5-10-2023

Efficiently Annotating Source Code Identifiers Using a Scalable Part of Speech Tagger

Gavin Burris
gb9951@rit.edu

Efficiently Annotating Source Code Identifiers Using a Scalable Part of Speech Tagger

by

Gavin Burris

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences
Rochester Institute of Technology
Rochester, NY

May 10, 2023

Committee Approval:

_____

Dr. J Scott Hawker Date
SE Graduate Program Director

_____

Dr. Mohamed Wiem Mkaouer Date
Assistant Professor

_____

Dr. Christian D. Newman Date
Assistant Professor

Abstract

This thesis details the process in which a part-of-speech tagger is developed in order to determine grammar patterns in source code identifiers. These grammar patterns are used to aid in the proper naming of identifiers in order to improve reader comprehension. This tagger is a continuation of an effort of a previous Ensemble Tagger [62], but with a focus on increasing the tagging rate while maintaining the accuracy, in order to make the tagger scalable. The Scalable Tagger will be trained on open source data sets, with a machine learning model and training features that are chosen to best suit the needs for accuracy and tagging rate. The results of the experiment will be contrasted with the results of the Ensemble Tagger to determine the Scalable Tagger's efficacy.

I would like to dedicate this thesis to my family for their encouragement and support throughout my academic career.

Contents

List of Tables

Chapter 1

Introduction

The ability of a software engineer to comprehend code that others have written is an integral part of the software engineering process [62]. In fact, it is estimated that developers spend more time reading and comprehending code than actually writing the code itself. A developer's increased comprehension, while reading other's code, can lead to a decrease in time spent reading code, a reduction in developer stress, and better overall code development [62]. One of the primary ways that code is understood is through identifiers, which compromise approximately 70 percent of characters found in code [62]. Improving the comprehension of these identifiers can therefore have the aforementioned desired positive effects.

One way to begin addressing the problem of source code comprehension is to understand how identifiers are connected with program behavior. This can be accomplished by using a part-of-speech tagger [62] . A part-of-speech tagger is a natural language processing technique in which words in a sentence

are annotated based on their linguistic role and how they interact with surrounding words. Part-of-speech tags can be used to partially understand how an identifier conveys program behavior [64]. The majority of part-of-speech taggers deal with the annotation and tagging of conventional text, however, the tagger addressing the problem of code comprehension needs to tag source code identifiers. This issue has been faced before as although part-of-speech tagging is the most popular method for natural language semantics, it has been widely inaccurate and therefore been deemed untrustworthy. [62]

Prior researchers have attempted to solve this issue by creating an ensemble part-of-speech tagger for tagging identifiers in source code [62]. The tagger was an ensemble of SWUM, POSSE, and Stanford taggers and used the decision tree and random forest machine learning techniques in order to achieve significant results [62] . The goal of this thesis is to create a part-of-speech tagger that maintains or improves on the state-of-the-art accuracy [62] while significantly improving the tagging rate.

The rest of this thesis is organized as follows: Chapter 2 details the motivations behind the creation of the scalable part-of-speech tagger, and lists the research questions that this study sets out to answer. Chapter 3 is the related work that is relevant to key concepts in the thesis. Chapter 4 details grammar pattern definitions and gives context to the part-of-speech annotation. Chapter 5 details the methodology behind the Scalable Tagger's creation. Chapter 6 presents the evaluation of the Scalable Tagger and gives answers to the research questions. Chapter 7 goes into potential threats to the validity of the thesis. Lastly, Chapter 8 summarizes everything in the conclusion.

Chapter 2

Research Objective

2.1 Motivation and Contribution

The research in this thesis, as stated in Chapter 1, builds upon the research of an ensemble part-of-speech tagger [62] . The Ensemble Tagger achieved an accuracy rate of 75% for tagging identifiers and an 86% accuracy rate for tagging at the word level, which is an increase of 17% from the closest individual tagger which it's composed of [62]. Despite its apparent success, the Ensemble Tagger has a major ineptitude: it can only identify one identifier per second. This flaw significantly limits its capability to scale to large source code bases as at this rate tagging one million identifiers could take longer than 12 days. This thesis has two main goals: (1) The creation of a scalable part-of-speech tagger that can tag identifiers in source code with at least the same accuracy as the state-of-the-art Ensemble Tagger, and (2) the Scalable Tagger must significantly improve its tagging rate relative to the state-of-the-art Ensemble

Tagger. In the long run, this will help support program comprehension research and tools. In particualr it will support further research into grammar patterns which are critical to understanding the naming structures that developers use in code [64].

2.2 Research Questions

- **RQ1:** ***What is the accuracy of this approach compared to the state of the art Ensemble Tagger?***

  This research question seeks to determine whether increasing the tagging rate is detrimental to the Scalable Tagger's overall accuracy in annotating source code identifiers. We base the success of the Scalable Tagger on whether it acheives an equal or greater accuracy metrics.

- **RQ2:** ***How scalable is this approach compared to the state of the art Ensemble Tagger?***

  This research question seeks to determine the increased tagging rate of the developed scalable part-of-speech tagger compared to the state-of-the-art, Ensemble Tagger. The increase in the tagging rate of the Scalable Tagger will determine its scalability, especially in contrast to the Ensemble Tagger's inability to scale, which was its major deficiency [62].

Chapter 3

Related Work

Many studies addressed challenged to software maintenance in general [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,38,39,40,41,42,43,44,45,46,48,49,51,52,53,54,55,56,57,58,59,60,61,65,66,68,69,70,71,72,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,94], and program comprehension in particular. Part-of-speech tagging is a commonly used method to analyze code in order to understand underlying relationships amongst identifiers. This has inspired numerous studies to be performed and papers to be written about this topic.

3.1 Part-of-Speech Taggers

Newman et al. [62] created an ensemble part-of-speech tagger for annotating identifiers in source code. The tagger comprised the SWUM, POSSE, and Stanford models, and achieved a 75% success rate at tagging identifiers and an

86% rate tagging at the word level [62] . This tagger is used as a reference in this paper as this study is meant to address its slow tagging rate.

Yitagesu et al. [95] developed a part-of-speech tagger for security vulnerability descriptions. That study set out to create a tagger that could tag SVDs much more accurately than a word-level part-of-speech tagger. Through the use of a neural network, the tagger was able to achieve a 93% accuracy for tagging SVDs.

Gupta et al. [47] developed a part-of-speech tagger that also specializes in tagging source code identifiers. This tagger uses POSSE (POS tagger for Software Engineering) to achieve an 11-20% increase in accuracy over other traditional taggers used for the same purpose.

Toutanova et al. [92] developed a part-of-speech tagger which optimizes for maximum entropy in the text by enriching tagging information sources. This tagger improved the accuracy of tagging individual words by using additional context provided by outside information.

Partachi et al. [67] researched Software Engineers' usage of mixed natural language and source code in their communications and developed a part-of-speech tagger, called POSIT to solve the problems of language identification and token tagging amongst this mixed text. The tagger could tag code tokens with their part-of-speech with an accuracy of 85.6%.

3.2 Part-of-speech-based Analysis of Identifiers

Newman et al. [63] describe the categorization of source code identifiers rather than their common place connotations. This study categorized source code

identifiers by their type, behavior, context, use, and meaning to provide a more descriptive definition to the identifier. This study saw a high correlation between the identifiers underlying meaning and the meaning of the words that comprised it.

Caprile et al. [37] research the intrinsic value in identifier names and how restructuring identifiers can improve their meaningfulness. The study was able to replace words in identifiers with little meaning with other words that retained the same connotation, but expressed a more significant meaning to the reader.

Peruma et al. [73] researched the importance of names of test methods and how proper use of grammar patterns can assist developers in better understanding test methods over time. The study confirmed the usefulness of grammar patterns in the understanding of test method names, and in understanding how they relate to code behavior.

Høst et al. [50] extracted rules for naming methods in order to point out "naming bugs" in code. An automatic suggestion of more suitable names was studied and presented to have a high correlation between the name given to a method and the method's underlying functionality.

Wu et al. [93] created a process for identifying non-descriptive test method names in JUnit tests to increase the comprehension of the purpose of the test method to the reader. This was done by providing information on how the test method name can be improved with a 95% accuracy rate in determining which method names needed to be improved.

Chapter 4

Grammar Pattern Definitions

By definition, the usage of a part-of-speech tagger involves the annotation of words with their parts of speech in grammar in a given context. This tagging of identifiers in code then results in the formation of a grammar pattern. A grammar pattern is the sequence of part-of-speech tags assigned to words within an identifier. For example, an identifier called "GetDogData" would start with the tagging of individual words (Get, Dog, and Data), and result in a grammar pattern of verb noun-adjunct noun. This grammar pattern can then be used to classify identifiers that share the same grammar patterns. This then can be used to correlate identifiers that have different meanings and connotations but share similarities in their program behavior.

Table 4.1: Part-of-Speech Categories to be Used by POS Tagger

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

4.1 Annotating Words in Identifiers

As identifiers are broken down to have their comprised vocabulary annotated, it is important to label and categorize these annotations. The categories used in the part-of-speech tagger are all included in Table I. These categories are the most common parts of speech that are recognizable from common English grammar parts of speech. However, there are two categories in Table I that are uncommon and need to be addressed, preambles and noun modifiers. In the context of code identifiers, and in this thesis, a preamble is an abbreviation that occurs at the beginning of an identifier to provide metadata, name-space an identifier, or highlight the identifier's type without changing the reader's understanding of the identifier. For example, the preamble p_ gives context that the identifier is being used as a pointer. Noun modifiers are defined in this thesis as an adjective or a noun that is being used as an adjective, called a noun-adjunct. An example of a noun-adjunct can be seen in the identifier "GetDogData" as the noun Dog is being used to describe the noun Data, making the word Dog a noun-adjunct in this identifier.

Chapter 5

Methodology

In order to evaluate the effectiveness of the Scalable Tagger created in this thesis against the Ensemble Tagger, both taggers will be trained on the same data-set. The process and means of sourcing the data-set used for training the Ensemble Tagger are not included in this thesis, but can be found in the paper on the Ensemble Tagger [62] if desired. The data-set is comprised of 1,335 manually annotated identifiers to label their part-of-speech, which were sourced from 20 different software systems. The sourcing of this data is broken down and shown in Table 5.1. The data-set was then broken down into a training set and test set in order to validate the results of the Scalable Tagger across different data sets. The test set was comprised of the identifiers from 5 of the software systems and the remaining 15 comprised the training set, from which more identifiers were extracted in order to maintain the 1,335 identifier basis.

In order to reduce over-fitting, 5-fold validation is used on the training

Table 5.1: 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	322
NM	1520	415
NPL	220	78
P	91	32
PRE	83	33
V	330	81
VM	12	3
Total	3449	977

data. The training data sourced from 15 software systems is split into 5 smaller train-test sets, or folds, which will be referred to as the "5-fold test set". The test set comprised of identifiers that are removed from the training process is referred to as the "unseen test set". The distribution of the annotations from the identifiers into the training set and unseen test set can be seen in Table 5.2.

Table 5.2: 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#

5.1 5-fold test set

Another form of validation of the Scalable Tagger is through 5-fold validation. During the training of the model, the training data comprised of 1,335 identifiers from 15 software systems are split into 5 smaller folds. This allows for a split amongst the folds for a percentage to be used as training data while the remaining is used as testing data to validate the trained model. After the data is split into the 5 folds, 70% is used as training data, and the remaining 30% is used as testing data. This is repeated 5 times until all of the sets have been used for testing. This is an application of k-fold cross-validation sets in which the value of k has been set to 5. This is appropriate with choices from researchers who often use k values of 5 or 10, from which 5 is used considering the size of our data-set and the distribution of the annotations. Each testing round is accompanied by the collection of metrics to evaluate the effectiveness of the model, which is described in the next section.

5.2 Quality Model Measurements

The quality of the Scalable Tagger is measured using common metrics for categorization problems. The metrics used are Accuracy, Precision, Recall, and F1 Score. Another metric used is the balanced accuracy of the 5-fold testing which is the average of each average accuracy for each annotation category (i.e., N, NM, CJ, etc.). This allows for each part-of-speech annotation to be equally represented, which helps with unbalanced data-sets by giving more weight to underrepresented annotations.

5.3 Choosing a machine learning approach

The Scalable Tagger uses a machine learning model in order to annotate identifiers with a part-of-speech. The model must be trained to follow a primary machine learning approach from which the following were evaluated for use: Random Forest, Decision Tree, XGBoost, Logistic Regression, Support Vector Classification, and K-Nearest Neighbors. From these Random Forest and Decision Tree were selected to be used as the primary approaches due to their out-performance of the other approaches in terms of the quality model metrics. The building of the models requires a training and testing set, which is from a splitting of the training data in which 70% is used as the training data and 30% is used as the testing data. Through further analysis of the performance of the Random Forest and Decision Tree approaches the Decision Tree approach was determined to be the better approach. Throughout the formation of the Scalable Tagger Random Forest would reach a stagnation to how well it performed, while Decision Tree continued to improve, eventually surpassing the performance of Random Forest. Random Forest is also a much slower approach than Decision Tree which is counterproductive to the goal of creating a scalable part-of-speech tagger.

5.4 Features

Machine learning algorithms must be trained on a set of features in order to derive patterns and predict outputs from given inputs. The basis for features used for the Scalable Tagger is based on relevant features used for the Ensemble

Tagger. The relevancy of features is determined by how they impact the quality model measurements detailed in Chapter 5.2. New features were developed in order to improve these quality measurements, thus improving the Scalable Tagger. Certain features did not end up being used either due to their negative impact to the Scalable Tagger. The testing of the features is detailed further in Chapter 5.4.4.

5.4.1 Existing Features

The Scalable Tagger uses the finding of the Ensemble Tagger as a basis of the features used for training. These existing features are those from the Ensemble Tagger continuing to be used due to their continued relevance. They are as follows:

1. 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.

2. Identifier size: The length, in words, of the identifier of which the word was originally part.

5.4.2 New Features

The training of the model focused on creating new features in order to better the Scalable Tagger's quality measurement metrics. Each new feature provides new insight into understanding key similarities and differences between part-of-speech categories. The new features are as follows:

1. Word Embeddings: Word vectorization is applied to each word to analyze the similarities and differences between them as they relate to other words. Word vectorization is the process of converting words in text into numerical arrays, in order for their use in training the model. Gensim, a popular natural language processing library, is used to provide a pre-trained model of a plethora of word vectors. Since this feature is an array of numbers, each index in the array is used as a separate feature that comprises the whole. The length of the array depends on the Gensim model used, which is "fasttext-wiki-news-subwords-300" making there be 300 Word Embedding features.

2. Last letter: The ASCII value of the last letter of the word. This provides better contrast between nouns and noun plurals as the difference usually occurs with noun plurals ending in the letter "S". For example the difference between "Dog" being a noun and "Dogs" being a noun plural, in which the last letter of the word being the key differentiator.

3. Digit: Determines whether the word is a digit or not. Is set to a value of 0 if the word is a digit or a 1 if the word is not.

4. Word Length: The length of the word. This can be used to differentiate

preambles, prepositions and digits which are typically shorter in length from nouns, noun plurals and verbs which are typically longer.

5.4.3 Removed and Failed Features

The Scalable Tagger detailed in this thesis used the features of the Ensemble Tagger as a basis for its starting features. This also means that certain features were removed due to their lack of relevance. The SWUM, POSSE, and Stanford annotation features were three part-of-speech taggers that were used to comprise the ensemble nature of the Ensemble Tagger. These features are the source of the slow tagging rate experienced by the Ensemble Tagger, which is the grounds for their removal. The Word, Data Type, Position, and Context features were also removed as they had a negligible effect on the Scalable Tagger's quality measurement metrics, which is further explained in Chapter 5.4.4. More information on these discarded features can be found in the Ensemble Tagger paper [62].

Other features were attempted to increase the quality measurement metrics besides those mentioned in Chapter 5.4.2, but they did not result in improvements. Further attempts at the use of word vectors were made through cosine correlation between word vectors. These cosine similarities were flawed in their over-fitting to given data, and when corrected presented no statistical improvements. Other features tried involved determining if a word was a commonly used conjunction or determiner. These part-of-speech categories are comprised of limited sets, however, this approach was unfruitful. These features were not useful in the end, but the process of finding statistically beneficial features

involves an arduous period of trial and error.

5.4.4 Feature Testing

Further testing was used to analyze the features to ensure which features were statistically significant to the quality measurement metrics detailed in Chapter 5.2. The analysis of the features involved two techniques: Drop-column feature importance and permutation importance. Drop-column feature importance is the training of the model on every possible subset of the available features, which can allow for the best possible subset to be determined. Permutation importance involves first training the model. Then each feature is randomly shuffled out, and the resulting importance is measured from the difference in the quality measurement metrics. This then gives a value to the importance of each feature, and whether it should be used. Each feature was run using these techniques to value their importance, from which the features stated in Chapters 5.4.1 and 5.4.2 were all validated. Testing using permutation importance was can be shown on these validated features in Tables 5.3, 5.4, and 5.5, which detail the importance of the features relating to F1 measure, balanced accuracy, and accuracy. The key exception is that of Word Length, which can be shown with no importance in all of these categories. Unlike the other features that have shown no importance, Word Length did have noticeable effects on the quality model measurements when removed. It should be noted, that these techniques do not represent non-linear relationships between features and the target outcome and do not always reflect the true importance of a feature, as can be seen in the instance of the Word Length feature.

Table 5.3: F1 weighted importances for best features

Feature Set	F1 Weighted Importances					Average
Normalized Position	0.36	0.37	0.36	0.35	0.36	0.36
Last Letter	0.09	0.09	0.09	0.09	0.09	0.09
Identifier Size	0.03	0.03	0.03	0.03	0.03	0.03
Digit	0.01	0.01	0.01	0.01	0.01	0.01
Word Length	0.00	0.00	0.00	0.00	0.00	0.00
Word Embeddings	0.54	0.54	0.55	0.54	0.54	0.54

Table 5.4: Balanced accuracy importances for best features

Feature Set	Balanced Accuracy Importances					Average
Normalized Position	0.34	0.35	0.31	0.31	0.30	0.32
Last Letter	0.12	0.13	0.12	0.14	0.14	0.13
Identifier Size	0.02	0.03	0.02	0.02	0.02	0.02
Digit	0.08	0.08	0.08	0.08	0.08	0.08
Word Length	0.00	0.00	0.00	0.00	0.00	0.00
Word Embeddings	1.33	1.38	1.38	1.29	1.34	1.34

Table 5.5: Accuracy importances for best features

Feature Set	Accuracy Importances					Average
Normalized Position	0.37	0.36	0.37	0.36	0.35	0.36
Last Letter	0.08	0.08	0.08	0.08	0.08	0.08
Identifier Size	0.03	0.03	0.03	0.03	0.03	0.03
Digit	0.01	0.01	0.01	0.01	0.01	0.01
Word Length	0.00	0.00	0.00	0.00	0.00	0.00
Word Embeddings	0.53	0.51	0.51	0.49	0.51	0.51

5.5 Tagger Scalability Timing

The essential flaw of the previous state-of-the-art part-of-speech tagger for source code identifiers, the Ensemble Tagger, was its inability to scale. The Scalable Tagger will be made scalable through the removal of the ensemble nature while attempting to maintain the accuracy and other quality measurement metrics set by the Ensemble Tagger.

The tagging rate for the Scalable Tagger will be determined by timing how long the Scalable Tagger takes to tag the unseen test set. This allows for the evaluation of the Scalable Tagger's tagging rate on a set of data that was not used in training. The tagging rate will therefore be calculated using the following equation:

$$Tagging\ Rate = \frac{\#\ of\ words\ tagged}{Total\ tagging\ time} \tag{5.1}$$

The unseen test set is comprised of 977 words from 384 identifiers, causing the tagging rate of the unseen test set to be:

$$Tagging\ Rate = \frac{977\ words}{Total\ tagging\ time} \tag{5.2}$$

The tagging rate has units of part-of-speech annotation, or tag, per second (tag/sec). One thing to mention is that the time it takes for the new features created for the Scalable Tagger was not applied to the time it takes for tagging. It is assumed that the features have been applied to the data set before the tagging is initiated.

Chapter 6

Analysis & Discussion

6.1 Tagger Accuracy

RQ1: *What is the accuracy of this approach compared to the state-of-the-art Ensemble Tagger?*

The evaluation of the Scalable Tagger's accuracy is determined in two ways. The first method by using the 5-fold cross validation, which uses a set of 1,335 manually-annotated identifiers. This allows for the evaluation of the model during its training. The second method is by running the trained model on the unseen test set of 384 identifiers. This provides an unbiased set of identifiers to provide assurance of the model's accuracy and that it isn't over-fitting on the training data.

Table 6.1: Benchmark Tagger quality measurement metrics for each annotation category

Annotation	Precision	Recall	F1-Score	Support
CJ	0.00	0.00	0.00	2
D	0.89	1.00	0.94	8
DT	0.00	0.00	0.00	4
N	0.75	0.85	0.80	333
NM	0.83	0.94	0.88	509
NPL	0.20	0.05	0.08	77
P	0.00	0.00	0.00	24
PRE	0.00	0.00	0.00	31
V	0.57	0.55	0.56	76
VM	0.00	0.00	0.00	1

Table 6.2: Benchmark Tagger's average quality measurement metrics

Metric	Average
Accuracy	0.77
Balanced Accuracy	0.34
Weighted F1 Score	0.72
Weighted Precision	0.69
Weighted Recall	0.77

6.1.1 5-fold Test Results

The first run of the Scalable Tagger, before any improvements were made, involved using a stripped down version of the original Ensemble Tagger [62], excluding the SWUM, POSSE, and Stanford annotations which were the cause of the Ensemble Tagger's detrimental slow tagging rate. This established a benchmark for improvement upon which the Scalable Tagger's improved results are compared. The Benchmark Tagger is shown in Tables 6.1 and 6.2. The evaluation of the Scalable Tagger is run using the Decision Tree machine learning approach as detailed in Chapter 5. The results from this evaluation can be seen in Tables 6.3 and 6.4. Both tables, 6.2 and 6.4, give the averaged results of the 5-fold evaluations which are the tagger's quality measurement metrics, which include the tagger's accuracy, balanced accuracy, weighted F1 score, weighted precision, and weighted recall. Tables 6.1 and 6.3 detail the precision, recall, and F1 score measured for each annotation category, which characterize the tagger's effectiveness across each category.

The Benchmark Tagger can be seen to have many inadequacies, as shown in Tables 6.1 and 6.2. The accuracy of the Benchmark Tagger, at 77%, is lower than that of the state-of-the-art Ensemble Tagger, at 86%. This needs to be improved upon in order to meet the set standards. The balanced accuracy, however, of the Benchmark is very low at 34%, as compared to the Ensemble's balanced accuracy of 55%, indicating that many categories are being mis-annotated by the Benchmark Tagger. This can be clearly seen as 5 of the 10 annotation categories are not tagged accurately at all, with 0% measurements for precision, recall, and F1 score. This contrast between ac-

Table 6.3: Scalable Tagger quality measurement metrics for each annotation category

Annotation	Precision	Recall	F1-Score	Support
CJ	1.00	1.00	1.00	2
D	1.00	0.88	0.93	8
DT	0.80	1.00	0.89	4
N	0.92	0.86	0.89	333
NM	0.87	0.94	0.90	509
NPL	0.89	0.82	0.85	77
P	0.80	0.83	0.82	24
PRE	0.89	0.52	0.65	31
V	0.68	0.63	0.65	76
VM	0.00	0.00	0.00	1

Table 6.4: Scalable Tagger's average quality measurement metrics

Metric	Average
Accuracy	0.87
Balanced Accuracy	0.75
Weighted F1 Score	0.87
Weighted Precision	0.87
Weighted Recall	0.87

curacy and balanced accuracy is due to the heavy support of nouns(N) and noun modifiers(NM), which are more accurately tagged than other annotation categories.

The Scalable Tagger improves on the inadequacies of the Benchmark Tagger in order to surpass the accuracy of the Ensemble Tagger, as can be seen in Tables 6.3 and 6.4. There are enhancements in every annotation category, except for that of VM, which can be attributed to its low support. The accuracy of the Scalable Tagger, at 87%, increased 10% from the Benchmark Tagger, as well as 1% from that of the Ensemble Tagger [62]. The biggest improvement is the Scalable Tagger's balance accuracy, at 75% is a 41% improvement over the Benchmark Tagger, and a 20% improvement over the Ensemble Tagger. This is due to the focus on addressing the tagging failures of the conjunction(CJ), determinant(DT), noun plural(NPL), preposition(P), and preamble(PRE) annotation categories which allows for more accurate tagging on a wider variety of identifiers that aren't comprised of the more common noun and noun modifiers.

6.1.2 Unseen Test Set Results

The unseen test set is an unbiased test set used to validate the trained model on data that hasn't been used to train it. The model was run on the unseen test set, resulting in the accuracy evaluations shown in Table 6.5. This result validates the finding from the 5-fold test validation as the accuracy of the Scalable Tagger remains congruent in both testing methods across most of the annotations, except for conjunctions which has a low sample size. The

Annotation	Number Correct	Number Incorrect	Accuracy
CJ	0	1	0.0%
D	7	0	100.0%
DT	4	1	80.0%
N	280	42	87.0%
NM	400	15	96.4%
NPL	69	9	88.5%
P	28	4	87.5%
PRE	22	11	66.7%
V	59	22	72.8%
VM	1	2	33.3%
Total	870	107	89.0%

testing of the unseen test set resulted in a greater accuracy of 89%, which is 2% greater than that of the 5-fold validation. This demonstrates a very low chance of over-fitting in the model, and that the Scalable Tagger's improved accuracy holds true.

6.2 Tagger Scalability

RQ2: *How scalable is this approach compared to the state-of-the-art Ensemble Tagger?*

The lack of scalability of the Ensemble Tagger is a major limitation. This limits its ability to perform tagging on large scale code bases, as users would

like to use a tool that is timely and effective. The lack of scalability is due to its slow tagging rate of approximately 1 part-of-speech annotation per second. At this rate, the tagging of one million words from a large scale repository with a very large amount of source code identifiers would take 12 days.The Scalable Tagger attempts to have its tagging rate be swift in order to correct for this limitation.

The scalability, through means of the tagging rate, has been evaluated by measuring the amount of time the Scalable Tagger takes to tag all the identifiers in the unseen test set. The time that the Scalable Tagger took to annotate all 977 words from 384 identifiers is 3.984 seconds. The Scalable Tagger's tagging rate can then be calculated by using Equation 5.2 using the newly measured time:

$$Tagging\ Rate = \frac{977\ words}{3.984\ seconds} \tag{6.1}$$

This computes to a tagging rate of approximately 245 tags per second. This rate is **245 times** faster than that of the Ensemble Tagger, which clearly improves upon its slow tagging rate. This allows for a much more Scalable Tagger, as in the aforementioned example of a large repository with one million words to be tagged, the Scalable Tagger would only take approximately 1.2 hours to tag every word. This is a great improvement from the 12 days that the Ensemble Tagger would take to achieve the same task.

It is also important to mention that this feat would not be as important without the achievements evaluated in Chapter 6.1. The ability for the Scalable Tagger to be much faster in its tagging rate, thus being scalable would not be as useful if the Scalable Tagger's accuracy, balanced accuracy, precision, recall,

and F1-score suffered as a consequence.

Chapter 7

Threats to Validity

This thesis sources its datasets from those used by the Ensemble Tagger. Therefore, one concern is that the collection of 1,335 identifiers could have been improperly annotated by the authors of the ensemble tagging paper [62], leading to imperfections in the data. This was validated by the cross-validation amongst annotators for all grammar patterns, and by having the data sets publicly available allowing for corrections from outside sources.

Another concern is the confidence we have that the features selected in the training of the model have a true cause-and-effect relationship with the evaluation of the Scalable Tagger's accuracy and other quality measurement metrics and that the positive evaluation of these measurements are not caused by other external factors. The use of drop-column and permutation importances validated that the features were directly affecting these measurements, as they were tested through 5-fold cross-validation. Outside external factors were mitigated by keeping the same training and testing sets for the Scalable

Tagger as the Ensemble Tagger.

Over-fitting is another threat that must be acknowledged. In order to ensure that the Scalable Tagger is not only valid on the data used to train it, the Scalable Tagger went through the testing method of 5-fold cross-validation, which used train-test folds to ensure that each sample of training data was also used to test the Scalable Tagger. This was compounded by the use of an external testing set, unseen to the Scalable Tagger during training. The validation that both of these testing methods proved to have positive evaluations on the Scalable Tagger's quality measurement metrics indicates that over-fitting has been properly mitigated.

Many machine learning approaches, including Random Forest, Decision Tree, XGBoost, Logistic Regression, Support Vector Classification, and K-Nearest Neighbors, were considered for this model along with numerous features to train them. This resulted in the usage of a model implementing the Decision Tree approach and the features discussed in Chapter 5. These were then evaluated using the metrics of accuracy, balanced accuracy, recall, precision, and F1-score. This resulted in a new state-of-the-art tagger according to these metrics that is scalable, however, this should not be taken as the final result in source code identifier tagging. It would be a naive judgment that the Scalable Tagger could not be potentially improved upon since the Scalable Tagger itself is an improvement on the previous state-of-the-art tagger. To mitigate this, the data-sets and code for the Scalable Tagger have been made publicly available on GitHub.

Chapter 8

Conclusion

This thesis has detailed the creation and evaluation of a new scalable part-of-speech tagger. The Scalable Tagger was modeled using the Decision Tree machine learning algorithm and trained using various features to target the better tagging of specific annotation categories while allowing for a rapid tagging rate. This resulted in the Scalable Tagger having an 87% accuracy, which is a 1% increase over its predecessor, a balanced accuracy of 75%, which is a 20% improvement over its predecessor, and a tagging rate of 245 annotations per second, which is 245 times faster than its predecessor.

This part-of-speech tagger has, therefore, become the new state-of-the-art tagger for the annotation of source code identifiers. It has addressed the major inadequacy of the Ensemble Tagger, its predecessor, of lacking scalability due to its slow tagging rate. The Scalable Tagger detailed in this thesis is deemed successful as it has addressed this thesis' main goal of being scalable while maintaining the accuracy set by the Ensemble Tagger.

Future work on the subject of the tagging of source code identifiers may arise around handling certain limitations of this project. Future taggers can implement other machine learning algorithms that were not tested in this thesis in order to increase the scalability and/or the accuracy of the tagger. The usage of a larger data set to train the model, whose annotations are validated by more analysts, would aid in allowing the tagger to recognize patterns that may not be present in the data set used for this thesis.

Bibliography

[1] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ocinneide, Ali Ouni, and Yuanfang Cai. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961, 2018.

[2] Wajdi Aljedaani, Mona Aljedaani, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Stephanie Ludi, and Yousef Bani Khalaf. I cannot see you—the perspectives of deaf students to online learning during covid-19 pandemic: Saudi arabia case study. *Education Sciences*, 11(11):712, 2021.

[3] Wajdi Aljedaani, Mohammed Alkahtani, Stephanie Ludi, Mohamed Wiem Mkaouer, Marcelo M Eler, Marouane Kessentini, and Ali Ouni. The state of accessibility in blackboard: Survey and user reviews case study. In *20th International Web for All Conference*, pages 84–95, 2023.

[4] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Anthony Peruma, and Stephanie Ludi. Do the test smells assertion roulette and eager test im-

pact students' troubleshooting and debugging capabilities? *arXiv preprint arXiv:2303.04234*, 2023.

[5] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[6] Bader Alkhazi, Andrew DiStasi, Wajdi Aljedaani, Hussein Alrubaye, Xin Ye, and Mohamed Wiem Mkaouer. Learning to rank developers for bug report assignment. *Applied Soft Computing*, 95:106667, 2020.

[7] Nuri Almarimi, Ali Ouni, Salah Bouktif, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Mohamed Aymen Saied. Web service api recommendation for automated mashup creation using multi-objective evolutionary search. *Applied Soft Computing*, 85:105830, 2019.

[8] Nuri Almarimi, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. csdetector: an open source tool for community smells detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1560–1564, 2021.

[9] Nuri Almarimi, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. On the detection of community smells using genetic programming-based ensemble classifier chain. In *Proceedings of*

the 15th International Conference on Global Software Engineering, pages 43–54, 2020.

[10] Nuri Almarimi, Ali Ouni, and Mohamed Wiem Mkaouer. Learning to detect community smells in open source software projects. *Knowledge-Based Systems*, 204:106201, 2020.

[11] Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, and Ali Ouni. Recommending relevant classes for bug reports using multi-objective search. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 286–295. IEEE, 2016.

[12] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 51–58. IEEE, 2019.

[13] Eman Abdullah AlOmar, Salma Abdullah AlOmar, and Mohamed Wiem Mkaouer. On the use of static analysis to engage students with software quality improvement: An experience with pmd. *arXiv preprint arXiv:2302.05554*, 2023.

[14] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering:*

Software Engineering in Practice (ICSE-SEIP), pages 348–357. IEEE, 2021.

[15] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[16] Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian Newman, and Mohamed Wiem Mkaouer. Satdbailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming*, 213:102693, 2022.

[17] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. On the documentation of refactoring types. *Automated Software Engineering*, 29(1):1–40, 2022.

[18] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology*, 140:106675, 2021.

[19] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Mining and managing big data refactoring for design improvement: Are we

there yet? *Knowledge Management in the Development of Data-Intensive Systems*, pages 127–140, 2021.

[20] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, 171:110821, 2021.

[21] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[22] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane 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*, 167:114176, 2021.

[23] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, page e2395, 2021.

[24] Eman Abdullah AlOmar, Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. On the relationship between developer experience and refactoring: An exploratory study and preliminary

results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 342–349, 2020.

[25] Eman Abdullah AlOmar, Philip T Rodriguez, Jordan Bowman, Tianjia Wang, Benjamin Adepoju, Kevin Lopez, Christian Newman, Ali Ouni, and Mohamed Wiem Mkaouer. How do developers refactor code to improve code reusability? In *International Conference on Software and Software Reuse*, pages 261–276. Springer, 2020.

[26] Eman Abdullah Alomar, Tianjia Wang, Vaibhavi Raut, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: an empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2022.

[27] Eman Abdullah AlOmar, Tianjia Wang, Raut Vaibhavi, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: An empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2021.

[28] Hussein Alrubaye, Deema Alshoaibi, Eman Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. How does library migration impact software quality and comprehension? an empirical study. In *International Conference on Software and Software Reuse*, pages 245–260. Springer, 2020.

[29] Hussein Alrubaye, Stephanie Ludi, and Mohamed Wiem Mkaouer. Comparison of block-based and hybrid-based environments in transfer-

ring programming skills to text-based environments. *arXiv preprint arXiv:1906.03060*, 2019.

[30] Hussein Alrubaye and Mohamed Wiem Mkaouer. Automating the detection of third-party java library migration at the function level. In *CASCON*, pages 60–71, 2018.

[31] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. Learning to recommend third-party library migration opportunities at the api level. *Applied Soft Computing*, 90:106140, 2020.

[32] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. Migrationminer: An automated detection tool of third-party java library migration at the method level. In *2019 IEEE international conference on software maintenance and evolution (ICSME)*, pages 414–417. IEEE, 2019.

[33] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. On the use of information retrieval to automate the detection of third-party java library migration at the method level. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 347–357. IEEE, 2019.

[34] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Anthony Peruma. Variability in library evolution: An exploratory study on open-source java libraries. In *Software Engineering for Variability Intensive Systems*, pages 295–320. Auerbach Publications, 2019.

[35] Deema Alshoaibi, Kevin Hannigan, Hiten Gupta, and Mohamed Wiem Mkaouer. Price: Detection of performance regression introducing code changes using static and dynamic metrics. In *International Symposium on Search Based Software Engineering*, pages 75–88. Springer, 2019.

[36] Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Increasing the trust in refactoring through visualization. In *2020 IEEE/ACM 4th International Workshop on Refactoring (IWoR)*, 2020.

[37] Caprile and Tonella. Restructuring program identifier names. In *Proceedings 2000 International Conference on Software Maintenance*, pages 97–107, 2000.

[38] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamon Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. Anti-patterns in modern code review: Symptoms and prevalence. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 531–535. IEEE, 2021.

[39] Moataz Chouchen, Ali Ouni, and Mohamed Wiem Mkaouer. Androlib: Third-party software library recommendation for android applications. In *International Conference on Software and Software Reuse*, pages 208–225. Springer, 2020.

[40] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Whoreview: A multi-objective search-based ap-

proach for code reviewers recommendation in modern code review. *Applied Soft Computing*, 100:106908, 2021.

[41] Motaz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Recommending peer reviewers in modern code review: a multi-objective search-based approach. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 307–308, 2020.

[42] Marwa Daaji, Ali Ouni, Mohamed Mohsen Gammoudi, Salah Bouktif, and Mohamed Wiem Mkaouer. Multi-criteria web services selection: Balancing the quality of design and quality of service. *ACM Transactions on Internet Technology (TOIT)*, 22(1):1–31, 2021.

[43] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. On the classification of bug reports to improve bug localization. *Soft Computing*, 25(11):7307–7323, 2021.

[44] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology*, 135:106566, 2021.

[45] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On the classification of software change messages using multi-label active learning. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1760–1767, 2019.

[46] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1303–1313, 2021.

[47] Samir Gupta, Sana Malik, Lori 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)*, pages 3–12, 2013.

[48] Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. An empirical study on the impact of refactoring on quality metrics in android applications. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 28–39. IEEE, 2021.

[49] Oumayma Hamdi, Ali Ouni, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. A longitudinal study of the impact of refactoring in android applications. *Information and Software Technology*, 140:106699, 2021.

[50] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 294–317, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[51] Licelot Marmolejos, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On the use of textual feature extraction techniques to support the automated detection of refactoring documentation. *Innovations in Systems and Software Engineering*, pages 1–17, 2021.

[52] Montassar Ben Messaoud, Ilyes Jenhani, Nermine Ben Jemaa, and Mohamed Wiem Mkaouer. A multi-label active learning approach for mobile app user review classification. In *International Conference on Knowledge Science, Engineering and Management*, pages 805–816. Springer, 2019.

[53] Mohamed W Mkaouer, Marouane Kessentini, Slim Bechikh, and Daniel R Tauritz. Preference-based multi-objective software modelling. In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 61–66. IEEE, 2013.

[54] Mohamed Wiem Mkaouer. Interactive code smells detection: An initial investigation. In *International Symposium on Search Based Software Engineering*, pages 281–287. Springer, Cham, 2016.

[55] Mohamed Wiem Mkaouer and Marouane Kessentini. Model transformation using multiobjective optimization. In *Advances in Computers*, volume 92, pages 161–202. Elsevier, 2014.

[56] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software

refactoring using nsga-iii. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1263–1270, 2014.

[57] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 331–336, 2014.

[58] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, and Mel Ó Cinnéide. A robust multi-objective approach for software refactoring under uncertainty. In *International Symposium on Search Based Software Engineering*, pages 168–183. Springer, Cham, 2014.

[59] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545, 2016.

[60] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Ó Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, 22(2):894–927, 2017.

[61] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software

remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[62] C. D. Newman, M. J. Decker, R. S. Alsuhaibani, A. Peruma, M. Mkaouer, S. Mohapatra, T. Vishnoi, M. Zampieri, T. J. Sheldon, and E. Hill. An ensemble approach for annotating source code identifiers with part-of-speech tags. *IEEE Transactions on Software Engineering*, 48(09):3506–3522, sep 2022.

[63] Christian D. Newman, Reem S. AlSuhaibani, Michael L. Collard, and Jonathan I. Maletic. Lexical categories for source code identifiers. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 228–239, 2017.

[64] Christian D. Newman, Reem S. AlSuhaibani, Michael J. Decker, Anthony Peruma, Dishant Kaushik, Mohamed Wiem Mkaouer, and Emily Hill. On the generation, structure, and semantics of grammar patterns in source code identifiers. *Journal of Systems and Software*, 170:110740, 2020.

[65] Christian D Newman, Michael J Decker, Reem Alsuhaibani, Anthony Peruma, Mohamed Mkaouer, Satyajit Mohapatra, Tejal Vishoi, Marcos Zampieri, Timothy Sheldon, and Emily Hill. An ensemble approach for annotating source code identifiers with part-of-speech tags. *IEEE Transactions on Software Engineering*, 2021.

[66] Christian D Newman, Mohamed Wiem Mkaouer, Michael L Collard, and Jonathan I Maletic. A study on developer perception of transformation

languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 34–41, 2018.

[67] Profir-Petru Pârtachi, Santanu Kumar Dash, Christoph Treude, and Earl T. Barr. Posit: Simultaneously tagging natural and programming languages. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1348–1358, New York, NY, USA, 2020. Association for Computing Machinery.

[68] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA, 2019. IBM Corp.

[69] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pages 193–202, 2019.

[70] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Sympo-*

sium on the Foundations of Software Engineering, ESEC/FSE 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[71] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsdetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2020.

[72] Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Christian D Newman. Using grammar patterns to interpret test method name evolution. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 335–346. IEEE, 2021.

[73] Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Christian D. Newman. Using grammar patterns to interpret test method name evolution. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 335–346, 2021.

[74] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 26–33. ACM, 2018.

[75] Anthony Peruma, Mohamed Wiem Mkaouer, Michael John Decker, and Christian Donald Newman. Contextualizing rename decisions using refactorings and commit messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 74–85. IEEE, 2019.

[76] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 350–357, New York, NY, USA, 2020. Association for Computing Machinery.

[77] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27(1):1–43, 2022.

[78] Nasir Safdari, Hussein Alrubaye, Wajdi Aljedaani, Bladimir Baez Baez, Andrew DiStasi, and Mohamed Wiem Mkaouer. Learning to rank faulty source files for dependent bug reports. In *Big Data: Learning, Analytics, and Applications*, volume 10989, page 109890B. International Society for Optics and Photonics, 2019.

[79] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. On the prediction of continuous integration build failures using

search-based software engineering. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 313–314, 2020.

[80] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128:106392, 2020.

[81] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Bf-detector: an automated tool for ci build failure detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1530–1534, 2021.

[82] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Web service api anti-patterns detection as a multi-label learning problem. In *International Conference on Web Services*, pages 114–132. Springer, 2020.

[83] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1):1–61, 2022.

[84] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Fabio Palomba. On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Information and Software Technology*, 138:106618, 2021.

[85] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. Towards automated microservices extraction using muti-objective evolution-

ary search. In *International Conference on Service-Oriented Computing*, pages 58–63. Springer, Cham, 2019.

[86] Islem Saidani, Ali Ouni, and Wiem Mkaouer. Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering*, 2021.

[87] Ian Shoenberger, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the use of smelly examples to detect code smells in javascript. In *European Conference on the Applications of Evolutionary Computation*, pages 20–34. Springer, Cham, 2017.

[88] Makram Soui, Mabrouka Chouchane, Narjes Bessghaier, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the impact of aesthetic defects on the maintainability of mobile graphical user interfaces: An empirical study. *Information Systems Frontiers*, pages 1–18, 2021.

[89] Makram Soui, Mabrouka Chouchane, Ines Gasmi, and Mohamed Wiem Mkaouer. Plain: Plugin for predicting the usability of mobile user interface. In *VISIGRAPP (1: GRAPP)*, pages 127–136, 2017.

[90] Makram Soui, Mabrouka Chouchane, Mohamed Wiem Mkaouer, Marouane Kessentini, and Khaled Ghedira. Assessing the quality of mobile graphical user interfaces using multi-objective optimization. *Soft Computing*, 24(10):7685–7714, 2020.

[91] Taryn Takebayashi, Anthony Peruma, Mohamed Wiem Mkaouer, and Christian D Newman. An exploratory study on the usage and readabil-

ity of messages within assertion methods of test cases. *arXiv preprint arXiv:2303.00169*, 2023.

[92] Kristina Toutanova and Christopher 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*, EMNLP '00, page 63–70, USA, 2000. Association for Computational Linguistics.

[93] Jianwei Wu and James Clause. A pattern-based approach to detect and improve non-descriptive test names. *Journal of Systems and Software*, 168:110639, 2020.

[94] Xin Ye, Yongjie Zheng, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. Recommending pull request reviewers based on code changes. *Soft Computing*, 25(7):5619–5632, 2021.

[95] Sofonias Yitagesu, Xiaowang Zhang, Zhiyong Feng, Xiaohong Li, and Zhenchang Xing. Automatic part-of-speech tagging for security vulnerability descriptions. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 29–40, 2021.