Improving Identifier Informativeness using Part of Speech Information

Dave Binkley  Matthew Hearn  Dawn Lawrie
Loyola University Maryland
Baltimore MD
21210-2699, USA
{binkley, lawrie}@cs.loyola.edu, mthearn@loyola.edu

Keywords: source code analysis tools, natural language processing, program comprehension, identifier analysis

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 is software differs from that found in standard prose. This difference potentially limits the effectiveness of off-the-shelf tools. The presented empirical investigation finds that this limitation can be partially overcome, resulting in a tagger that is up to 88% accurate when applied to source code identifiers. The investigation then uses the improved part-of-speech information to tag a large corpus of over 145,000 field names. From patterns in the tags several rules emerge that seek to improve structure-field naming.

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 the newswire and similar artifacts; however, their effectiveness degrades as the input moves further away from the highly structured sentences found in traditional newswire articles.

Figure 1. Process for POS tagging of field names.

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 limited POS information to successfully improve retrieval results from software repositories [1, 11] and have also investigated the comprehensibility of source code identifiers [4, 6]. 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 suggest rules specific for software that would 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 then used to tag over 145,000 structure-field names. Equivalence classes of tags are then examined to produce rules for the automatic identification of poor names (as described in Section 3) and 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 SE 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 using XML tags by srcML [5] to 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.

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. 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 obviate the need for filtering through vocabulary normalization in which non-words are split into their abbreviations and then expanded to their natural language equivalents [9].

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 [12]. The default options are used including the pre-trained bidirectional model [10].

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 found in 171 programs. From this full data set, 1500 names were randomly chosen as a test set (683 came from C++ files and 817 from Java files). A human accessor (and university student majoring in English) tagged the 1500 field names with POS information producing the oracle set. This oracle set is used to evaluate the accuracy of automatic tagging techniques when applied to the test set.

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. 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 using each template applied to the test set with the output compared to the oracle. 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 its joint percentage of 71.7%. Overall, 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.

As shown in Table 2 overall at least one template correctly tagged 88% of the test set. This 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.
Table 1. Each percentage is the percent of correctly tagged field names using both the row and column technique; thus the major diagonal represent each technique independently.

<table>
<thead>
<tr>
<th>Sentence</th>
<th>List Item</th>
<th>Verb</th>
<th>Noun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence</td>
<td>79.1%</td>
<td>76.5%</td>
<td>71.7%</td>
</tr>
<tr>
<td>List Item</td>
<td>76.5%</td>
<td>81.7%</td>
<td>71.0%</td>
</tr>
<tr>
<td>Verb</td>
<td>71.7%</td>
<td>71.0%</td>
<td>70.6%</td>
</tr>
<tr>
<td>Noun</td>
<td>77.0%</td>
<td>76.0%</td>
<td>70.8%</td>
</tr>
</tbody>
</table>

Table 2. Correctly tagged identifiers

As illustrated in the next section, the identification is sufficiently accurate for use by downstream consumer applications.

3 Rules for Improving 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. Analysis of these classes led to four rules for improving the names of structure fields. Rule violations can be automatically identified using POS tagging. Further, as illustrated in the examples, by mining the source code it is possible to suggest potential replacements.

The assumption behind each rule is that high quality field names will provide better conceptual information, which aids an engineer in the task of forming a mental understanding of the code. Correct part-of-speech information can help inform the naming of identifiers, a process that is essential in communicating intent to future programmers.

Each rule is first informally introduced and then formalized. After each rule, the percentage of fields that violate the rule is given. Finally, some rules are followed by a discussion of rule exceptions or related notions.

The first rule observes that field names represent objects not actions; thus they should avoid present-tense verbs. For example, the field name `create_mp4`, clearly implies an action, which is unlikely the intent (unless perhaps the field represent a function pointer). Inspection of the source code reveals that this field holds the desired mp4 video stream container type. Based on the context of its use, a better, less ambiguous name for this identifier is `created_mp4_container_type`, which includes the past-tense verb `created`. A notable exception to this is fields of type boolean, like, for example, `is_logged_in` where the present tense of the verb “to be” is used. A present tense verb in this context is used to represent a current state, and is therefore not confusing.

**Rule 1** Non-boolean field names should never contain a present tense verb

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

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

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 appears to represent actions. However they actually represent GUI elements. Thus, a special context-sensitive case in the POS tagger would tag such verbs as adjectives.

The second rule considers field names that contain only a verb. For example the field name `recycle`. This name communicates little to a programmer unfamiliar with the code. Examination of the source code reveals that this variable is an integer and, based on the comments, it counts the “number of things recycled.” While this meaning can be inferred from the declaration and the comments surrounding it, field name uses often occur far from their declaration, reducing the value of the declared type and supporting comments. A potential fix in this case is to change the name to `recycled_count` or `things_recycled`. Both alternatives improve the clarity of the name.

**Rule 2** Field names should never be only a verb

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

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 “list” provides the insight that this field holds a list of “interesting” items. Replacing this field with `interesting_list` or `interesting_items` 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 “current.”

The final rule deals with field names for booleans. Boolean variables represent a state that is or is not and this notion needs to be obvious in the name. The identifier deleted offers a good example. By itself there is no way to know for sure what is being represented. Is this a pointer to a deleted thing? Is it a count of deleted things? Source code inspection reveals that such boolean variables tend to represent whether or not something is deleted. Thus a 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”

\[\text{<Word>}^* \rightarrow \text{is | was | should} \quad \text{<Word>}^*\]

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 something expected to happen in the future.

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 28.9% of the identifiers violated. Thus the tagging can be used to support improved naming.

Looking forward, two avenues of future work include automating this improvement and enhancing POS tagging for source code. For the first, the source code would be mined for related terms to be used in suggested improved names. The second would explore training a POS tagger using, for example, the machine learning technique domain adaptation [8], which emphasize the text in the training that is most similar to identifiers to produce a POS tagger for identifiers.

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.

References