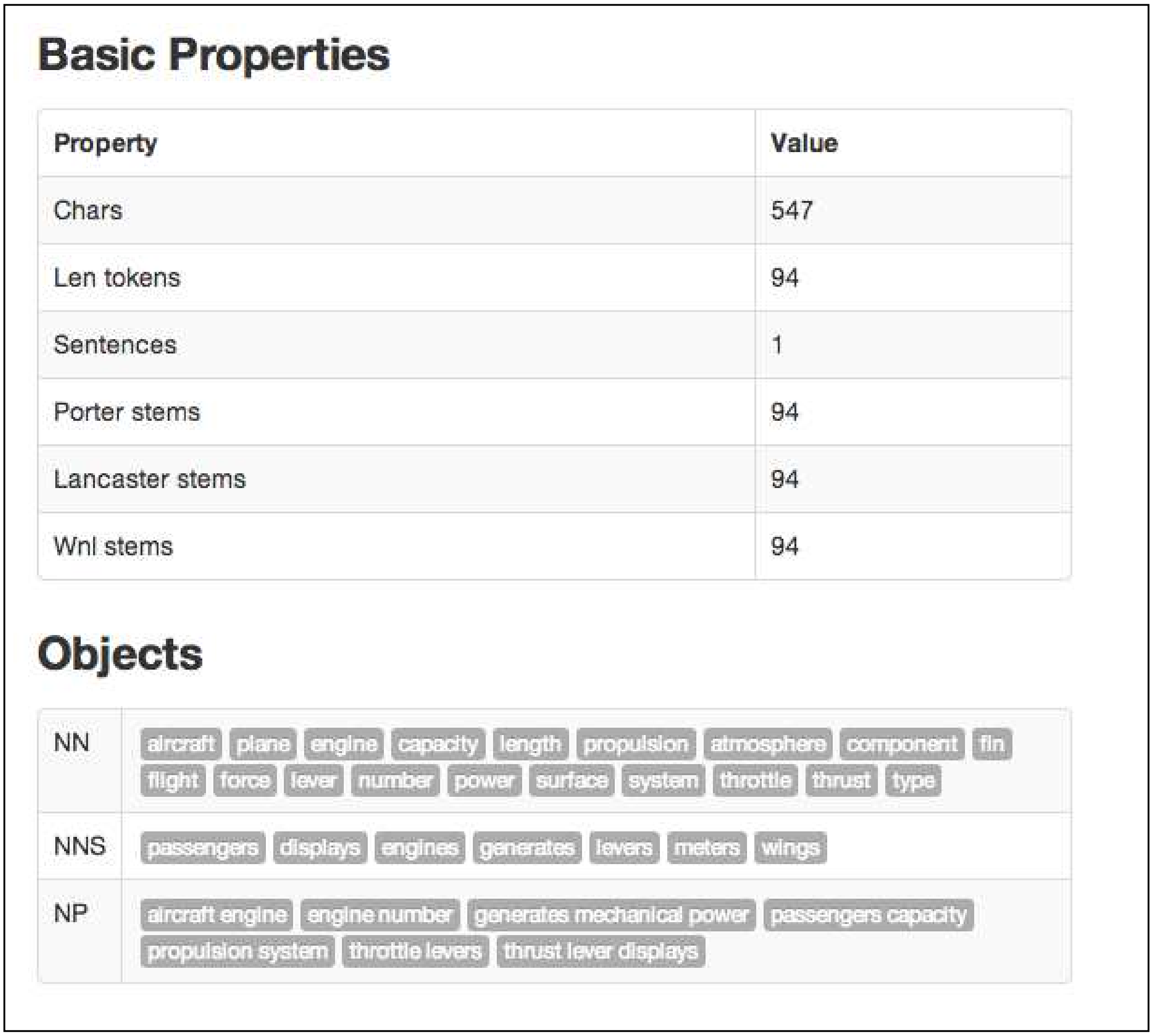
[12] University of Pennsylvania, "Penn Treebank Project." [Online]. Available: http://www.cis.upenn.edu/ treebank/

[13] B. Santorini, "Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision)," 1990. [Online]. Available: http://repository.upenn.edu/cis_reports/570

[14] Ruby on Rails. See http://rubyonrails.org/ (Accessed, March 2015).

[15] Bootstrap. See http://getbootstrap.com/2.3.2/ (Accessed, March 2015).

[16] E. Hull, K. Jackson, and J. Dick, Requirements Engineering. Springer, 2002. [Online]. Available: http://www.amazon.com/Requirements-Engineering-Elizabeth-Hull-ebook/dp/B000PY41OW



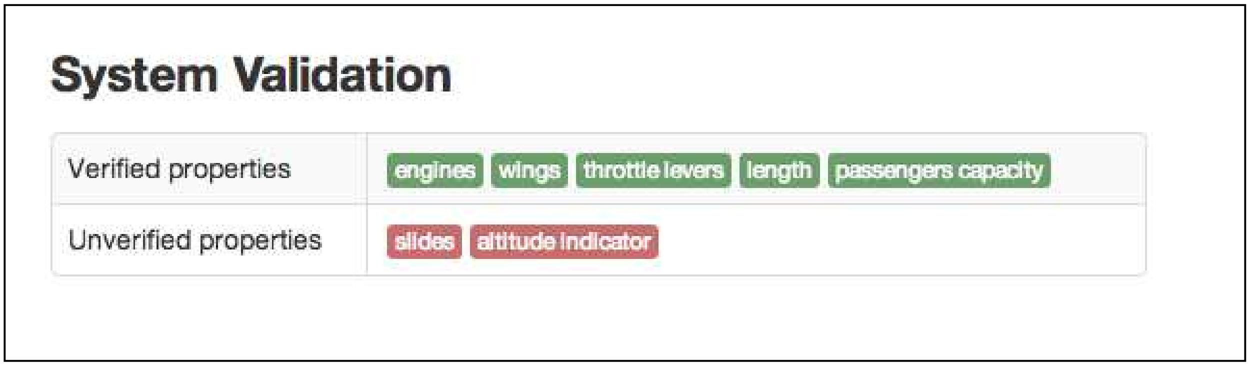
Basic Properties

Property	Value
Chars	547
Len tokens	94
Sentences	1
Porter stems	94
Lancaster stems	94
Wnl stems	94

Objects

NN	aircraft plane engine capacity length propulsion atmosphere component fin flight force lever number power surface system throttle thrust type
NNS	passengers displays engines generates levers meters wings
NP	aircraft engine engine number generates mechanical power passengers capacity propulsion system throttle levers thrust lever displays

Figure 7. Basic stats from the text, and a list of the entities recognized in it.



System Validation

Verified properties	engines wings throttle levers length passengers capacity
Unverified properties	slides altitude indicator

Figure 8. This is the final output from the application workflow. It shows what properties are verified (i.e., are present in the system requirements) and which ones are not.