Improving Identifier Informativeness Using Part of Speech Information

Dave Binkley  
Loyola University Maryland  
Baltimore MD  
21210-2699, USA  
binkley@cs.loyola.edu

Matthew Hearn  
Loyola University Maryland  
Baltimore MD  
21210-2699, USA  
mthearn@loyola.edu

Dawn Lawrie  
Loyola University Maryland  
Baltimore MD  
21210-2699, USA  
lawrie@cs.loyola.edu

ABSTRACT

Recent software development tools have exploited the mining of natural language information found within software and its supporting documentation. To make the most of this information, researchers have drawn upon the work of the natural language processing community for tools and techniques. One such tool provides part-of-speech information, which finds application in improving the searching of software repositories and extracting domain information found in identifiers.

Unfortunately, the natural language found in software differs from that found in standard prose. This difference potentially limits the effectiveness of off-the-shelf tools. An empirical investigation finds that with minimal guidance an existing tagger was correct 88% of the time when tagging the words found in source code identifiers. The investigation then uses the improved part-of-speech information to tag a large corpus of over 145,000 structure-field names. From patterns in the tags several rules emerge that seek to understand past usage and to improve future naming.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms

Source code analysis tool

Keywords

Natural language processing, program comprehension, identifier analysis

1. INTRODUCTION

Software engineering can benefit from leveraging tools and techniques of other disciplines. Traditionally, natural language processing (NLP) tools solve problems by processing the natural language found in documents such as news articles and web pages. One such NLP tool is a part-of-speech (POS) tagger. Tagging is, for example, crucial to the Named-Entity Recognition [3], which enables information about a person to be tracked within and across documents.

Many POS taggers are built using machine learning based on newswire training data. Conventional wisdom is that these taggers work well on newswire and similar artifacts; however, their effectiveness degrades as the input moves further away from the highly structured sentences found in traditional newswire articles.

The text available in source-code artifacts, in particular a program’s identifiers, has a very different structure. For example the words of an identifier rarely form a grammatically correct sentence. This raises an interesting question: can an existing POS tagger be made to work well on the natural language found in source code?

Better POS information would aid existing techniques that have used POS information to successfully improve retrieval results from software repositories [1, 13] and have also investigated the comprehensibility of source code identifiers [4, 7]. Fortunately, machine learning techniques are robust and, as reported in Section 2, good results are obtained using several sentence forming templates. This initial investigation also suggests rules specific to software that improve tagging. For example the type of a declared variable can be factored into its tags.

As an example application of POS tagging for source code, the tagger is used to tag over 145,000 structure-field names. Equivalence classes of the tags are then examined to produce rules for the automatic identification of poor names (as described in Section 3) and to suggest improved names, which is left to future work.

2. PART-OF-SPEECH TAGGING

Before a POS tagger’s output can be used as input to downstream source-code analysis tools, the POS tagger itself needs to be vetted. This section describes an experiment performed to test the accuracy of POS tagging on field names mined from source code. The process used for mining and tagging the fields is first described, followed by the empirical results from the experiment.

Figure 1 shows the pipeline used for the POS tagging of field names. On the left, the input to the pipeline is source code. This is then marked up with XML tags by srcML [5] to help identify various syntactic categories. Third, field names are extracted from the marked-up source using XPath queries. Figure 2 shows the queries for C++ and Java.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACKNOWLEDGMENTS

MSR ’11, May 21-22, 2011, Waikiki, Honolulu, HI, USA  
Copyright 2011 ACM 978-1-4503-0574-7/11/05 ...$10.00.
The fourth stage splits field names by replacing underscores with spaces and inserting a space where the case changes from lowercase to uppercase. For example, the names spongeBob and sponge_bob become sponge bob. More sophisticated splitting approaches have been presented [11, 6, 10]. These should increase the tags applicability. After splitting, all characters are shifted to lowercase. This stage also filters names so that only those that consist entirely of dictionary words are retained. Filtering uses Debian’s American (6-2) dictionary package, which consists of the 98,569 words from Kevin Atkinson’s SCOWL word lists that have size 10 through 50 [2]. This dictionary includes some common abbreviations, which are thus included in the final data set. Future work will employ vocabulary normalization in the hope of obviating the need for filtering by replacing nonwords with natural language equivalents [10].

The fifth stage applies a set of templates (described below) to each separated field name. Each template effectively wraps the words of the field name in an attempt to improve the performance of the POS tagger. Finally, POS tagging is performed by Version 1.6 of the Stanford Log-linear POS Tagger [14]. The default options are used including the pre-trained bidirectional model [12].

The remainder of this section considers empirical results concerning the effectiveness of the tagging pipeline. A total of 145,163 field names were mined from 10,985 C++ files and 9,614 Java files. These files came from 171 programs that represent a convenience sample whose source code could be easily downloaded from the Internet. This sample covers a wide range of application areas including accounting, aerospace, operating systems, program environments, movie editing, and games. From this full data set, 1500 names were randomly chosen (683 came from C++ files and 817 from Java files). A student majoring in English was then given the task of correcting the POS tags assigned to these names. The 1500 names and their corrected tags formed the oracle set, which is used to evaluate the accuracy of automatic tagging techniques.

Preliminary study of the Stanford tagger indicates that it needed guidance when tagging field names. Following the work of Abebe and Tonella [1], four templates were used to provide this guidance. Each template includes a slot into which the split field name is inserted. Their accuracy is then evaluated using the oracle set.

- **Sentence Template**: `<split field name>`.
- **List Item Template**: `- <split field name>`.
- **Verb Template**: `Please, <split field name>`.
- **Noun Template**: `<split field name>` is a thing.

The Sentence Template, the simplest of the four, considers the identifier itself to be a “sentence” by appending a period to the split field name. The List Item Template exploits the tagger having learned about POS information found in the sentence fragments used in lists. The Verb Template tries to encourage the tagger to treat the field name as a verb or a verb phrase by prefixing it with “Please,” since usually a command follows. Finally, the Noun Template tries to encourage the tagger to treat the field as a noun by postfixing it with “is a thing” as was done by Abebe and Tonella [1].

Table 1 shows the accuracy of the output from using each template and the 1500 names of the oracle set. The major diagonal represents each technique in isolation while the remaining entries require two techniques to agree and thus lowering the percentage. The similarity of the percentages in a column gives an indication of how similar the set of correctly tagged names is for two techniques. For example, considering Sentence Template, Verb Template has the lowest overlap of the remaining three as indicated by it’s joint percentage of 71.7%.

Of the four, the List Item Template performs the best, and the Sentence Template and Noun Template produce essentially identical results getting the correct tagging on nearly all the same fields. Perhaps unsurprising, the Verb Template performs the worst. Nonetheless, it is interesting that this template does produce the correct output on 3.2% of the fields where no other template succeeds.

<table>
<thead>
<tr>
<th></th>
<th>Sentence</th>
<th>List Item</th>
<th>Verb</th>
<th>Noun</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sentence</strong></td>
<td>79.1%</td>
<td>76.5%</td>
<td>71.7%</td>
<td>77.0%</td>
</tr>
<tr>
<td><strong>List Item</strong></td>
<td>76.5%</td>
<td>81.7%</td>
<td>71.0%</td>
<td>76.0%</td>
</tr>
<tr>
<td><strong>Verb</strong></td>
<td>71.7%</td>
<td>71.0%</td>
<td>76.0%</td>
<td>70.8%</td>
</tr>
<tr>
<td><strong>Noun</strong></td>
<td>77.0%</td>
<td>76.0%</td>
<td>70.8%</td>
<td>78.7%</td>
</tr>
</tbody>
</table>

Table 1: The percentage of correctly tagged field names using both the row and column technique; thus the major diagonal represent each technique independently.

Overall 68.9% of the identifiers were correctly tagged in all templates and 88.0% were correctly tagged in at least one. The latter percentage suggests that it may be possible to combine these results, perhaps using machine learning, to produce higher accuracy than achieved using the individual templates. Although 88% is lower than the 97% achieved by natural language taggers on the newswire data, the performance is still quite high considering the lack of context provided by the words of a single structure field. As illustrated in the next section, the identification is sufficiently accurate for use by downstream consumer applications.

3. RULES TO IMPROVE FIELD NAMES

As an example application of POS tagging for source code, the 145,163 field names of the full data set were tagged using the List Item Template, which showed the best performance in Table 1. The resulting tags were then used to form equivalence classes of field names. Inspection of the classes and the source led to four rules for improving the names of struc-
Rule 1 Non-boolean field names should not contain a present tense verb

\[<\text{Word}>^* \text{Past Tense Verb} <\text{Word}>^* \rightarrow <\text{Word}>^* \text{Past Tense Verb} <\text{Word}>^*\]

Violations detected: 27,743 (19.1% of field names)

When looking at the violations of Rule 1, one pattern that emerges suggests an improvement to the POS tagger that would better specialize it to source code. A pattern that frequently occurs in GUI programming finds verbs used as adjectives when describing GUI elements such as buttons. Recognizing such fields based on their type should improve tagger accuracy. Consider the fields delete_button and to a lesser extent continue_box. In isolation these appear to represent actions. However, they actually represent GUI elements. Thus a special, context-sensitive, case would tag such verbs as adjectives.

Rule 2 Field names should never be only a verb

\[<\text{Verb}> \rightarrow \begin{cases} <\text{Noun Phrase}> <\text{Past Tense Verb}> <\text{Verb}> <\text{Noun Phrase}> \end{cases}\]

Violations detected: 4,661 (3.2% field names identifiers)

The third rule considers field names that contain only an adjective. While adjectives are useful when used with a noun, an adjective alone relies too much on the type of the variable to fully explain its use. For example, consider the identifier interesting. In this case, the declared type of “int” provides the insight that this field holds a list of “interesting” items. Replacing this field with, for example, interesting_items removes the need to uncover the declared type and thus should improve code understanding.

Rule 3 Field names should never be only an adjective

\[<\text{Adjective}> \rightarrow <\text{Adjective}> <\text{Noun Phrase}>\]

Violations detected: 5,487 (3.8% field names identifiers)

An interesting exception to this rule occurs with data structures where the field name has an established conventional meaning. For example, when naming the next node in a linked list, next is commonly accepted. Other similar common names include “previous” and “last.”
needs to be obvious in the name. The identifier deleted offers a good example. Absent its type, does it represent a pointer to a deleted thing or perhaps a count of deleted things? Source code inspection reveals that such boolean variables tend to represent whether or not something is deleted. Thus potential improved names include is_deleted or was_deleted.

**Rule 4** Boolean field names should contain third person forms of the verb “to be” or the auxiliary verb “should”

```xml
<Word> is | was | should <Word>
```

Violations detected: 5,487 (3.8% field names identifiers)

Simply adding “is” or “was” to booleans does not guarantee a fix to the problem. For example, take a boolean variable that indicates whether something should be allocated in a program. In this case, the boolean captures whether some event should take place in the future. In this example an appropriate temporal sense is missing from the name. A name like allocated does not provide enough information and naming it is_allocated does not make logical sense in the context of the program. A solution to this naming problem is to change the identifier to should_be_allocated, which includes the necessary temporal sense communicating that this boolean is a flag for a future event.

4. RELATED WORK

This section briefly reviews three projects that use POS information. Each uses an off-the-shelf POS tagger or lookup table. First, Host et al. study naming of Java methods using a lookup table to assign POS tags [8]. Their aim is to find what they call “naming bugs” by checking to see if the method’s implementation is properly indicated by the name of the method. Second, Abebe and Tonella study class, method, and attribute names using a POS tagger based on a modification of minipar to formulate domain concepts [1]. Nouns in the identifiers are examined to form ontological relations between concepts. Based on a case study, their approach improved concept searching. Finally, Shepherd et al. considered finding concepts in code using natural language information [13]. The resulting Find-Concept tool locates action-oriented concerns more effectively than the other tools and with less user effort. This is made possible by POS information applied to source code.

5. SUMMARY

This paper presents the results on an experiment into the accuracy of the Stanford Log-linear POS Tagger applied to field names. The best template, List Item, has an accuracy of 81.7%. If an optimal combination of the four templates were used the accuracy rises to 88%. These POS tags were then used to develop field name formation rules that 29.9% of the identifiers violated. Thus the tagging can be used to support improved naming. Looking forward, two avenues of future work include additional empirical experimentation and tool improvements. The first includes, for example, considering the distribution of violations over programs and time. The second will consider mining the source for terms to be used in suggested name improvements and using more source-code specific POS tagging algorithm.

6. ACKNOWLEDGMENTS

Special thanks to Mike Collard for his help with srcML and the XPath queries and Phil Hearn for his help with creating the oracle set. Support for this work was provided by NSF grant CCF 0916081.

7. REFERENCES


