A Comparative Study on the Effectiveness of Part-of-Speech Tagging Techniques on Bug Reports

Yuan Tian and David Lo
School of Information Systems,
Singapore Management University, Singapore
{yuan.tian.2012,davidlo}@smu.edu.sg

***Abstract*—Many software artifacts are written in natural language or contain substantial amount of natural language contents. Thus these artifacts could be analyzed using text analysis techniques from the natural language processing (NLP) community, e.g., the part-of-speech (POS) tagging technique that assigns POS tags (e.g., verb, noun, etc.) to words in a sentence. In the literature, several studies have already applied POS tagging technique on software artifacts to recover important words in them, which are then used for automating various tasks, e.g., locating buggy files for a given bug report, etc.**

There are many POS tagging techniques proposed and they are trained and evaluated on non software engineering corpus (documents). Thus it is unknown whether they can correctly identify the POS of a word in a software artifact and which of them performs the best. To fill this gap, in this work, we investigate the effectiveness of seven POS taggers on bug reports. We randomly sample 100 bug reports from Eclipse and Mozilla project and create a text corpus that contains 21,713 words. We manually assign POS tags to these words and use them to evaluate the studied POS taggers. Our comparative study shows that the state-of-the-art POS taggers achieve an accuracy of 83.6%-90.5% on bug reports and the Stanford POS tagger and the TreeTagger achieve the highest accuracy on the sampled bug reports. Our findings show that researchers could use these POS taggers to analyze software artifacts, if an accuracy of 80-90% is acceptable for their specific needs, and we recommend using the Stanford POS tagger or the TreeTagger.

I. Introduction

In recent years, software engineering researchers have analyzed textual information in software artifacts, such as source code, bug reports, and requirements, to build automated solutions that help developers in program comprehension, software development and evolution tasks. For instance, by analyzing textual contents in bug reports, researchers have built various bug report management tools, e.g., bug report assignment tools [2], [3], [23], duplicate bug report detection tools [19], [20], bug report localization tools [14], [22], [13], etc.

To analyze textual contents in software artifacts, software engineering researchers have often made use of off-the-shelf text analysis tools borrowed from the natural language processing (NLP) community. One such tool is a part-of-speech (POS) tagger. A POS tagger assigns part-of-speech tags (e.g., noun, verb, etc.) to words in a sentence. POS tagging is a well studied problem in the NLP community and has been widely used in tasks such as information retrieval, word sense disambiguation, text parsing, etc.

In the software engineering community, POS taggers have shown their power in increasing the informativeness of source code identifiers [5], [9], [10], and in identifying important words from software artifacts such as bug reports [8], [18]. However, despite the increasing usage of POS taggers in software engineering domain, there exists no evaluation of the performance and applicability of such taggers for software engineering domain. Such study will shed light on whether POS taggers are accurate when applied on software artifacts, and which of the POS taggers perform the best for software artifacts. The evaluation of taggers' accuracy is important considering that off-the-shelf POS taggers are often trained on general English text (e.g., news article) that might contain little (if any) software engineering contents. Results of this study help future researchers to make an informed decision on whether to use POS taggers in their work and which of the POS taggers they should use.

To address the above mentioned need, in this work, we conduct a comparative analysis of seven state-of-the-art POS tagging techniques on software artifacts. We use popular off-the-shelf implementations of these taggers that have been trained on general English text. We choose bug reports as the target software artifact since a POS tagger has been applied to build an automated bug report management tool [18]. To compare the different POS taggers, we sample 100 bug reports from Eclipse and Mozilla projects and create a ground truth set by manually tagging the words in the bug reports. For each bug report, we extract and preprocess textual information from its summary, description and comment fields. This process results in a text corpus that contains a total of 21,713 words. We apply seven POS taggers on the sampled bug reports and evaluate their performance by comparing the tags assigned by these POS taggers and our ground truth set. We also evaluate the performance of the POS taggers for various word types.

Our comparative study finds that:

- The state-of-the-art POS taggers achieve an accuracy of 83.6%-90.5% when applied on bug reports. The Stanford POS tagger performs the best, followed by the TreeTagger. The accuracies of the studied POS taggers on bug reports are worse than their accuracies on regular English corpus – which could be as high as 97%.
- The TreeTagger performs the best in assigning tags to nouns and verbs, while the Stanford tagger performs the best in assigning tags to adjectives and adverbs.

978-1-4799-8469-5/15 © 2015 IEEE

SANER 2015, Montréal, Canada

II. State-of-the-art POS Taggers

In this section, we introduce the state-of-the-art POS taggers that could be applied on bug reports.

A. Overall View

In the NLP community, various techniques have been proposed to assign POS tags to words in a sentence. These approaches could be categorized into two types: rule-based and stochastic-based. The former one assigns tags based on rules while the later one assigns tags based on probability models. Both rule-based and stochastic-based approaches can be supervised or unsupervised. Supervised taggers typically learn models (or rules) from human-tagged textual corpus and apply them on unseen sentences. In contrast, unsupervised taggers do not require a pre-tagged corpus but instead use computational methods to automatically generate word groupings (i.e., tag sets). Those word groupings are then used to either calculate the probabilistic information needed by stochastic taggers or generate the rules needed by rule-based systems. Therefore, unsupervised taggers are usually integrated as preprocessing tools for system that build upon them [4], rather than be directly used to assign POS tags. In this work, we consider four types of widely used and well evaluated supervised POS taggers that have been trained on a tagged regular English corpus, given the fact that most of the bug reports analyzed in the previous software engineering studies are written in English.

B. POS Taggers

1) Unigram Tagger: A unigram POS tagger simply assigns to a word, the tag that is most likely for the word, based on a tagged corpus (i.e., training corpus). For instance, it assigns "verb" to the word "fix" if "fix" is more often tagged as a "verb" in the training corpus. To find the most likely tag for each word, a unigram POS tagger computes and stores the frequency of tags used for each word based on the tagged training corpus. For those words that do not appear in the training corpus, a unigram tagger assigns "noun" to them as the default tag. The unigram POS tagger is simple and fast, and achieves an acceptable accuracy if the training corpus is large enough. However, it ignores the surrounding (i.e., context) of a word when it assigns a POS tag to it.

2) Hidden Markov Model (HMM) Based Tagger: Similar to the unigram tagger, a hidden Markov model (HMM) based tagger assigns POS tags by searching for the most likely tag for each word in a sentence. However, a HMM based tagger finds a tag sequence for a sentence as a whole, rather than finding a tag for each word separately. Given a sentence $w_1 \dots w_n$, a HMM based tagger chooses a tag sequence $t_1 \dots t_n$ that maximizes the following joint probability:

$$P(t_1 \dots t_n, w_1 \dots w_n) = P(t_1 \dots t_n)P(w_1 \dots w_n | t_1 \dots t_n)$$

In practice, it is often impractical to compute $P(t_1 \dots t_n)$. Therefore many different taggers have been proposed to simplify this probability computation. ***TnT***, one of the most commonly used HMM based tagger, uses second order Markov models to simplify the computation; it assumes that the tag of a word is determined by the POS tags of the previous two words [6]. *Tree tagger* is another popular HMM based tagger, which leverages decision trees to get more reliable estimates of parameters in Markov models [17].

3) Maximum Entropy Based Tagger: The unigram and HMM based taggers are easy to build, however given the nature of their probability models, it is hard to incorporate more complex features into them. The maximum entropy (ME) based tagger is introduced to provide a principled way of incorporating complex features into probability models [16]. Given a sentence $w_1 \dots w_n$, an ME based tagger models the conditional probability of a tag sequence $t_1, \dots t_n$ as:

$$P(t_1, \dots t_n | w_1 \dots w_n) \approx \prod_{i=1}^{N} P(t_i | C_i)$$

where $C_1, \dots C_n$ are the corresponding contexts for each word appearing in the sentence. The context C of a word w includes the previous assigned tags before w.

An ME based tagger introduces the concept of *features* which encode elements of a context C useful for predicting the tag t of a word w. Features are binary valued functions that represent constraints. An ME based tagger will use the features to compute $P(t_i|C_i)$. It will learn the weights of the features that can maximize the entropy of the probability model using the training corpus. Different ME based POS taggers that make use of different features have been proposed. The NLTK natural language toolkit contains a ME based POS tagger that is implemented by Loper and Chichkov[1] (referred to as ***NLTK tagger*** in this paper). ***Stanford POS tagger*** is another popular ME based tagger that improves the original maximum entropy based tagger [16] by considering two more types of features related to the writing style of letters (e.g., whether the first letter is capitalized or not) [21].

4) Transformation based Tagger: A transformation based POS tagger assigns POS tags to words based on linguistic knowledge, expressed as rules, that are automatically learned from a training corpus. In particular, the transformation based tagger first uses a simple stochastic based tagger to get an initial tag for a word and then go back to fix the error if the word is wrongly tagged. In this way, the rules that could turn a badly tagged text into a good one are automatically learned. ***TBT tagger*** is the POS tagger proposed by Brill, who first introduced the idea of transformation based tagger [7]. The TBT tagger uses a unigram tagger to get the initial tags. *Annie tagger* is another popular transformation based tagger [12].

III. Methodology

In this section, we describe the methodology to collect and preprocess bug reports, and build a ground truth set. We also present the POS taggers that we compare in this study and their implementations.

A. Bug Report Collection

Bug tracking systems, such as Bugzilla, Jira, etc., are often used by developers to collect and maintain bug reports. Each bug report contains information that could help developers to reproduce and fix the bug. A bug report also contains related discussion on the reported bug. Table I describes the fields in a bug report that are of interest to us.

[1] http://www.nltk.org/modules/nltk/classify/maxent.html

TABLE I: Textual fields in a bug report

Field	Description
Summary	A short description of the bug that is written by the bug reporter.
Description	A long description of the bug that is written by the bug reporter. It typically includes information such as error message, steps to reproduce the bug, stack traces, etc.
Comments	Discussions about the bug that are contributed by developers. Typically these discussions include information such as the root cause of the bug, review of a patch, etc.

In this work, we consider bug reports from Eclipse and Mozilla project that are both stored in a Bugzilla[2] bug tracking system. Eclipse is a popular multi-language integrated development environment (IDE). Mozilla is a large non-profit organization that hosts and develops products such as Firefox, Thunderbird, etc. We crawl bug reports from the Bugzilla websites of these projects. Given the fact that it's hard to manually label all words in all existing bug reports of Eclipse and Mozilla, we randomly select 50 bug reports from each project and generate a sample set that contains 100 bug reports. Table II shows the basic statistics of our dataset.

TABLE II: Dataset

Project	Period	#Bug Report	#Sample Bug Report	#Sentences	#Words
Eclipse	Jan-Dec 2012	29,589	50	832	8,313
Mozilla	Jan-Dec 2012	95,696	50	1,291	13,400
Total	Jan-Dec 2012	125,285	100	2,123	21,713

B. Bug Report Preprocessing

Some pieces of information extracted from the summary, description and comments fields of a bug report might be hard to be preprocessed directly using the existing English POS taggers. These include non natural language contents such as code segments and stack traces. These two kinds of information require other techniques, e.g., the code POS tagger proposed by Binkley et al. [5], to help in the assignment of POS tags. Therefore, in this work, we manually remove code segments and stack traces from the sampled 100 bug reports. Note that method names, class names, and variable types appearing in text segments are still included. We also remove quoted sentences inside comments (since they are duplicate text) and fix format errors inside bug reports (e.g., relink broken sentences).

C. Ground Truth Construction

We ask six participants, which include five PhD students and one postdoc with more than five years of experience in programming, to assign tags to words in the sampled bug reports. We ask participants to assign to each word inside a bug report one of the tags in a predefined tag set. Since nouns contain the most important information in a bug report [8], [18], and verbs, adjectives and adverbs could also potentially be helpful [5], we have the following tag set:

- Noun
- Verb
- Adverb & adjective
- Others (e.g., determiner, conjunction, etc.)

We merge adverbs and adjectives into the same group because they perform a similar function (i.e., describe a noun/verb). Name of a method, class or type of a variable appearing in text segments is regarded as noun.

To increase the confidence in the assigned POS tags, we make sure that each bug report is assigned by two different people. We then check whether the two people give the same POS tag for each word in the bug report and ask them to discuss and make a decision if their assigned tags are different.

TABLE III: Selected POS taggers

Tagger	Type	Tool
Unigram Tagger	Unigram	NLTK toolkit [3]
TnT Tagger	Hidden Markov Model	NLTK toolkit
TreeTagger	Hidden Markov Model	TreeTagger [4]
NLTK Tagger	Maximum Entropy	NLTK toolkit
Stanford Tagger	Maximum Entropy	Stanford Tagger [5]
TBT Tagger	Transformation	NLTK toolkit
Annie Tagger	Transformation	Gate toolkit [6]

D. POS Tagger Selection and Implementation

We select seven commonly used POS taggers to represent the four types of POS taggers that are introduced in Section II. The seven selected POS taggers include the Annie tagger, the TreeTagger, and the Stanford tagger, which have been applied on software artifacts in previous studies [5], [8], [9], [10], [18]. Table III shows the seven POS taggers and the tools that implement them. These POS taggers are all supervised taggers and thus they require a pre-tagged training corpus. We use a commonly used tagged regular English corpus, i.e., the Wall Street Journal (WSJ) corpus [15] as the training corpus.

IV. Preliminary Experiment

In this section, we first introduce two research questions. We then present and analyze the results of our preliminary experiments. Finally, we describe several threats to validity.

A. Research Questions

We compare the state-of-the-art POS taggers by answering the following two research questions:

RQ1 How effective are the POS tagging techniques on bug reports, compared with their performance on a general English corpus?

RQ2 Which POS tagging technique performs the best for various word types, i.e., all non stop-words[7], nouns, verbs, adjectives and adverbs?

[2]http://www.bugzilla.org/

[3]http://www.nltk.org/

[4]http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/

[5]http://nlp.stanford.edu/software/tagger.shtml

[6]https://gate.ac.uk/

[7]Stop words are frequently occurring words that have little semantic value, e.g., "is", "are", etc.

B. Performance of POS Taggers on Bug Reports (RQ1)

We use accuracy (i.e., the ratio of the number of correctly tagged words to the total number of tagged words) to measure the performance of a POS tagger on a text corpus. Table IV shows the accuracy of each POS tagger on our created ground truth set and their accuracy on a regular English corpus, i.e., the WSJ corpus. Note that the accuracy on the regular English corpus is computed by training a POS tagger on a part of the WSJ corpus and applying it on a different part of the WSJ corpus. From Table IV, we observe that the studied POS taggers could achieve an accuracy of 83.6% to 90.5% on the sampled bug reports. The Stanford POS tagger performs the best followed by the TreeTagger, while the unigram POS tagger performs the worst. The taggers performance on the bug reports are all worse than their performance on the regular English corpus. The last column in Table IV shows the ratio of the accuracy of a POS tagger on the bug reports and its accuracy on the regular English corpus. Although the unigram tagger performs the worst, the difference between its performance on bug reports and that on the regular English corpus is small. For the other POS taggers, the accuracy of the TnT HMM-Based and NLTK ME-Based taggers decrease the most when they are applied on bug reports.

TABLE IV: Performance of the POS taggers on bug reports and a regular English news corpus. Values in column "Ratio" are the ratios of the values in column "Bug Reports" and the values in column "Regular English Corpus".

Tagger	Bug Reports	Regular English Corpus	Ratio
Unigram	83.6%	86%	97.2%
TnT	84.9%	96%	88.4%
Tree	90.4%	97%	93.2%
NLTK	85.1%	97%	87.7%
Stanford	**90.5%**	97%	93.3%
TBT	87.2%	97%	89.9%
Annie	88.6%	97%	91.3%

C. Comparative Analysis of the Seven POS Taggers for Various Word Types (RQ2)

In this research question, we investigate the performance of the selected POS taggers under four different settings: 1) on tagging all non stop words; 2) on tagging nouns; 3) on tagging verbs; and 4) on tagging adjectives and adverbs. Performance on the first setting is evaluated using accuracy, while the performance on the other three settings are evaluated using F-measure. We do not use F-measure to evaluate results on non stop words, because non stop words contains words with various tag type while we can only compute F-measure for each tag type. We compute the F-measure as the harmonic mean of precision and recall, where precision is the number of correctly tagged nouns divided by the number of all words tagged as noun, and recall is the number of correctly tagged nouns divided by the number of nouns. Table V shows the performance of the seven POS taggers. The table shows that the TreeTagger performs the best for the first three settings, i.e., on tagging non stop-words, nouns, and verbs, while the Stanford tagger performs the best on the last setting, i.e., on tagging adjectives and adverbs. Overall, the performance of the TreeTagger and the Stanford tagger is close to each other and these two taggers perform better than the other POS taggers for all settings.

TABLE V: Performance of POS taggers on bug reports

Tagger	Non Stop-words (Accuracy)	Noun (F-measure)	Verb (F-measure)	Adj. & Adv. (F-measure)
Unigram	77.8%	83.4%	69.0%	73.5%
TnT	79.6%	84.7%	73.2%	74.8%
Tree	**89.2%**	**92.7%**	**90.0%**	81.0%
NLTK	80.0%	84.8%	78.7%	71.7%
Stanford	88.9%	92.3%	88.9%	**82.7%**
TBT	84.2%	88.3%	82.4%	76.7%
Annie	85.5%	89.4%	84.2%	80.6%

D. Threats to Validity

Threats to internal validity relates to experimental errors and biases. Our tag set contains four types of POS tag, which software engineers might be interested in. However, it ignores many of the errors in tagging, e.g., a word being tagged as an adverb instead of an adjective. We would like to consider a more fine-grained tag set in the future. To build the ground truth dataset, we ask six participants, who have more than five years of programming, to assign POS tags to words that appear in the sampled bug reports. These participants are fluent English speakers and writers, but might still wrongly label some of the words because they are not experts in English. To reduce bias introduced by the human labeling process, we assign each bug report to two different participants. These two participants will discuss and make a final decision on the POS tag of a word when they give different tags to the word.

Threats to external validity relates to the generalizability of our approach. We use 100 randomly sampled bug reports from Mozilla and Eclipse to evaluate POS tagging techniques. Although these bug reports contain 21,713 words, results on these bug reports might not represent the results on all bug reports or other types of software artifacts (e.g., source code comments). In the future, we plan to expand this study by including more bug reports from additional open source projects and other software artifacts.

Threats to construct validity relates to the suitability of our evaluation metrics. Similar with a previous work [5], we use the percentage of correctly tagged words (i.e., accuracy) to measure the performance of a POS tagger. To measure the performance of a POS tagger for different types of words (i.e., nouns, verbs, adjectives and adverbs), we compute F-measure, which is a well-known evaluation metric [11].

V. Related Work

A. POS Tagging for Identifying Important Words

Capobianco et al. use a POS tagging technique (i.e., TreeTagger) to improve the performance of information retrieval (IR) based traceability recovery approaches [8].[8] In particular, they apply the TreeTagger to identify noun terms and non-noun terms (terms with other part-of-speech tags, e.g., verbs, adjectives, etc.) from high-level textual artifacts (e.g., user cases), and show that by only retaining the noun terms in the textual artifacts, the accuracy of various IR-based traceability recovery approaches significantly improves.

[8]Traceability recovery is the task of reconstructing the links between software artifacts. Usually the links are between source code and other artifacts, e.g., use cases, test cases, requirements, etc.

Similarly, Shokripour et al. integrate a POS tagger (i.e., Annie tagger) into their proposed automated bug report assignment approach to identify noun terms in software artifacts (i.e., bug reports and source code identifiers) [18]. The above two studies directly apply POS tagging techniques on software artifacts without checking their accuracy in assigning part-of-speech tags to terms in software artifacts, which might impact the final performance of their approaches. Additionally, it is unknown whether the POS taggers used in their studies are the best ones for software artifacts, among existing state-of-the-art POS taggers. To complement these existing studies, we perform this comparative study.

B. POS Tagging for Program Comprehension

Abebe and Tonella apply natural language parsing to sentences generated from source code identifiers for better program comprehension [1]. Following the work of Abebe and Tonella, Binkley et al. investigate the performance of a POS tagging technique (i.e., Stanford tagger) on source code identifiers [5]. They first form sentences from identifiers following Abebe and Tonella's approach and then pass the sentences to the Stanford POS tagger. They find that the POS tagger applied on the sentences has an accuracy of 81.7%. Falleri et al. propose an automated approach to extract identifiers from source code and organize them in a network to give insight on the software architecture [9]. Their generated network captures the relations (i.e., synonymy, hyponymy and hyperonymy) between identifiers with their constitutent terms tagged by a POS tagger (i.e., TreeTagger). Gupta et al. propose a new POS tagger for program identifiers [10]. They use WordNet[9] to assign POS tags to terms splitted from identifiers, and then use a set of rules based on common naming conventions to correct the initial POS tags. The above studies focus on applying or customizing POS taggers for source code identifiers rather than bug reports that are considered in our study.

VI. Conclusion and Future Work

Previous textual analysis work on bug reports and code identifiers show the potential power of part-of-speech tagging in supporting software engineering tasks, such as program comprehension and bug assignment. In this work, we compare the effectiveness of seven state-of-the-art POS taggers on bug reports. We build a ground truth set that contains 21,713 tagged words from 100 sampled bug reports from Eclipse and Mozilla project. Our preliminary experiment results show that the state-of-the-art POS taggers could achieve a reasonable accuracy on bug reports (83.6%-90.5%), although worse than its accuracy on a regular English corpus (97%, for most taggers). We also investigate the performance of POS taggers for different types of words. Generally, the Stanford and TreeTagger perform the best.

In the future, it would be interesting to increase the number of tagged bug reports and train POS taggers using tagged bug reports, rather than using regular English corpus. It would also be interesting to perform a qualitative study on the cases in which different types of taggers produce wrong tags. This might help us create an improved POS tagging technique that performs better than off-the-shelf POS taggers when apply on bug reports.

[9] http://wordnet.princeton.edu/

References

[1] S. L. Abebe and P. Tonella. Natural language parsing of program element names for concept extraction. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE, 2010.

[2] M. Alenezi, K. Magel, and S. Banitaan. Efficient bug triaging using text mining. *Journal of Software*, 8(9):2185–2190, 2013.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE*. ACM, 2006.

[4] C. Biemann, C. Giuliano, and A. Gliozzo. Unsupervised part-of-speech tagging supporting supervised methods. In *RANLP*, 2007.

[5] D. Binkley, M. Hearn, and D. Lawrie. Improving identifier informativeness using part of speech information. In *MSR*, 2011.

[6] T. Brants. Tnt: a statistical part-of-speech tagger. In *Proceedings of the sixth conference on Applied natural language processing*, pages 224–231. Association for Computational Linguistics, 2000.

[7] E. Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, pages 543–565, 1995.

[8] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella. Improving IR-based traceability recovery via noun-based indexing of software artifacts. *Journal of Software: Evolution and Process*, 25(7):743–762, 2013.

[9] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic extraction of a wordnet-like identifier network from software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE, 2010.

[10] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *ICPC*, 2013.

[11] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 2000.

[12] M. Hepple. Independence and commitment: Assumptions for rapid training and execution of rule-based POS taggers. In *ACL*. Association for Computational Linguistics, 2000.

[13] P. S. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: Do they matter? In *ASE*, pages 803–814. ACM, 2014.

[14] T.-D. B. Le, S. Wang, and D. Lo. Multi-abstraction concern localization. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 364–367. IEEE, 2013.

[15] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

[16] A. Ratnaparkhi et al. A maximum entropy model for part-of-speech tagging. In *Proceedings of the conference on empirical methods in natural language processing*, volume 1, pages 133–142. Philadelphia, PA, 1996.

[17] H. Schmid. Improvements in part-of-speech tagging with an application to german. In *ACL SIGDAT Workshop*, 1995.

[18] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *MSR*, 2013.

[19] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE*. IEEE, 2011.

[20] Y. Tian, C. Sun, and D. Lo. Improved duplicate bug report identification. In *CSMR*, pages 385–390. IEEE, 2012.

[21] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for Computational Linguistics, 2003.

[22] S. Wang and D. Lo. Version history, similar report, and structure: putting them together for improved bug localization. In *ICPC*, pages 53–63. ACM, 2014.

[23] X. Xia, D. Lo, X. Wang, and B. Zhou. Accurate developer recommendation for bug resolution. In *WCRE*, pages 72–81. IEEE, 2013.