An Ensemble Approach for Annotating Source Code Identifiers with Part-of-speech Tags

Christian D. Newman*, Michael J. Decker†, Reem S. Alsuhaibani‡, Anthony Peruma*, Mohamed Wiem Mkaouer*, Satyajit Mohapatra*, Tejal Vishnoi*, Marcos Zampieri§, Timothy J. Sheldon¶, Emily Hill‖

*Software Engineering Deptartment, Rochester Institute of Techonology, Rochester, NY
†Software Engineering Department, Bowling Green State University, Bowling Green, OH
‡Computer Science Department, Prince Sultan University, Riyadh, Saudi Arabia
§Language Technology Group, Rochester Institute of Techonology, Rochester, NY
¶Financial Services Sector, Risk and Compliance, BNY Mellon, Pittburgh, PA
‖Department of Math and Computer Science, Drew University, Madison, NJ

***Abstract*—This paper presents an ensemble part-of-speech tagging approach for source code identifiers. Ensemble tagging is a technique that uses machine-learning and the output from multiple part-of-speech taggers to annotate natural language text at a higher quality than the part-of-speech taggers are able to obtain independently. Our ensemble uses three state-of-the-art part-of-speech taggers: SWUM, POSSE, and Stanford. We study the quality of the ensemble's annotations on five different types of identifier names: function, class, attribute, parameter, and declaration statement, at the level of both individual words and full identifier names. We also study and discuss the weaknesses of our tagger to promote the future amelioration of these problems through further research. Our results show that the ensemble achieves 75% accuracy at the identifier level and 84-86% accuracy at the word level. This is an increase of +17% points at the identifier level from the closest independent part-of-speech tagger.**

***Index Terms*—program comprehension, software maintenance, natural language processing, part-of-speech tagging**

I. Introduction

Program comprehension is a significant factor in the time it takes to develop and maintain software [1], [2]. Developers spend much more time reading code than they spend writing; 10 times more by some estimates [2]. Increased understanding of developer comprehension will lead to approaches that not only increase the ability of developers and program analysis tools to be productive, but also increase the accessibility of software development (e.g., by supporting programmers that prefer top-down or bottom-up comprehension styles [3], [4]) and help developers avoid stress stemming from code that is hard to understand. One of the primary ways a developer comprehends code is by reading identifier names which make up, on average, about 70% of the characters found in a body of code [5]. Therefore, improving identifier naming practices can have a significant impact on comprehension.

One challenge to studying identifiers is the difficulty in understanding how to map the meaning of natural language phrases to the behavior of the code. For example, when a developer names a method, the name should reflect the behavior of the method such that another developer can understand what the method does without the need to read the method body. Understanding this connection between name and behavior presents challenges for humans and tools; both of which use this relationship to comprehend, generate, or critique code. A second challenge lies in the natural language analysis techniques themselves, many of which are not trained to be applied to software [6], which introduces significant threats [7]. Addressing these problems is vital to improving the developer experience and augmenting tools which leverage natural language.

Analysis of identifier names can be done in many ways, including word frequency analysis [8] (e.g., ngrams) or semantic analysis using lexical ontologies like wordnet [9]. These are applied to a large number of problems, including rename refactoring analysis [10], [11], [12], [13], linguistic anti-patterns [14], identifier splitting [15], and part-of-speech tagging [16], [17], [18]. In this paper, we focus on part-of-speech tagging (POS); a technique whereby words in a sentence, or in an identifier in this case, are annotated based on the role they play within the context of the words surrounding them or based on their typical usage in the case where we are dealing with a single-word identifier. Part-of-speech tagging is one of the most popular methods for measuring the natural language semantics of identifier names and has been used in numerous other research [19], [17], [20], [21], [22], [23], [14], [13], [24]. Unfortunately, part-of-speech taggers for identifiers are still inaccurate [25], [18], making it difficult to trust their output.

The goal of this paper is to discuss and present an ensemble tagging technique that improves the accuracy of part-of-speech taggers, and supports a larger variety of POS tags than other software engineering based POS taggers, specifically SWUM and POSSE [17], [16]. The ensemble approach will use machine-learning algorithms such as Decision Tree [26] and Random Forest [27], which are common in other software research tasks [28], [29] and have been used for part-of-speech tagging of standard English documents [30]. The main contributions of this work are as follows:

1) An implementation of the most accurate (to-date) part-of-speech tagger for source code identifiers, built using data that was curated via significant manual-annotation effort made by the authors in prior work [25]. This approach is trained to support more types of annotations (i.e., POS tags) specifically oriented for source code than any other approach currently available. In addition, the ensemble has been made fully available (see Section IV), and is intended for long term support by the research team as we expand the training set and include identifiers from different contexts (e.g., test code).
2) Confirmation of observations we made in prior work [25] that indicate 1) the importance of the position of a word in an identifier, 2) the importance of the context of an identifier when annotating using part-of-speech, and 3) the complementarity of three part-of-speech taggers.
3) An expanded set of manually-annotated identifiers, based on the original set constructed in [25], that can be used to train and create more accurate tagging approaches and for other natural language problems. As with the implementation, this has been made fully available to the research community (see Section IV).
4) A thorough evaluation of the ensemble approach at both the identifier- and word-level, including a discussion of the features, which were empirically derived by the authors in prior work [25], that most positively influence the tagger's performance.
5) A discussion that provides a clear path for future work on part-of-speech tagger accuracy and effectiveness. This discussion is based on the authors' experience in prior work [25], [31] combined with phenomena observed during the evaluation of the tagger.

In addition to advancing the state-of-the-art of part-of-speech tagging, we also discuss where our approach is still weak, highlighting situations to which it may not sufficiently generalize due to potential limitations in our technique and our dataset. We answer the following Research Questions (RQs):

RQ1: How accurate is the ensemble part-of-speech tagger at the individual word level and how does this compare to the independent taggers? In prior research [25], we found that the output of three part-of-speech taggers complemented one another by applying them all to a manually-curated dataset of grammar patterns. This question will address just how much we can improve the accuracy of part-of-speech taggers on source code by combining the output of these taggers using machine learning.

RQ2: How accurate is the ensemble part-of-speech tagger at the identifier level and how does this compare to the independent taggers? In RQ1, we explore word-level accuracy. In RQ2, we will look at how accurate our ensemble is when we must get the entire identifier correct, since, as shown in prior work [25], even if a tagger has high accuracy on individual words, it may have low accuracy on full identifiers.

TABLE I: Examples of grammar patterns

Identifier Example	Grammar Pattern
1. GList* **tile list head** = NULL;	adjective adjective noun
2. GList* **tile list tail** = NULL;	adjective adjective noun
3. Gulong **max tile size** = 0;	adjective adjective noun
4. GimpWireMessage **msg**;	noun
5. **g list remove link** (tile list head, list)	preamble noun verb noun
6. **g list last** (list)	preamble adjective noun
7. **g assert** (tile_list_head != tile_list_tail);	preamble verb

RQ3: What are the most frequently mis-used part-of-speech tags and grammar patterns? This question investigates patterns in the way our approach mis-annotates identifiers. We explore these cases and discuss what further information the ensemble requires in order to handle these cases properly. For example, in prior work [25], we found that implementation details have an effect on the correct tag sequence for related identifiers and are thus required to properly tag these identifiers. In this question, we take a deeper look at this problem among others.

This paper is organized as follows: Section II provides the necessary definitions and background to understand the paper. Our methodology is detailed in Section IV. Section VI presents the evaluation of our ensemble and answers to our RQs. Section VII elaborates on our threats to validity. Section III explains related work and Section VIII summarizes our results, discusses future work, and concludes.

II. Definitions & Grammar Pattern Generation

The application of a part-of-speech tagger to an identifier results in a grammar pattern [25]. A *grammar pattern* is the sequence of part-of-speech tags (also referred to as annotations) assigned to individual words within an identifier. For example, for an identifier called GetUserToken, we assign a grammar pattern by splitting the identifier into its three constituent words: Get, User, and Token. We then run the split-sequence (i.e., Get User Token) through a part-of-speech tagger to get the grammar pattern: Verb Noun-adjunct Noun, which can help us understand how the individual words in this identifier are related. The advantage to using grammar patterns for identifier analysis is that a given pattern is not unique to any individual identifier, but is shared with many potential identifiers that use similar words. Thus, we can relate identifiers that contain different words to one another using their grammar pattern; GetUserToken, RunUserQuery, and WriteAccessToken share the same grammar pattern and, while they do not express the exact same semantics, there are similarities in their semantics which their grammar patterns reveal. Specifically, a verb (get, run, write) applied to a noun (token, query) with a specific role/context (user, access).

A. Annotating identifiers

Since part of the goal of this paper is to study an ensemble part-of-speech tagger, we use multiple taggers; POSSE [17],

TABLE II: Part-of-speech categories in dataset and supported by ensemble

Abbreviation	Expanded Form	Examples
N	noun	Disneyland, shoe, faucet, mother
DT	determiner	the, this, that, these, those, which
CJ	conjunction	and, for, nor, but, or, yet, so
P	preposition	behind, in front of, at, under, above
NPL	noun plural	Streets, cities, cars, people, lists
NM	noun modifier (adjectives, noun-adjuncts)	red, cold, hot, scary, beautiful, small
V	verb	Run, jump, spin
VM	verb modifier (adverb)	Very, loudly, seriously, impatiently
PR	pronoun	she, he, her, him, it,we,they,them
D	digit	1, 2, 10, 4.12, 0xAF
PRE	preamble	Gimp, GLEW, GL, G, p, m, b

SWUM [16], and Stanford [32][1]. POSSE and SWUM are part-of-speech taggers created specifically to be run on software identifiers; they are trained to deal with the specialized context in which identifiers appear. Both POSSE and SWUM take advantage of static analysis to provide annotations. For example, they will look at the return type of a function to determine whether the word *set* is a noun or a verb. Additionally, they are both aware of common naming structures in identifier names. For example, methods are more likely to contain a verb in certain positions within their name (e.g., at the beginning) [17], [16]. They leverage this information to help determine what part-of-speech to assign different words. Stanford is a popular part-of-speech tagger for general natural language (e.g., English) text. For our study, Stanford provides a baseline; it is not specialized for source code but was found to be reasonably accurate on method names [18] and shown to be competitive with POSSE [25].

This paper uses the part-of-speech tags given in Table II. Note, in this table, the *preamble* category, which does not exist in general natural language part-of-speech tagging approaches but has been defined and discussed in prior work [16], [25]. We use the definition from [25]: A Preamble is an abbreviation that occurs at the beginning of an identifier and does one of the following:

1) Namespaces an identifier without augmenting the reader's understanding of its behavior (e.g., XML in XML_Reader is not a preamble)
2) Provides language-specific metadata about an identifier (e.g., identifies pointers or member variables)
3) Highlights an identifier's type. When a preamble is highlighting an identifier's type, the type's inclusion must not add any new information to the identifier name.

For example, given an identifier *float* fPtr*, 'f' does not add any information about the identifier's role within the system, but reminds the developer that it has a type 'float'. However, given an identifier *char* sPtr*, 's' informs the developer that this is a c-string as opposed to a pointer to some other type of character array; 's' would not be considered a preamble under this definition. Additionally, some developers will put *p_* in front of pointer variables or m_ in front of variables that are members of a class; these are due to naming conventions and/or Hungarian notation [33], [22], [34], [35]. In the GIMP and GLEW open-source projects, GIMP and G_ are preambles to many variables, as seen in the Gimp example in Table I. Intuitively, the reason for identifying preambles in an identifier is because they do not provide any information with respect to the identifier's role within the system's domain. Instead, they provide one of the three types of information above. For this reason, when analyzing identifiers, tools should be able to determine when a word is a preamble so that they do not make false assumptions about the word's descriptive purpose.

Another tag to note from Table II is *noun modifier (NM)*, which is annotated on words that could be considered either pure adjectives or noun-adjuncts. A noun-adjunct is a word that is typically a noun but is being used as an adjective. An example of this is the word *bit* in the identifier *bitSet*. In this case, *bit* is a noun which describes the type of *set*, i.e., it is a set of bits. So we consider it a noun-adjunct. These are found in English (e.g., compound words), but generally not annotated as their own individual part-of-speech tag. Refer to our prior work for more information [25].

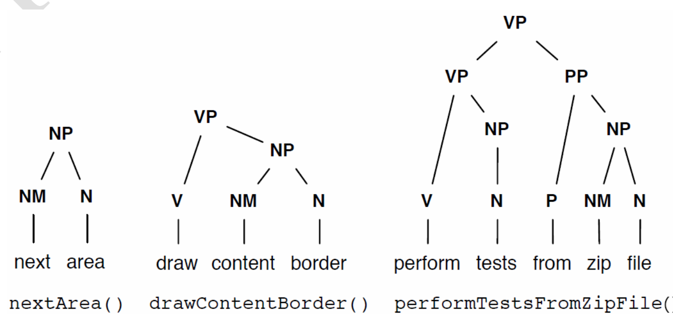

Fig. 1: Examples of noun, verb, and prepositional phrases

The tagset in Table II is a smaller set than some standard natural language part-of-speech tagsets, such as the Penn Treebank tagset used by Stanford [32], due to the fact that POSSE [17] and SWUM [16] use a limited tagset. Because SWUM/POSSE both rely on a more limited set, we use a manually curated mapping from the Penn Treebank set to our narrower tagset, provided in Table III, which we now discuss. Many of these mappings take subcategories of various part-of-speech annotations and translate them to the broadest category. For example, proper noun $\longrightarrow$ noun and modal $\longrightarrow$ verb, where a proper noun is a more specific kind of noun, and a modal is just a more specific kind of verb.

Past tense verb (VBD), present participle verb (VBG), third-person verb (VBZ), and past participle verb (VBN) are all used as adjectives in some situations within our data set. For

[1] Version: 3.9.2, taggermodel: english-bidirectional-distsim.tagger, jdk version: openJDK 11.0.7

TABLE III: How Penn Treebank annotations were mapped to the reduced set of annotations

Penn Treebank Annotation	Annotation Used In Study
Conjunction (CC)	Conjunction (CJ)
Digit (CD)	Digit (D)
Determiner (DT)	Determiner (DT)
Foreign Word (FW)	Noun (N)
Preposition (IN)	Preposition (P)
Adjective (JJ)	Noun Modifier (NM)
Comparative Adjective (JJR)	Noun Modifier (NM)
Superlative Adjective (JJS)	Noun Modifier (NM)
List Item (LS)	Noun (N)
Modal (MD)	Verb (V)
Noun Singular (NN)	Noun (N)
Proper Noun (NNP)	Noun (N)
Proper Noun Plural (NNPS)	Noun Plural (NPL)
Noun Plural (NNS)	Noun Plural (NPL)
Personal Pronoun (PRP)	Pronoun (PR)
POSSEssive Pronoun (PRP$)	Pronoun (PR)
Adverb (RB)	Verb Modifier (VM)
Comparative Adverb (RBR)	Verb Modifier (VM)
Particle (RP)	Verb Modifier (VM)
Symbol (SYM)	Noun (N)
To Preposition (TO)	Preposition (P)
Verb (VB)	Verb (V)
Verb (VBD)	Verb or Noun Modifier (V or NM)
Verb (VBG)	Verb or Noun Modifier (V or NM)
Verb (VBN)	Verb or Noun Modifier (V or NM)
Verb (VBP)	Verb (V)
Verb (VBZ)	Verb (V)

example, *sortedIndicesBuf*, *waitingList*, and *adjustedGradient* where *sorted* is a past tense verb (VBD), *waiting* is a present participle (VBG), and *adjusted* is a past participle verb (VBN). Next, when Stanford assigns List Item (LS) and Symbol (SYM) to words in our data set, it is typically mis-annotating nouns, so we map these to a noun. We discuss more about these issues in Section IV. Even with our mapping, there are a few annotations that one or more taggers do not support. POSSE does not support NPL, CJ, or PRE and groups P, PR, and DT under a single annotation; closed list. SWUM does not support NPL, VM, or CJ. Stanford does not support PRE.

B. Noun, Verb, and Prepositional phrases

There are a few linguistic concepts that come up when we analyze part-of-speech tagger output. Specifically, we will be dealing with noun phrases, verb phrases, and prepositional phrases. We define these terms and give an example. A Noun Phrase (NP) is a sequence of noun modifiers, such as noun-adjuncts and adjectives, followed by a noun, and optionally followed by other modifiers or prepositional phrases [8]. The noun in a noun phrase is typically referred to as a *head-noun*; the entity which is being modified/described by the words to its left [5] (or, for programmers, sometimes on either side of the noun-phrase) in the phrase. A Verb Phrase (VP) is a verb followed by an NP and optionally a Prepositional Phrase (PP). A PP is a preposition plus an NP. PPs can also be part of a larger VP or NP, as seen in Figure 1.

Figure 1 presents an example NP, VP, and VP with PP for three method name identifiers. The phrase structure nodes are NP, VP, and PP, while the other nodes (i.e., N, NM, V, P) are part-of-speech annotations. The leaf nodes are the individual words split from within the identifier. Each word in the identifier is assigned a part-of-speech, which can then be used to derive the identifier's phrase structure. We cannot build a phrase structure without part-of-speech information. One important thing to note about these phrases is how the words in the phrases work together. For example, in noun phrases, noun modifiers (i.e., noun-adjuncts and adjectives) work to modify (i.e., specify) the concept represented by the head-noun that is part of the same phrase. In Figure 1, *contentBorder* is a noun phrase where *content* modifies our understanding of the noun *border*. It tells us that we are talking about the content border as opposed to another type of border; a *window border*, for example. When we make it into a verb phrase by adding draw to get *drawContentBorder*; we add an action (i.e., draw) that will be applied to the particular type of border (i.e., the content border) represented by the identifier.

III. MOTIVATION & RELATED WORK

Part-of-speech tagging is an important analysis technique for understanding the meaning of words found in identifiers and the relationships between these words. Numerous prior works rely on some form of part-of-speech tagging to draw conclusions from identifier names [19], [14], [36], [13], [12], [25], [37], [21], [22], [38], [39], [31]. Thus, improved part-of-speech tagging technology can significantly improve the ability of researchers to understand the connection between identifier behavior and code behavior. Understanding this connection allows for stronger identifier name appraisal and suggestion as shown in prior research. For example, Host and Ostvold [19] use part-of-speech tags to design a method of ”debugging” (i.e., appraising) method names based partially on their part-of-speech sequence. Arnaoudova et al extends this idea by introducing linguistic anti-patterns [14], and recent literature draws inspiration, in part, from these papers to create a way of checking method naming consistency using AI [40]. All based on an initial understanding that *leveraged POS tagging*.

Arnaoudova et al also used part-of-speech tags to help categorize changes made to identifier names during rename operations [11]. Peruma et al relies on Arnaoudova's approach to understand the context around rename changes [36], [13],

[12] and identify patterns in the way names evolve. Prior work also relied on part-of-speech information to perform empirical analysis on identifiers in production code [25], [37], [21], [22], [38] to understand and taxonomize different naming structures and discuss how these varying structures are related to different types of program behavior. Part-of-speech information was also used to taxonomize test naming patterns and create test naming templates with a goal of constructing/improving tools to suggest and appraise test names [39], [31]. All of this work relies on POS tagging or the concept of POS tags.

In essence, part-of-speech tagging is an important activity in the analysis of identifier names. Improved part-of-speech techniques will provide significant support for current and future research that seeks to understand identifier naming and its connection to source code behavior. It also helps these research tasks scale to larger and larger datasets. In the rest of this section we discuss the techniques above more specifically.

A. Part-of-speech taggers

Posse [17] and Swum [16] are part-of-speech taggers created specifically to be run on software identifiers; they are trained to deal with the specialized context in which identifiers appear. Both Posse and Swum take advantage of static analysis to provide annotations. For example, they will look at the return type of a function to determine whether the word *set* is a noun or a verb. Additionally, they are both aware of common naming structures in identifier names. For example, methods are more likely to contain a verb in certain positions within their name (e.g., at the beginning) [17], [16]. They leverage this information to help determine what POS to assign different words. Stanford [32] is a popular POS tagger for general natural language (e.g., English) text. Olney et al. [18] compared taggers for accuracy on 200+ identifiers, but only on Java method names. They found that SWUM and POSSE were the most accurate taggers for source code. Newman et al [25] compared the same taggers but on a larger dataset (1,335 identifiers) and five identifier categories: function, class, attribute, parameter, and declaration statement. They found that SWUM was the most accurate overall, with an average accuracy around 59.4% at the identifier level. Thus, there is significant room for improvement. We refer the reader to [25] for a thorough evaluation and discussion of tagger accuracy on all types of identifiers. However, we provide some accuracy metrics for each tagger in our evaluation.

Our dataset supports a larger tagset than POSSE and SWUM, but a narrower tagset (besides PRE) than Stanford, whose tagset is designed for standard human language sentence structure. Stanford supports all annotations except Preamble (PRE). POSSE does not support noun plurals (NPL), conjunctions (CJ), or preambles (PRE) and it groups prepositions (P), determiners (DT), and pronouns (PR) under the same annotation, which they call *closed list*. We manually checked and determined that words annotated as *closed list* were most frequently a preposition, thus we map these to preposition in POSSE's output. This may not always be correct, but there is no way to configure POSSE to provide the specific sub-annotation; it always annotates these as *closed list*. Prior work [25] shows that putting all of these under the same category may hinder analysis of the identifier and its code context, since determiners and prepositions provide useful hints about how an identifier is related to the code surrounding it. SWUM does not support noun plurals (NPL), adverbs (VM), or conjunctions (CJ). The fact that each tagger lacks support for one or more annotations does mean that their accuracy is lower on our dataset as a result.

Posit [24] is an approach to tag mixed text; primarily motivated by the need to analyze emails and Stackoverflow text which may contain both code and standard natural language text. They use Stanford tagger [32] as part of their approach, which is appropriate since they need to tag standard natural language sentences. This tagger is solving a somewhat different problem than ours, since it must differentiate code tokens from natural language tokens and then provide the correct categorization depending on which type the token falls under. Their technique specifically annotates code tokens with AST information rather than part-of-speech information. Our approaches are complementary; our tagger may be used by theirs to tag identifiers with both part-of-speech information to augment the AST information they provide.

B. Part-of-speech-based Analysis of Identifiers

Butler's work [33] extends their previous work on Java class identifiers [37] to show that flawed method identifiers are also associated with low-quality code according to static analysis-based metrics. Caprile and Tonella [41] analyze the syntax and semantics of function identifiers. They create classes which can be used to understand the behavior of a function; grouping function identifiers by leveraging the words within them to understand some of the semantics of those identifiers. They also use the classes identified in this prior work to propose methods for restructuring program identifiers [42]. Fry and Shepherd [43], [44] study verb-direct objects to link verbs to the natural-language-representation of the entity they act upon in order to assist in locating action-oriented concerns.

Butler [21] studied class identifier names and lexical inheritance; analyzing the effect that interfaces or inheritance has on the name of a given class. For example, a class may inherit from a super class or implement a particular interface. Sometimes this class will incorporate words from the interface name or inherited class in its name. His study builds on work by Singer and Kirkham [45], who identified a grammar pattern for class names of (adjective)* (noun)+ and studies how class names correlate with micro patterns. Butler also studied Java field, argument, and variable naming structures [22]. Among other results, they identify noun phrases as the most common pattern for field, argument, and variable names. Verb phrases are the second most common.

Høst and Østvold study method names as part of a line of work discussed in Høst's disseration [46]. This line of work starts by analyzing a corpus of Java method implementations to establish the meanings of verbs in method names based on method behavior, which they measure using a set of attributes

TABLE IV: Total number per category of identifiers and unique grammar patterns across all systems

Category	Total Identifiers Across All Systems	# of Unique Grammar Patterns in Dataset
Decls	920778	45
Classes	37117	40
Functions	428748	96
Parameters	1197047	40
Attributes	159562	53
Total	2743252	277

which they define [47]. They automatically create a lexicon of verbs that are commonly used by developers and a way to compare verbs in this lexicon by analyzing their program semantics. They build on this work in [48] by using full method names which they refer to as phrases and augment their semantic model by considering a richer set of attributes. They extend this use of phrases by presenting an approach to debug method names [19]. In this work, they designed automated naming rules using method signature elements. They use the phrase refinement from their prior paper, which takes a sequence of part-of-speech tags (i.e., phrases) and concretes them by substituting real words. (e.g., the phrase <verb>-<adjective> might refine to is-empty). They connect these patterns to different method behaviors and use this to determine when a method's name and implementation do not match. They consider this a naming bug. Finally, in [49], Høst and Østvold analyzed how ambiguous verbs in method names makes comprehension of Java programs more difficult. They proposed a way to detect when two or more verbs are synonymous.

Binkley et al. [38] study grammar patterns for attribute names in classes. They come up with four rules for how to write attribute names: 1) Non-boolean field names should not contain a present tense verb, 2) field names should never only be a verb, 3) field names should never only be an adjective, and 4) boolean field names should contain a third person form of the verb "to be" or the auxiliary verb "should".

Our work is complementary to the research above in that our improved tagger can be used to increase the quality of tools and techniques they present.

IV. Methodology

In prior work [25], we constructed a dataset of 1,335 identifiers from 20 systems and manually annotated these identifiers with part-of-speech tags. For our evaluation of the ensemble tagger, we needed to create both a test set and a training set. We wanted the test set to contain identifiers from systems that were not present in the training set. Thus, we removed 5 systems and all corresponding identifiers from the original 20 system dataset and used these to create a test set. Since we wanted to maintain the same size as the original dataset, we collected additional identifiers from the remaining 15 systems so that the size of the dataset continued to be

TABLE V: Distribution of Annotations in Training and Test Sets

Annotation	Training Set	Unseen Test Set
CJ	11	1
D	20	7
DT	13	5
N	1149	321
NM	1520	414
NPL	220	78
P	91	32
PRE	83	33
V	330	81
VM	12	3
Total	3449	977

1,335 identifiers. Thus, the training set used for our ensemble is derived from our original dataset [25] but is not the exact same. The 15 system identifier set and the ensemble tagger are available through our webpage[2] and on github[3].

Below, we explain our steps as if we collected and annotated the full 15 system dataset from scratch, as opposed to deriving it from prior work and expanding it, since these are all the steps used on all identifiers in our training and test sets. Explaining it this way simplifies the discussion. One thing to note is that we assess the quality of our model in multiple ways, meaning we have multiple test sets. We name these differently to ease the burden of reading. One set is called the "unseen test set" which is made up of identifiers from systems that were not in the training set. The other test set(s) are constructed using 5-fold cross validation, which splits the 15 system training set into smaller train-test sets. We call this the "5-fold test set". We will stick to this terminology for clarity throughout this section.

A. Training Set Construction

We grouped identifiers into five categories: class names, function names, parameter names, attribute names (i.e., data members), and declaration-statement names. A declaration-statement name is a name belonging to a function-local or global variable. We use this terminology as it is consistent with srcML's terminology [50] for these variables and we used srcML to collect identifiers.

This dataset includes 1,335 identifiers which break down into 3,449 words (Table V). The number was chosen by taking a sample from the total number of identifiers at 95 confidence level and 6 confidence interval from each of the five categories we support, meaning that we sampled 267 identifiers from each category (5*267=1,335). We collected our identifier set from 15 open-source systems. We chose these systems to vary in terms of size and programming language while also being mature and having their own development communities. We

[2]https://scanl.org/

[3]https://github.com/SCANL/ensemble_tagger

TABLE VI: Systems used to create training (unbolded) and unseen test (**bolded**) sets.

Name	Size (kloc)	Age (years)	Language(s)
junit4	30	19	Java
mockito	46	9+	Java
okhttp	54	6	Java
antlr4	92	27	Java/C/C++/C#
openFrameworks	130	14	C/C++
jenkins	156	8	Java
irrlicht	250	13	C/C++
kdevelop	260	19	C/C++
ogre	370	14	C/C++
quantlib	370	19	C/C++
coreNLP	582	6	Java
swift	601	5	C++/C
calligra	660	19	C/C++
gimp	777	23	C/C++
telegram	912	6	Java/C/C++
opencv	1000	19	C/C++
elasticsearch	1300	9	Java
bullet3	1300	10+	C/C++/C#
blender	1600	21	C/C++
grpc	1800	5	C++/C/C#

did this to make sure that the identifiers in these systems have been seen by multiple contributors and that the identifiers we collected are not biased toward a specific programming language. There are two reasons for choosing identifiers from multiple languages. 1) We want to know what patterns cross-cut between languages, such that most Java/C/C++ developers are familiar with and leverage these patterns. Focusing on just one language might mean the patterns we find are not common to developers outside of the chosen language. 2) Many systems are written in more than one language, and it is important to understand how well part-of-speech tagging technologies will work on these systems. Thus, running our study systems written in different programming languages helps us study part-of-speech tagger results in an environment leveraging multiple programming languages.

We provide the list of systems and their characteristics in Table VI. The systems we picked were 615 KLOC on average with a median of 476 KLOC, a min of 30 KLOC, and a max of 1,800 KLOC. Further, most of these systems have been in active development for the past ten years or more and all of them for five years or more. The younger systems in our set are popular, modern programs. For example, Swift is a well-known programming language supported by Apple, Telegram is a popular messaging app, and Jenkins is a popular development automation server. Because we are trying to train an ensemble part-of-speech tagger to work well on as much code as possible, our goal is not necessarily to include only high-quality identifier names, but to include names that are closely representative of the average name for open-source systems. Additionally, we remove identifiers that appear in test files, in part because they sometimes have specialized naming conventions (e.g., include the word ‘test’, ‘assert’, ‘should’, etc). This is supported by other research on test names [51], [52], [39]. We exclude test-related identifiers by ignoring annotated test files and directories; any directory, file, class, or function containing the word *test*. While it is possible that identifiers in test code have similar grammar patterns to identifiers outside of test code, it is also possible that they do not. We did not want to risk introducing divergent grammar patterns. We think it would be appropriate to create a separate dataset of test identifier grammar patterns.

To collect the 1,335 identifiers, we scanned each of our 15 systems using srcML [50] and collected both identifier names/types and the category that they fell into (e.g., class, function). Then, for each category, we randomly selected one identifier from each system using a round-robin algorithm (i.e., we picked a random identifier from system 1, then randomly selected an identifier from system 2, etc. until we hit 267). This ensured that we got either 17 or 18 identifiers from each system (267/15 = 17.8) per category and mitigates the threat of differing system size.

The dataset is balanced in terms of system (i.e., equal number of observations from each system) and in terms of code category (i.e., equal number of function names, parameter names, etc). However, the dataset is **not** balanced in terms of annotation. As shown in Table V, there are a different number of observations for each annotation we support in our tag set. We do not balance it for 3 reasons: 1) the unbalanced nature of the dataset mirrors reality more accurately; some annotations are much less common than others in English. 2) Balancing it would cause our data to be significantly different from the average distribution of identifier names [25]. 3) Because no automated tagger is 100% accurate, it would not be possible to automatically balance the dataset.

We did not expand abbreviations. The reason for this is that abbreviation expansion techniques are not widely available (e.g., cannot be easily integrated into different languages or frameworks, cannot be readily trained, are not fully or publicly implemented) and still not very accurate [53]. Therefore, a realistic worst-case scenario for developers and researchers is that no abbreviation-expansion technique is available to use; their part-of-speech taggers must work in this worst-case scenario. We also tried not to split domain-term abbreviations (e.g., some splitters will make IPV4 into IPV 4; we corrected this back to IPV4). We did this because some taggers may recognize these domain terms. It is also the view of the authors that they should be recognized and appropriately tagged in their abbreviated (i.e., their most common) form.

As stated in Section II-A, some verb forms are used as either adjectives of verbs. Stanford tagger is the only tagger that recognizes derivative verb forms such as past-tense or modal. Thus, it is the only one we need to configure. In prior work [25] we measured how Stanford’s accuracy is influenced in different contexts when we assume that verbs are being used as adjectives or verbs. In short, Stanford's accuracy increases when we assume that verb conjugations are adjectives in every context except function names. For function names, it is better to assume that its verb annotations are verbs. Thus, our training set reflects this reality.

Finally, when we apply the Stanford tagger to function names, we append the letter *I* to the beginning of the name. This is a known technique– the Stanford+I technique, used to help Stanford tag identifiers that represent actions more accurately. It was used in previous studies applying part-of-speech tags to method identifier names [18], [25], [38], [54] to increase Stanford's accuracy and confirmed to increase Stanford's accuracy on function names by Newman et al. [25].

B. Unseen Systems Test Set

Our test set is made up of 384 identifiers that break down into 977 words (Table V) grouped by the same five categories used for the training set. It is constructed from identifiers contained in 5 systems that were removed from the original dataset [25], as explained at the beginning of Section IV and shown **bolded** in Table VI. We based the number, 384, on a sample at 95 confidence level and 5 confidence interval. The population from which the sample was derived is the full set of identifiers across all categories. The size and breakdown of the full population, 2,743,252 identifiers, is found in Table IV. Given this sample size, we collected 76 or 77 (384/5 = 76.8) identifiers from each category. These were balanced for category as well as system (i.e., an equal number of identifiers from each system). Like the training set, these identifiers were manually annotated with part-of-speech tags by three of the authors; each author taking a set to annotate on their own, and cross-validated by swapping sets to confirm (i.e., agree or disagree) that the manual annotation is correct. The authors came to full agreement on each identifier.

There are multiple versions of this dataset. All of them have the same identifiers, but the annotations change based on which configuration was used to generate the set. The configurations come in pairs. One pair is *normalized* or *conjugated* and the other pair is *augmented* or *plain*. *Normalized* datasets are those which convert all verb conjugations detected by standford to standard verbs (V in Table II). *Conjugated* datasets are the opposite; they used all Stanford's verb conjugations. *Augmented* datasets remove all annotations that have a frequency less than 25 and replaced them with an OTHER category. We use this to study whether rarely-seen part-of-speech tags have a negative effect on the overall quality of the tagger. *Plain* datasets include all annotations shown in Table II.

C. 5-fold test set

In addition to the unseen test set, use k-fold cross validation to help us understand the generality of our ensemble tagging model. Typically, using k-fold cross validation, prior researchers choose k as either 5 or 10. In this work, we choose 5 since it is most appropriate considering the distribution of annotations and the size of our dataset. The 5-fold test set is constructed from the training set of 1,335 identifiers. Effectively, the training set is split into five smaller sets. 30% of these is chosen as a testing set and the other 70% are used as training. Then, after training on four and testing on one, following typical 5-fold cross validation procedure, other sets are chosen as the test sets and the rest (now including some data that was just used as testing) are used to train. At each train-test step, we collect metrics about the effectiveness of our model as discussed in the next subsection.

D. Measuring Model Quality

We measure the quality of our ensemble using typical metrics for categorization problems. That is, we use Accuracy, Precision, Recall, and F1 Score. In addition to the metrics above, we also report balanced accuracy in our 5-fold results, which is similar to accuracy except we calculate the average proportion of correct predictions for each individual annotation (i.e., N, NM, CJ, etc.) first and then divide by the number of annotations. Balanced accuracy helps when dealing with unbalanced datasets by giving more weight to annotations with lower frequency in the dataset.

E. Choosing and training machine learning approaches

For the evaluation of our ensemble, we chose to use Random Forest and Decision Tree as our primary machine learning approaches. Initially, we considered Support Vector Classification, Logistic Regression, K-Nearest Neighbors, and Multinomial Naive Bayes. However, our preliminary analysis shows that Random Forest and Decision Tree outperforms the other classifiers in terms of our model quality metrics. Hence, we focus on evaluating the quality of our approach to using these two algorithms. To build our optimized model, we first split the dataset into a training and test set. The training set comprises of 70% observations, while the remaining 30% observations were part of the test set as validation data. To ensure that we are constructing an optimized model, we perform a hyperparameter optimization process. The purpose of this process is to evaluate a series of parameter values associated with the model to determine the set of values that result in the best performance of the model [55]. Our hyperparameter tuning process involved an exhaustive grid search [56] and 5-fold cross-validation on the training dataset. Grid search utilizes a brute force technique to evaluate all combinations of hyperparameters to obtain the best performance. Provided in Table VII, are the optimal hyperparameter values for the classification algorithms in our study.

F. Dataset preparation and Features

To prepare the datasets to be annotated by our ensemble tagger, we first needed to run the three part-of-speech taggers that our ensemble uses on the dataset. That is, we run SWUM [16], POSSE [17], and Stanford [32] on each identifier to obtain their annotations. We provide any information required by the three taggers above (e.g., SWUM requires function signatures). Since the dataset is pre-split from prior work [25],

TABLE VII: Optimal parameter values for the classification algorithms.

Algorithm	Parameter	Value
Random Forest	max_depth	83
	n_estimators	250
	criterion	gini
	bootstrap	True
Decision Tree	criterion	entropy
	max_depth	9

we do not have to worry about splitting. In addition, as stated prior, the correct annotation is already provided. After we have run these three taggers on the data, we vectorize the data by splitting all identifiers into their individual words along with the part-of-speech provided to them by SWUM, POSSE, Stanford, and the human annotators. In addition, we collect several data characteristics to serve as features to help our ML approaches correctly annotated the data. We explain these characteristics now.

Machine learning algorithms use characteristics of the data they are trained on to learn the nuances of that data such that they are able to use these characteristics to categorize unseen data. These characteristics are typically called Features. Our ensemble uses several features of identifiers to annotated (i.e., categorize) identifiers using part of speech tags. The features that we considered for our model are based on empirical results from a prior study we performed on the grammar patterns latent in source code identifiers [25]. To summarize, we found that certain annotations are heavily correlated with 1) words that appear in certain positions. For example, nouns appear at the end of an identifier, noun-adjuncts appear at the beginning or middle; 2) with the type of an identifier. For example, verbs are more common in boolean-type identifiers; and 3) certain contexts. For example, verb phrases are more common in function names. We also noticed that certain taggers are better at recognizing certain annotations than others. For example, SWUM is great at recognizing noun-adjuncts, Stanford is great at recognizing conjunctions and prepositions. Therefore, we chose features that will help our ensemble take advantage of these patterns. Most of these features are also very easy to obtain using static analysis, making them very accessible in different environments and, thus, helping guarantee ease of integrating our approach into another applications. The features that we considered for our model are as follows:

1) **Word** - The word itself.
2) **Data Type** - the type (or return type) of an identifier (or function identifier).
3) **SWUM annotation** - The annotation that the SWUM POS tagger applied to a given word.
4) **POSSE annotation** - The annotation that the POSSE POS tagger applied to a given word.
5) **Stanford annotation** - The annotation that the Stanford POS tagger applied to a given word.
6) **Position** - The position of a given word within its original identifier. For example, given an identifier: *GetXMLReaderHandler*, *Get* is in position 1, *XML* is in position 2, *Reader* is in position 3 and *Handler* is in position 4.
7) **Identifier size** - The length, in words, of the identifier of which the word was originally part.
8) **Normalized position** - We normalized the position metric described above such that the first word in the identifier is in position 1, all middle words are in position 2, and the last word is in position 3. For example, given an identifier: *GetXMLReaderHandler*, *Get* is in position 1, *XML* is in position 2, *Reader* is in position 2 and *Handler* is in position 3. The reason for this feature is to mitigate the sometimes-negative effect of very long identifiers.
9) **Context** - The dataset contains five categories of identifier name: function, parameter, attribute, declaration, and class. We provide the category to which the given identifier belongs as one of the features to allow the ensemble to learn patterns that are more pervasive for certain identifier types versus others. For example, function identifiers contain verbs at a higher rate [25], [17], [16], [19] than other types of identifiers.

We tested all of these features using 5-fold cross validation and the metrics that we used to measure model quality (Described in Section IV-D). Specifically, we were trying to determine what set of features maximized F1, Accuracy, and Balanced accuracy. To do this, we used 2 techniques: **Drop-column feature importance** and **permutation importance**. Drop-column feature importance can by calculated by creating a power set (i.e., all subsets) of the full set of features and then retraining your model with each subset. In this way, we can consider every possible subset of features for our feature set and determine which subset gives us the best performance with respect to F1, Accuracy, and Balanced Accuracy. While performing Drop-column feature importance, we also performed permutation importance. Permutation importance is done after a model has been fitted. It is defined as the decrease in a model score (i.e., our metrics) when a single feature's value is randomly shuffled. In essence, it measures how our metrics change when a feature is shuffled. Thus, for each subset of features in our ensemble, we also measure permutation importance.

Since there are a lot of subsets (i.e., power set of our 9 features is $2^9 = 512$), we only present data about the best feature set: **SWUM**, **POSSE**, and **Stanford** annotations, **Normalized position**, and **Context**. In addition, we present the permutation importances for these features in Tables VIII and IX. These tables correspond to permutation importances for our best Random-Forest-based ensemble and our best Decision-Tree-based ensemble. In each table, you can see how each of the best features influenced F1, Balanced Accuracy, and Accuracy. Since we used 5-fold validation, there are 5 values in each row followed by an average of those values. *A higher number means a feature is more important.* In general, out of our three taggers, SWUM had the highest influence on F1 and Accuracy, while Stanford had the highest influence on Balanced Accuracy. Of the non-tagger features

TABLE VIII: Decision Tree feature importances for best features

Feature Set	F1 Weighted Importances					Average
SWUM	0.26	0.26	0.25	0.26	0.26	0.26
POSSE	0.15	0.14	0.15	0.15	0.15	0.15
Stanford	0.23	0.23	0.23	0.23	0.23	0.23
Context	0.02	0.02	0.02	0.02	0.02	0.02
Nomalized Position	0.13	0.13	0.12	0.13	0.13	0.13
Feature Set	Balanced Accuracy Importances					Average
SWUM	0.29	0.27	0.25	0.28	0.27	0.27
POSSE	0.21	0.21	0.21	0.24	0.21	0.22
Stanford	0.51	0.51	0.49	0.53	0.51	0.51
Context	0.05	0.07	0.05	0.07	0.07	0.06
Nomalized Position	0.10	0.10	0.08	0.10	0.08	0.09
Feature Set	Accuracy Importances					Average
SWUM	0.26	0.26	0.26	0.26	0.26	0.26
POSSE	0.14	0.13	0.13	0.13	0.13	0.13
Stanford	0.22	0.21	0.20	0.20	0.21	0.21
Context	0.02	0.02	0.02	0.02	0.02	0.02
Nomalized Position	0.13	0.13	0.14	0.13	0.14	0.13

TABLE IX: Random Forest feature importances for best features

Feature Set	**F1 Weighted Importances**					**Average**
SWUM	0.22	0.22	0.21	0.21	0.22	0.21
POSSE	0.14	0.15	0.15	0.14	0.15	0.14
Stanford	0.21	0.21	0.22	0.21	0.21	0.21
Context	0.03	0.03	0.03	0.03	0.03	0.03
Nomalized Position	0.17	0.16	0.15	0.16	0.15	0.16
Feature Set	**Balanced Accuracy Importances**					**Average**
SWUM	0.22	0.22	0.21	0.26	0.25	0.23
POSSE	0.23	0.23	0.20	0.23	0.21	0.22
Stanford	0.47	0.47	0.50	0.50	0.49	0.48
Context	0.05	0.04	0.07	0.06	0.06	0.06
Nomalized Position	0.15	0.12	0.15	0.19	0.21	0.17
Feature Set	**Accuracy Importances**					**Average**
SWUM	0.23	0.22	0.21	0.22	0.22	0.22
POSSE	0.14	0.13	0.13	0.13	0.13	0.13
Stanford	0.19	0.18	0.19	0.19	0.19	0.19
Context	0.03	0.03	0.03	0.02	0.02	0.03
Nomalized Position	0.16	0.16	0.15	0.16	0.16	0.16

(i.e., Normalized Position and Context), Normalized Position had the highest influence on all three metrics. We discuss more about why these features were most important in the Evaluation Section (Section VI).

Finally, the features that were removed: Word, data type, position, and identifier size degraded our model performance. Our prior work [25] gives us some insight as to why this might be. Starting with *position* and *max position*, we found that verb and noun phrases tended to begin with a particular annotation; a verb or noun modifier respectively. They also ended with a specific annotation: a Noun (i.e., a head-noun). Between the starting verb/noun modifier and the ending head-noun are a sequence of Noun Modifiers. Notice that this correlates to a *beginning*, *middle*, *end* structure where the first word has a specific tag, all the middle words have the same tag, and the final word has a specific tag. Position and Max position confuse the ensemble because identifiers have varying lengths. Normalized position categorizes position as beginning, middle, or end. Thus, it improves the performance of the ensemble whereas position and max position hurt the performance. Word and data type can help the model recognize certain words and their correlation to different tags. However, on unseen data this may cause the ensemble to become confused because it sees words that it has not seen before. Thus, the ensemble will tag common words more accurately with these features turned on but uncommon/unseen words less accurately.

V. Evaluation Setup

The dataset described in Section IV has several configurations that we use during evaluation. These configurations are as follows:

1) The type of machine learning approach used; either Decision Tree or Random forest. They have the codes DT and RF respectively.
2) The version of the dataset being used; either the plain dataset or the augmented dataset. These have the codes P and A respectively.
3) Whether or not the Stanford data within the dataset is using verb conjugations or is normalized. These have the codes C and N respectively.

To determine which configuration you are looking at when reading our results, look at the code present in each table. For example, if some data in a table has the code RFCP, then it used **random forest** with **conjugated** stanford identifiers and the **plain** dataset.

VI. Evaluation

A. RQ1: How accurate is the ensemble part-of-speech tagger at the individual word level and how does this compare to the independent taggers?

We evaluate accuracy in two ways. The first way is by running 5-fold cross validation using the manually-curated set of 1,335 identifiers. The second way is by running our model on an unseen test set of 384 identifiers. We will split our discussion of RQ1 results into these individual evaluations.

1) 5-fold test results: The results from this evaluation are found in Table XIV and Table XV. These tables give the 5-fold results for five different metrics: accuracy, balanced accuracy, weighted f1, weighted precision, and weighted recall. We ran this 5-fold evaluation on 8 different configurations of our ensemble tagger. You can find the meanings of the abbreviations in Section V. Overall, our results indicate that Random Forest gives the best results regardless of configuration; achieving the highest average in all five metrics used to gauge the quality of our ensemble when compared to decision tree. There is one exception, which is DTCP vs RFCP in Table XV. DTCP achieves the same averages compared with RFCP except with respect to balanced accuracy and weighted f1, where DTCP has better balanced accuracy and RFCP has a better weighted f1. If we look across both tables, then the best configurations that maximize all quality metrics are: DTCP, RFCP, RFNP, and RFCA. Out of those configurations, the authors would advise that the best configuration is DTCP or RFCP. The

TABLE X: Per-annotation and Overall Accuracy of Ensemble Tagger on Augmented Dataset

		DTNA				RFNA				DTCA				RFCA			
Annotation	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total
N	332	0.87	0.89	**0.88**	329	0.86	0.89	**0.88**	331	0.84	0.89	0.87	340	0.87	0.90	**0.88**	332
NM	428	0.85	0.92	**0.89**	448	0.85	0.92	**0.89**	446	0.87	0.91	**0.89**	435	0.86	0.92	**0.89**	447
NPL	80	0.90	0.68	0.77	59	0.93	0.67	0.78	56	0.89	0.72	0.79	63	0.96	0.69	**0.81**	56
OTHER	18	0.69	0.56	0.62	13	0.73	0.69	**0.71**	15	0.56	0.31	0.40	9	0.50	0.25	0.33	8
P	34	0.70	0.81	0.75	37	0.71	0.84	0.77	38	0.68	0.78	0.72	37	0.72	0.88	**0.79**	39
PRE	32	0.64	0.21	0.32	11	0.64	0.21	0.32	11	0.78	0.21	0.33	9	0.77	0.30	**0.43**	13
V	103	0.79	0.78	0.78	80	0.79	0.78	0.78	80	0.79	0.81	0.80	84	0.80	0.81	**0.81**	82
Accuracy		**0.84**				**0.85**				**0.84**				**0.85**			

TABLE XI: Per-annotation and Overall Accuracy of Ensemble Tagger on Plain Dataset

		DTNP				RFNP				DTCP				RFCP			
Annotation	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total
CJ	1	0.50	1.00	0.67	2	1.00	1.00	**1.00**	1	1.00	1.00	**1.00**	1	1.00	1.00	**1.00**	1
D	7	0.88	1.00	**0.93**	8	0.88	1.00	**0.93**	8	0.88	1.00	**0.93**	8	0.88	1.00	**0.93**	8
DT	5	1.00	0.20	0.33	1	1.00	0.40	0.57	2	1.00	0.60	**0.75**	3	1.00	0.60	**0.75**	3
N	322	0.86	0.90	0.88	338	0.87	0.89	0.88	328	0.85	0.90	0.88	340	0.88	0.89	**0.89**	327
NM	415	0.85	0.93	**0.89**	452	0.85	0.93	**0.89**	453	0.88	0.91	**0.89**	430	0.87	0.92	**0.89**	438
NPL	78	0.98	0.65	0.78	52	0.93	0.67	0.78	56	0.90	0.72	**0.80**	62	0.93	0.69	0.79	58
P	32	0.65	0.81	0.72	40	0.72	0.88	**0.79**	39	0.68	0.84	0.75	40	0.70	0.88	0.78	40
PRE	33	0.70	0.21	0.33	10	0.64	0.21	0.32	11	0.78	0.21	0.33	9	0.77	0.30	**0.43**	13
V	81	0.85	0.77	0.81	73	0.81	0.77	0.78	77	0.80	0.81	**0.81**	82	0.79	0.83	0.81	85
VM	3	0.00	0.00	0.00	1	0.50	0.33	0.40	2	0.50	0.33	0.40	2	0.50	0.67	**0.57**	4
Accuracy		**0.85**				**0.85**				**0.85**				**0.86**			

TABLE XII: Per-annotation and Overall Accuracy of the Independent Taggers on Plain Dataset - N/A = annotation not supported by tagger

		SWUM				POSSE				Stanford			
Annotation	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total
CJ	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**1.00**	**1.00**	**1.00**	1
D	7	0.33	0.14	0.20	3	N/A	N/A	N/A	N/A	**0.78**	**1.00**	**0.88**	9
DT	5	0.50	**0.60**	0.55	6	N/A	N/A	N/A	N/A	**1.00**	0.40	**0.57**	2
N	322	**0.74**	0.89	**0.81**	384	0.42	0.90	0.57	692	0.47	**0.93**	0.62	647
NM	415	0.78	**0.94**	**0.85**	500	0.82	0.21	0.33	106	**0.83**	0.09	0.17	47
NPL	78	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**0.86**	**0.73**	**0.79**	66
P	32	**0.88**	0.44	0.58	16	0.61	0.63	0.62	33	0.58	**0.91**	**0.71**	50
PRE	33	**0.50**	**0.03**	**0.06**	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
V	81	**0.89**	0.72	**0.79**	65	0.77	0.75	0.76	79	0.51	**0.90**	0.65	44
VM	3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**0.30**	**1.00**	**0.46**	10
Accuracy		**0.77**				**0.47**				**0.52**			

TABLE XIII: Accuracy of independent taggers and best two ensemble taggers in different contexts at the word-level

	Total	DTCP	RFCP	SWUM	POSSE	Stanford
Attribute	194	0.82	**0.83**	0.72	0.42	0.45
Class	200	0.87	**0.89**	0.84	0.43	0.40
Declaration	184	0.83	**0.84**	0.79	0.48	0.45
Function	231	**0.85**	**0.85**	0.74	0.55	0.75
Parameter	168	**0.90**	0.89	0.77	0.45	0.53
Overall	977	0.86	0.86	0.77	0.47	0.52

reason for this is that these configurations 1) are the most accurate on average; 2) use Stanford's conjugations instead of normalizing them away, meaning that they require less dataset preparation; and 3) they use the plain dataset, meaning that they are operating on the full tagset instead of grouping low-frequency annotations together under the OTHER category. This allows them to provide these annotations when they are used to tag identifiers.

2) Unseen test set results: The results from this evaluation are found in Table X and Table XI. Each table shows the

TABLE XIV: Five-fold validation results for each machine learning approach and configuration

			DTNA			Average			RFNA			Average
Accuracy	0.80	0.84	0.79	0.84	0.82	**0.82**	0.81	0.83	0.79	0.84	0.84	**0.82**
Balanced Accuracy	0.54	0.68	0.65	0.66	0.59	0.62	0.57	0.66	0.71	0.65	0.75	**0.67**
Weighted F1	0.79	0.83	0.79	0.83	0.81	0.81	0.80	0.82	0.79	0.84	0.84	**0.82**
Weighted Precision	0.80	0.82	0.80	0.83	0.82	0.81	0.81	0.82	0.80	0.84	0.84	**0.82**
Weighted Recall	0.80	0.84	0.79	0.84	0.82	**0.82**	0.81	0.83	0.79	0.84	0.84	**0.82**
			DTCA			**Average**			**RFCA**			**Average**
Accuracy	0.82	0.85	0.81	0.84	0.84	0.83	0.84	0.86	0.81	0.85	0.86	**0.84**
Balanced Accuracy	0.51	0.62	0.55	0.53	0.58	0.56	0.52	0.69	0.62	0.48	0.65	**0.59**
Weighted F1	0.82	0.84	0.80	0.82	0.82	0.82	0.83	0.86	0.80	0.84	0.86	**0.84**
Weighted Precision	0.83	0.84	0.81	0.82	0.82	0.82	0.83	0.87	0.80	0.85	0.86	**0.84**
Weighted Recall	0.82	0.85	0.81	0.84	0.84	0.83	0.84	0.86	0.81	0.85	0.86	**0.84**

TABLE XV: Five-fold validation results for each machine learning approach and configuration using the plain dataset

			DTNP			Average			RFNP			Average
Accuracy	0.81	0.81	0.86	0.82	0.82	0.82	0.84	0.82	0.87	0.82	0.85	**0.84**
Balanced Accuracy	0.54	0.59	0.71	0.75	0.53	0.62	0.61	0.60	0.76	0.74	0.60	**0.66**
Weighted F1	0.79	0.80	0.86	0.81	0.80	0.81	0.82	0.82	0.87	0.81	0.84	**0.83**
Weighted Precision	0.80	0.80	0.87	0.82	0.79	0.82	0.82	0.82	0.87	0.81	0.85	**0.83**
Weighted Recall	0.81	0.81	0.86	0.82	0.82	0.82	0.84	0.82	0.87	0.82	0.85	**0.84**
			DTCP			**Average**			**RFCP**			**Average**
Accuracy	0.85	0.82	0.83	0.84	0.85	**0.84**	0.86	0.83	0.83	0.85	0.85	**0.84**
Balanced Accuracy	0.58	0.67	0.73	0.60	0.63	**0.64**	0.51	0.59	0.54	0.50	0.63	0.55
Weighted F1	0.85	0.82	0.83	0.84	0.84	0.83	0.86	0.82	0.82	0.84	0.84	**0.84**
Weighted Precision	0.85	0.81	0.83	0.83	0.83	**0.83**	0.86	0.82	0.82	0.83	0.83	**0.83**
Weighted Recall	0.85	0.82	0.83	0.84	0.85	**0.84**	0.86	0.83	0.83	0.85	0.85	**0.84**

TABLE XVI: Average Accuracy at the identifier-level

Category	Total	DTNA	RFNA	DTCA	RFCA	DTNP	RFNP	DTCP	RFCP
Attribute	76	0.70	0.70	**0.72**	**0.72**	0.68	0.70	**0.72**	**0.72**
Class	77	0.78	0.78	0.77	**0.82**	0.77	0.78	0.77	**0.82**
Declaration	77	0.66	0.71	0.66	0.73	**0.74**	0.71	0.69	0.71
Function	77	0.62	0.62	0.68	0.65	0.64	0.64	**0.71**	0.69
Parameter	77	0.79	0.79	0.78	0.79	**0.81**	**0.81**	**0.81**	0.79
Overall	**384**	0.71	0.72	0.72	0.74	0.73	0.73	0.74	**0.75**

TABLE XVII: Average accuracy at the identifier-level for state-of-the-art POS taggers

Category	Total	SWUM	POSSE	Stanford
Attribute	76	0.54	0.17	0.17
Class	77	0.61	0.13	0.19
Declaration	77	0.65	0.27	0.22
Function	77	0.48	0.30	0.29
Parameter	77	0.61	0.14	0.32
Overall	**384**	0.58	0.20	0.24

precision, recall, f1, and accuracy of each ensemble configuration on each individual annotation supported by the ensemble. Table X has fewer annotations since less-frequent annotations are grouped under the AUGMENTED category. Whereas Table XI contains all annotations supported in our tagset even if they were very infrequent in the dataset. In addition, these tables show the overall accuracy for each ensemble tagger configuration on the unseen test set. This accuracy is obtained by measuring how many words the ensemble tagger annotated correctly when compared to the manual annotations. One thing to note about these results is that the *total* column on the leftmost side of each table is the total of each annotations in the manually-annotated (i.e., gold) set. Therefore, each configuration would have a different distribution since they each incorrectly annotated some words.

The overall results agree with our 5-fold results. That is, the best taggers tend to be random forest based ensembles. Again, with the exception of DTCP, which is competitive with the other random forest ensembles. RFCP does marginally better than DTCP on the unseen test set, achieving an accuracy of .86 versus DTCP's .85. In addition, since they are trained on the plain dataset, they could be used to annotate tags that are less frequent in production code: DT, CJ, VM, and D; each of which were grouped into the OTHER category in the dataset that RFCA was trained on. It is also notable that the most accurate ensemble configurations were trained on the plain dataset, indicating that the greater tag granularity helped improve the ensemble's output. We come to the same conclusion in this analysis as we did in the 5-fold analysis; DTCP and RFCP are the best ensemble for all of the same advantages explained above alongside retaining the best average values on our metrics. However, RFCP is marginally the better of the two based on our results. One of the primary reasons we still include DTCP as a competitive alternative to RFCP is that DTCP is faster and would scale better for large datasets.

3) Comparison with independent taggers: Table XII shows the accuracy of the independent taggers at the word-level. Comparing the data in this table to Table XI, both DTCP and RFCP outperform or match the other taggers in every individual category. In addition, from Table XIII, we see that DTCP and RFCP maintain their performance advantage at the context level as well.

4) Discussion of Feature Importance: In prior work [25], we noticed that the individual taggers had strengths and weaknesses that complemented one another. Specifically, Stanford was able to annotate Conjunctions, Digits, Determiners, Noun Plurals, Prepositions, and Verb Modifiers with high accuracy. Meanwhile, SWUM and POSSE tended to outperform Stanford in annotating Noun Modifiers and Verbs. Thus, Stanford's higher balanced accuracy makes perfect sense; it is very complementary to SWUM and POSSE. In addition, we also noticed that word position is important to annotating certain tags, such as Noun Modifiers. This is because, in a noun phrase, the leftmost words tend to be Noun Modifiers while the right-most word is a Noun (i.e., specifically, a head-noun). Another example is that Verbs tend to be in the first position in a function name.

Providing the normalized position helps the ensemble learn these patterns. Normalized position tends to be more effective at this than plain position because normalized position identifies the beginning, middle, and end of an identifier specifically. In contrast, raw position confuses the ensemble since identifiers can be of varying length, making it difficult to identify where the middle and end of an identifier are. Context is, surprisingly, not as important as normalized position. However, it is still part of the best feature set, meaning that it performs better than the subset of features that excludes context but includes normalized position. Thus, knowing whether an identifier is a function name, parameter, etc is still important for annotating with part-of-speech using our approach.

In summary, we have taken two different approaches to evaluate our ensemble tagger. In addition, each of these approaches used a different dataset of unseen data to ensure that the ensemble is as general as possible; that the results from its evaluation will translate well to other unseen situations. Based on our data, DTCP and RFCP are equivalent in terms of average performance and have some minor differences between them when we look at which annotations they are most effective on. There is one advantage that DTCP has over RFCP that may be worth mentioning: it is faster. Since decision trees tend to be smaller than random forests, DTCP generally annotates more rapidly than RFCP.

B. RQ2: How accurate is the ensemble part-of-speech tagger at the identifier level and how does this compare to the independent taggers?

In RQ1, we explore the accuracy of our ensemble tagger at the level of individual words. That is, we want to know how many words it correctly annotates in our dataset. However, the number of correctly annotated words does not give us the full picture. Since most identifiers in the code are made up of multiple words, it is also important to understand how accurate our ensemble tagger is on full identifiers. For this reason, we took the unseen test set and analyzed how effective our ensemble was on full identifier names.

The results of this analysis are given in Table XVI and Table XVII. Table XVI shows the accuracy of our individual ensemble configurations, broken down by the five categories we used in our training set: attribute, class, declaration, function, and parameter. In addition, it shows the total number of each type of identifier in the dataset. Table XVII shows the individual accuracy of the three state-of-the-art part-of-speech taggers used to construct our ensemble. The numbers here are lower than in the prior tables from RQ1 because we are measuring full identifier names; if even a single word in the identifier name is mis-annotated, then we consider it incorrect.

Our results show that the overall accuracy of our ensemble on full identifier names is unsurprisingly lower than on individual words; about 11-13% lower in general. However, the ensemble still performs better than its closest competition according to both prior work [25] and our own analysis shown in Table XVII, where we show the accuracy of individual part-of-speech taggers on the same unseen test set on which we ran our ensemble. Comparing the performance of the ensemble and the individual part-of-speech taggers, we can see that the best configuration of our ensemble (i.e., RFCP) outperforms the best tagger, SWUM, by around +17% on average while it outperforms POSSE and Stanford by 55% and 51% respectively.

In summary, these results are promising, but not surprising. Our ensemble is trained using the output of these approaches, so we would expect that it is able to learn their mistakes and produce output at a higher accuracy than its constituent taggers. We have shown that we can use an ensemble approach to improve upon part-of-speech tagging approaches on source code identifiers at both the individual word level and the full identifier level.

C. RQ3: What are the most frequently mis-used part-of-speech tags and grammar patterns?

We have shown the effectiveness of the ensemble tagger at the level of both individual words and full identifier names. In this research question, we are interested in understanding the *weaknesses* of our ensemble; where can it be improved in future work and what types of identifiers is it more likely to get wrong? To answer this question, we calculated the patterns that were most frequently mis-annotated by our ensemble along with the frequency of these patterns in our data. We then divided the number of mis-annotations by the pattern frequency to get the proportion and sorted from largest to smallest. Table XVIII shows the top five mis-annotated grammar patterns per ensemble configuration along with the frequency and proportion information discussed above.

The data in this table shows some consistently mis-annotated patterns. In particular, *NM NM NM+ N*, *PRE NM* N*, and *V NM NM N* are all in the top 5 for each ensemble configuration. Where '+' means "one or more" of the annotation to its left and '*' means "zero or more" of the annotation to its left. A high frequency of mis-annotating *PRE NM* N* is unsurprising due to the fact that most ensemble configurations had trouble with annotating PRE; the best ensemble configuration achieving only .043 F1. Note that this low F1 score means that it **both** mis-annotates some NM+ N patterns by annotating PRE where it should have annotated NM as well as mis-annotating PRE NM+ N patterns as NM+ N; not recognizing the first word as a PRE and instead annotating NM. This fact helps explain one of the other patterns it frequently mis-annotates– the elongated noun-phrase patterns (*NM NM NM+ N*), since it is typically mis-annotating the leftmost NM as PRE. The other pattern it gets wrong frequently is the elongated verb phrase pattern. The ensemble seems to get shorter verb phrase patterns correct but the longer they become the harder it becomes for the ensemble to annotate them. One reason for this is likely the lack of verb phrases that are greater than four and five words in the training set. This may also have something to do with the fact that we use **normalized position** as one of our features, as discussed in SectionIV-F. This feature normalizes the length of an identifier by considering words to one be in one of three places: the beginning, the middle, or the end. This helps recognize the fact that the first and last words in an identifier are more likely to be a specific annotation (e.g., the last noun in an identifier is usually a head-noun, whereas middle-nouns are typically noun-adjuncts).

We manually looked at examples of each of these commonly mis-annotated patterns to understand the characteristics of these types of identifiers that the ensemble finds confusing. To make this analysis simpler, we will focus on the best ensemble configurations: DTCP and RFCP. Our manual analysis of the data shows that the most confusing factor in most of these patterns for both RFCP and DTCP is *PRE*. That is, when these are mis-annotated, it is because the correct annotation contains a preamble. For example, *eglewAndroidFrameBufferTarget* and *mRemoveUserDataResponseArgs* both have a grammar pattern that begins with a Preamble but they are mis-annotated as *NM NM NM NM+ N*. The one exception is the *V NM NM N* pattern, which tended to be mis-annotated because the correct pattern does not follow a standard verb-phrase pattern. These are function names like *clampFixMaxcolor* (*fix* is an abbreviation for *fixpoint*) and *ActionViewShowMasterPages*, which have non-standard function naming structure; *V N NM N* and *NM N V NM NPL* respectively. This does follow results from prior work [25], as we found that functions have the largest number of unique grammar patterns; many function names may follow a non-standard format, and the further they are from a standard verb phrase, the more difficult it may be for our tagger to annotate it correctly. If the reader is interested in common mis-annotations made by POS taggers, please refer to prior work for more information on the types of mistakes [25].

1) Discussion of Feature Importance: The results to this RQ show us that more context, in the form of features, is likely required to increase the accuracy of the ensemble. Specifically, context that can 1) help identify Preambles, such as word frequency or system naming conventions. And 2) identify stereotype-like [57] information that could tell the ensemble when it might see a function name that does not follow verb phrase patterns. Of course, other types of context may also be helpful, but these are two types of context that, based on our observations, would highly-likely increase the accuracy of our ensemble.

In summary, our ensemble has more trouble with longer identifier names; particularly longer verb-phrase identifiers in general and longer noun-phrase identifiers which contain a preamble. We found that the mis-annotated verb phrase identifiers are typically function names that do not follow a standard verb-phrase structure, while the mis-annotated noun phrases tended to be elongated and contain a preamble. By far, the most confounding factor for our ensemble is when an identifier contains a preamble. Preambles are difficult for the taggers that our ensemble is trained on to recognize and therefore, also our ensemble. While the ensemble is more effective than them individually, it still only recognizes them at an F1 of .043, meaning that it could be significantly improved. Preamble detection is possible, and we plan to address it in the near future, but it requires an analysis of the system-to-be annotated before commencing annotations. Specifically, to detect preambles, we have to detect frequently-occurring identifier prefixes and then determine whether those prefixes follow the preamble rules specified in Section II.

VII. Threats to validity

We train our approach on a collection of 1,335 identifiers manually annotated by three of the authors. There is a threat

TABLE XVIII: Top 5 most frequently mis-annotated grammar patterns for each ensemble configuration

DTCA				**DTCP**			
Grammar Pattern	**# Incorrect**	**Actual**	**Proportion**	**Grammar Pattern**	**# Incorrect**	**Actual**	**Proportion**
NM NM NM NM N	6	8	0.75	NM NM NM NM N	6	8	0.75
NM NM NM N	24	38	0.63	NM NM NM N	25	38	0.66
PRE N	5	9	0.56	PRE N	5	9	0.56
V NM NM N	9	22	0.41	V NM NM N	9	22	0.41
V NM NPL	4	12	0.33	NM	2	6	0.33
RFCA				**RFCP**			
Grammar Pattern	**# Incorrect**	**Actual**	**Proportion**	**Grammar Pattern**	**# Incorrect**	**Actual**	**Proportion**
PRE N	6	9	0.67	PRE N	6	9	0.67
NM NM NM NM N	5	8	0.63	NM NM NM N	21	38	0.55
NM NM NM N	20	38	0.53	NM NM NM NM N	4	8	0.50
NM NM NM NPL	2	4	0.50	V NM NM N	9	22	0.41
V NM NM N	9	22	0.41	PRE NM N	4	12	0.33
DTNA				**DTNP**			
Grammar Pattern	**# Incorrect**	**Actual**	**Proportion**	**Grammar Pattern**	**# Incorrect**	**Actual**	**Proportion**
NM NM NM NM N	7	8	0.88	NM NM NM NM N	7	8	0.88
NM NM NM N	29	38	0.76	NM NM NM N	31	38	0.82
PRE N	6	9	0.67	PRE N	5	9	0.56
V NM NM N	10	22	0.45	V NM NM N	9	22	0.41
PRE NM N	4	12	0.33	V N P N	2	5	0.40
RFNA				**RFNP**			
Grammar Pattern	**# Incorrect**	**Actual**	**Proportion**	**Grammar Pattern**	**# Incorrect**	**Actual**	**Proportion**
NM NM NM NM N	6	8	0.75	NM NM NM NM N	7	8	0.88
NM NM NM N	28	38	0.74	NM NM NM N	29	38	0.76
PRE N	6	9	0.67	PRE N	6	9	0.67
V NM NM N	9	22	0.41	V NM NM N	9	22	0.41
PRE NM N	4	12	0.33	PRE NM N	4	12	0.33

that the annotated set contains imperfections. To mitigate this, we used cross-validation, where all annotators performed a validation step on all grammar patterns; each grammar pattern was validated by two annotators beside the original annotator. Additionally, the dataset is publicly available; future corrections are possible. We calculated that a statistically representative sample for the size of our dataset of 15 systems is 267 given a 95 confidence level and a confidence interval of 6. We picked 95 and 6 as a trade-off between representativeness of the sample, the amount of manual labor required of the annotators, and the sample size used in prior studies.

The dataset that the ensemble is trained on does not control for design-level decisions that may influence naming structure such as design or architectural patterns used by the systems in our dataset. In addition, the dataset does not include test identifiers. This means that the tagger may not be as effective on test identifiers. In addition, while our 5-fold validation indicates that our ensemble works well on a general set of identifier names, identifiers that are written to specific naming conventions due to project-specific, or domain-specific requirements may confuse the ensemble and cause it to perform less-effectively than our 5-fold analysis indicates.

Related to the prior paragraph, we do not address the issue of annotating programming-language-specific naming patterns in this paper. However, our prior work [25] provides some insight into this problem. In that paper, we were unable to find a significant difference in the grammar patterns present in C/C++ and Java systems. This does not mean that there are not any, as stated in the same paper [25], but that our dataset and the variables we controlled for in constructing the dataset did not show a difference. However, we did notice that C/C++ systems tend to have more abbreviations in their identifiers than Java systems, which can, and did [25], hinder the effectiveness of part-of-speech tagger performance. Thus, effectiveness of a part-of-speech tagger can vary in different programming languages due to certain identifier name characteristics, such as the presence of abbreviations. We cannot draw conclusions, but future work should more thoroughly investigate the influence of programming language and other system characteristics on naming patterns and part-of-speech tagging.

Overfitting is a threat for the ensemble itself. We mitigate the threat of overfitting by evaluating using 5-fold cross validation, which creates sets of train-test folds to help validate the generality of approaches like our ensemble tagger. In addition, we constructed a manually annotated testing set of identifiers from systems that are not in the training set. Our ensemble's performance is consistent on unseen data from unseen systems, therefore the risk of overfitting is low.

We did consider other machine learning approaches. Specifically, we considered Support Vector Classification, Logistic Regression, K-Nearest Neighbors, and Multinomial Naive Bayes alongside Random Forest and Decision Trees. We performed a preliminary evaluation using 5-fold cross validation and hyper-tuning. Decision Trees and Random Forest outperformed these other approaches in terms of F1, Accuracy, and Balanced accuracy. However, our preliminary analysis should not be considered the final word on whether these other algorithms could be used to perform POS tagging accurately, because there may be other configurations or features which we have not considered in this paper. To help mitigate the threat that there are other/better ways to create and apply the ensemble, we have made our data and the ensemble tagger publicly available for both use and further research (see Section IV). It is worth noting that our purpose in this paper is not to prove the indisputably **best** method and feature set for creating an ensemble tagger, but rather to create a useful tool for the research community that outperforms what is currently available and supports future research to improve the state-of-the art both in terms of POS tagging and our understanding of identifier naming practices and patterns. We will continue incrementally improving the ensemble through further research outlined in the Discussion section.

VIII. Discussion & Conclusion

In this paper, we have described and evaluated an ensemble tagging technique which combines the output of three state-of-the-art part-of-speech taggers using the random forest and decision tree machine learning algorithms. We have shown that our approach improves the output of these taggers by learning their common mistakes and adjusting to correct these mistakes. The evaluation shows that the ensemble tagger achieves an accuracy, at the identifier level, up to 75%; a +17% point increase on top of the best runner-up tagger's accuracy (i.e., SWUM at 58%). In addition, our ensemble approach achieves up to 86% accuracy at the word-level.

We use several features to achieve this quality, including the annotations from SWUM, POSSE, and Stanford; the context of the identifier (e.g., whether it is a function or class identifier), and the normalized position of each word within the identifier. These features were compared against four other features: word, data type, position, and max position. We found that the addition of these features degraded the performance of the ensemble, so we did not use them in the final version of our ensemble tagger. While this does not mean that they are bad features to include in future research, it does indicate that they can negatively impact performance for our approach and should be used carefully if they are used at all.

Our ensemble has several potential weaknesses which must be evaluated and accounted for in future research. These weaknesses are as follows:

1) **Identifiers containing preambles, and non-verb-phrase function names.** Based on RQ3, our approach may have decreased accuracy on identifier names that grow past the average identifier size and are 1) function names that do not follow a standard verb phrase pattern or 2) elongated noun-phrases that contain a preamble.
2) **Test identifier names.** Our ensemble is trained on a data set that does not include test identifier names. Therefore, under the supported [51], [51] assumption that test identifiers have a different structure than production identifiers, our approach will be less accurate on these identifiers under its current training set.
3) **Design-, or architecture-, specific identifier structures.** In prior work, we noticed that certain identifier naming structures seem to indicate the use of certain design or architectural patterns [25]. However, we did not control for these in the construction of the dataset that is used to train this ensemble approach and there is no dataset of part-of-speech tagged identifiers that controls for these aspects to our knowledge. It is possible that the accuracy of our approach is negatively influenced due to not taking these design decisions into account.

Therefore, in future work, we plan to expand our dataset to handle these concerns. This will have the advantage of both increasing the size of our dataset as well as increasing our understanding of how different software development contexts influence identifier names.

In conclusion, we have presented an ensemble part-of-speech tagger for source code identifiers. The ensemble uses several features of identifiers, as well as three state-of-the-art part-of-speech taggers, to improve upon the quality of identifier part-of-speech annotations. In addition, we will release the tool to be used by other researchers. Our plan is to integrate it with the srcML framework [50] in the very near future. We are also investigating ways to remove its reliance on other taggers' output and create a fully self-contained part-of-speech tagger for source code.

IX. Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 1850412. We would also like to thank Aidan White for initial help with the manual-annotation effort.

X. Disclaimer

The views expressed in this paper are not affiliated or endorsed by BNY Mellon in any way.

REFERENCES

[1] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[2] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[3] A. von Mayrhauser and A. M. Vans, "Program understanding behavior during debugging of large scale software," in *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*, ser. ESP '97. New York, NY, USA: ACM, 1997, pp. 157–179. [Online]. Available: http://doi.acm.org/10.1145/266399.266414

[4] M. Fisher, A. Cox, and L. Zhao, "Using sex differences to link spatial cognition and program comprehension," in *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ser. ICSM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 289–298. [Online]. Available: https://doi.org/10.1109/ICSM.2006.72

[5] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, p. 261–282, Sep. 2006. [Online]. Available: https://doi.org/10.1007/s11219-006-9219-1

[6] D. Binkley, D. Lawrie, and C. Morrell, "The need for software specific natural language techniques," *Empirical Softw. Engg.*, vol. 23, no. 4, pp. 2398–2425, Aug. 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9566-5

[7] R. Jongeling, P. Sarkar, S. Datta, and A. Serebrenik, "On negative results when using sentiment analysis tools for software engineering research," *Empirical Software Engineering*, 01 2017.

[8] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[9] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[10] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE 2019. New York, NY, USA: ACM, 2019.

[11] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc, "Repent: Analyzing the nature of identifier renamings," *IEEE Trans. Softw. Eng.*, vol. 40, no. 5, pp. 502–532, May 2014. [Online]. Available: https://doi.org/10.1109/TSE.2014.2312942

[12] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "Contextualizing rename decisions using refactorings and commit messages," in *Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2019.

[13] A. Peruma, M. W. Mkaouer, M. Decker, and C. Newman, "Contextualizing rename decisions using refactorings, commit messages, and data types," *Journal of Systems and Software*, vol. 169, p. 110704, 06 2020.

[14] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y. Guéhéneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *2013 17th European Conference on Software Maintenance and Reengineering*, March 2013, pp. 187–196.

[15] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, "An empirical study of identifier splitting techniques," *Empirical Softw. Engg.*, vol. 19, no. 6, pp. 1754–1780, Dec. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9261-0

[16] E. Hill, "Integrating natural language and program structure information to improve software search and exploration," Ph.D. dissertation, Newark, DE, USA, 2010, aAI3423409.

[17] 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 *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, pp. 3–12.

[18] W. Olney, E. Hill, C. Thurber, and B. Lemma, "Part of speech tagging java method names," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 483–487.

[19] E. W. Høst and B. M. Østvold, "Debugging method names," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 294–317. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_14

[20] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 232–242.

[21] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Mining java class naming conventions," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 93–102.

[22] S. Butler, M. Wermelinger, and Y. Yu, "A survey of the forms of java reference names," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 196–206.

[23] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *In Proc. of the 18th Annual Psychology of Programming Workshop*, 2006.

[24] P.-P. Pârtachi, S. K. Dash, C. Treude, and E. T. Barr, "Posit: Simultaneously tagging natural and programming languages," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1348–1358. [Online]. Available: https://doi.org/10.1145/3377811.3380440

[25] C. D. Newman, R. S. AlSuhaibani, M. J. Decker, A. Peruma, D. Kaushik, M. W. Mkaouer, and E. Hill, "On the generation, structure, and semantics of grammar patterns in source code identifiers," *Journal of Systems and Software*, vol. 170, p. 110740, Dec 2020. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2020.110740

[26] D. H. Moore II, "Classification and regression trees, by leo breiman, jerome h. friedman, richard a. olshen, and charles j. stone. brooks/cole publishing, monterey, 1984,358 pages, $27.95," *Cytometry*, vol. 8, no. 5, pp. 534–535, 1987. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cyto.990080516

[27] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, p. 5–32, Oct. 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[28] D. Roy, S. Fakhoury, J. Lee, and V. Arnaoudova, "A model to detect readability improvements in incremental changes," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 25–36. [Online]. Available: https://doi.org/10.1145/3387904.3389255

[29] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, and M. Kessentini, "How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation," *Expert Systems with Applications*, p. 114176, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095741742030912X

[30] F. Dell'Orletta, "Ensemble system for part-of-speech tagging," in *Proceedings of EVALITA 2009*, 2009.

[31] A. Peruma, E. Hu, J. Chen, E. A. Alomar, M. W. Mkaouer, and C. D. Newman, "Using grammar patterns to interpret test method name evolution," in *Proceedings of the 29th International Conference on Program Comprehension*, ser. ICPC '21. New York, NY, USA: Association for Computing Machinery, May 2021.

[32] K. Toutanova and C. D. Manning, "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger," in *Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: Held in Conjunction with the 38th Annual Meeting of the Association for Computational Linguistics - Volume 13*, ser. EMNLP '00. Stroudsburg, PA, USA: Association for Computational Linguistics, 2000, pp. 63–70. [Online]. Available: https://doi.org/10.3115/1117794.1117802

[33] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 2010, pp. 156–165.

[34] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker, "Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 79–88. [Online]. Available: https://doi.org/10.1145/1370750.1370771

[35] C. Simonyi and M. Heller, "The hungarian revolution," *BYTE*, vol. 16, no. 8, p. 131–ff., Aug. 1991.

[36] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "An empirical investigation of how and why developers rename identifiers," in *International Workshop on Refactoring 2018*, 2018. [Online]. Available: http://doi.acm.org/10.1145/3242163.3242169

[37] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *2009 16th Working Conference on Reverse Engineering*, Oct 2009, pp. 31–35.

[38] D. Binkley, M. Hearn, and D. Lawrie, “Improving identifier informativeness using part of speech information,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR ’11. New York, NY, USA: ACM, 2011, pp. 203–206. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985471
[39] J. Wu and J. Clause, “A pattern-based approach to detect and improve non-descriptive test names,” *Journal of Systems and Software*, vol. 168, p. 110639, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121220301126
[40] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen, “Suggesting natural method names to check name consistencies,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1372–1384. [Online]. Available: https://doi.org/10.1145/3377811.3380926
[41] C. Caprile and P. Tonella, “Nomen est omen: analyzing the language of function identifiers,” in *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, Oct 1999, pp. 112–122.
[42] Caprile and Tonella, “Restructuring program identifier names,” in *Proceedings 2000 International Conference on Software Maintenance*, Oct 2000, pp. 97–107.
[43] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, “Using natural language program analysis to locate and understand action-oriented concerns,” in *Proceedings of the 6th International Conference on Aspect-oriented Software Development*, ser. AOSD ’07. New York, NY, USA: ACM, 2007, pp. 212–224. [Online]. Available: http://doi.acm.org/10.1145/1218563.1218587
[44] Z. P. Fry, D. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker, “Analysing source code: looking for useful verb-direct object pairs in all the right places,” *IET Software*, vol. 2, no. 1, pp. 27–36, February 2008.
[45] J. Singer and C. Kirkham, “Exploiting the correspondence between micro patterns and class names,” in *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, Sep. 2008, pp. 67–76.
[46] E. W. Høst, “Meaningful method names,” 2011.
[47] E. Host and B. Ostvold, “The programmer’s lexicon, volume i: The verbs,” 09 2007, pp. 193 – 202.
[48] E. W. Høst and B. M. Østvold, “The java programmer’s phrase book,” in *Software Language Engineering*, D. Gašević, R. Lämmel, and E. Van Wyk, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 322–341.
[49] E. W. ”Høst and B. M. Østvold, “Canonical method names for java,” in *Software Language Engineering*, B. ”Malloy, S. Staab, and M. van den Brand, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 226–245.
[50] M. L. Collard and J. I. Maletic, “srcml 1.0: Explore, analyze, and manipulate source code,” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 649–649.
[51] B. Zhang, E. Hill, and J. Clause, “Automatically generating test templates from test names (n),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 506–511.
[52] B. Zhang, E. Hill, and J. Clause, “Towards automatically generating descriptive names for unit tests,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 625–636. [Online]. Available: https://doi.org/10.1145/2970276.2970342
[53] C. D. Newman, M. J. Decker, R. S. AlSuhaibani, A. Peruma, D. Kaushik, and E. Hill, “An empirical study of abbreviations and expansions in software artifacts,” in *Proceedings of the 35th IEEE International Conference on Software Maintenance*. IEEE, 2019.
[54] S. L. Abebe and P. Tonella, “Natural language parsing of program element names for concept extraction,” in *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ser. ICPC ’10. USA: IEEE Computer Society, 2010, p. 156–159. [Online]. Available: https://doi.org/10.1109/ICPC.2010.29
[55] C. Albon, *Machine Learning with Python Cookbook: Practical Solutions from Preprocessing to Deep Learning*. O’Reilly Media, 2018. [Online]. Available: https://books.google.com/books?id=kIhQDwAAQBAJ
[56] P. Dangeti, *Statistics for Machine Learning*. Packt Publishing, 2017.
[57] N. Dragan, M. L. Collard, and J. I. Maletic, “Reverse engineering method stereotypes,” in *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ser. ICSM ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 24–34. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2006.54