Identifying Utility Functions using Random Forests

Tamara Mendes*, Marco Tulio Valente*, Andre Hora* and Alexander Serebrenik†
*UFMG, Belo Horizonte, Brazil
†Eindhoven University of Technology, Eindhoven, The Netherlands
tamara.mendes@dcc.ufmg.br, mtov@dcc.ufmg.br, hora@dcc.ufmg.br, a.serebrenik@tue.nl

***Abstract*—Utility functions are general purpose functions, which are useful in many parts of a system. To facilitate reuse, they are usually implemented in specific libraries. However, developers frequently miss opportunities to implement general-purpose functions in utility libraries, which decreases the chances of reuse. In this paper, we describe our ongoing investigation on using Random Forest classifiers to automatically identify utility functions. Using a list of static source code metrics we train a classifier to identify such functions, both in Java (using 84 projects from the Qualitas Corpus) and in JavaScript (using 22 popular projects from GitHub). We achieve the following median results for Java: 0.90 (AUC), 0.83 (precision), 0.88 (recall), and 0.84 (F-measure). For JavaScript, the median results are 0.80 (AUC), 0.75 (precision), 0.89 (recall), and 0.76 (F-measure).**

I. Introduction

Utility functions are general purpose functions, which are useful in many parts of a system. As examples, we have functions for date and time manipulation, string manipulation, and accessing data structures. They are usually implemented in specific libraries, to facilitate reuse. However, developers often miss opportunities to implement general-purpose functions in such libraries. For example, when describing an experience of remodularizing a large banking system (with more than 100 installations, across 50 countries), Sarkar et al. report that often "lower-granularity functions, such as date validation, existed in the same library—and sometimes in the same source-code file—as complex domain functions, such as interest calculation" [1].

Figure 1 shows examples of utility functions, extracted from three popular JavaScript systems: ACE (a source code editor), BRACKETS (another source code editor), and BOWER (a package manager). These functions implement low-level and general-purpose tasks, like simple tests (`isRegExp`, from ACE, and `isWhitespace`, from BRACKETS) and array processing tasks (`toArray`, from BOWER). However, they are implemented in domain-specific files, instead of utility libraries. We also found that utility libraries are common in many systems. For example, files with `util` in their path name are found in 95 out of 106 Java systems, which are part of the Qualitas Corpus [2]. For JavaScript, `util` files are less common than in Java, but they are still frequent. For example, we found such files in 42 out of 100 popular JavaScript systems.

In this paper, we propose the use of machine learning classifiers to identify utility functions, including the ones not implemented in utility libraries. Identifying and alerting developers about these functions is important because a *Move Method* refactoring is recommended in such cases, to move the functions to their correct module [1], [3]–[5]. Implementing utility functions in utility libraries is important to increase their visibility and chances of reuse and by consequence to reduce the chances of reimplementation.

```
/* ace/lib/ace/incremental_search.js */
function isRegExp(obj) {
  return obj instanceof RegExp;
}
```

```
/* brackets/src/language/HTMLTokenizer.js */
function isWhitespace(c) {
  return c === " " || c === "\t" || c === "\r" || ...";
}
```

```
/* bower/lib/commands/init.js */
function toArray(value, splitter) {
  var arr = value.split(splitter || /[\s,]/);
  arr = arr.map(function (item) {
      return item.trim();
  });
  arr = arr.filter(function (item) {
      return !!item;
  });
  return arr.length ? arr : null;
}
```

Fig. 1. Utility functions not implemented in util libraries

Specifically, we report our ongoing effort on building utility functions classifiers, using the Random Forest machine learning algorithm [6]. To build these classifiers, we assume two facts: (a) most functions implemented in utility libraries are indeed utility functions; (b) however, there are some utility functions which are not implemented in utility libraries. We first describe an exploratory study conducted to check these assumptions (Section II). We then describe the algorithms, datasets, and evaluation strategies used in the work (Section III). Our preliminary results (Section IV) show that Random Forest classifiers can identify utility functions with very high accuracy (median precision of 83% and 75%, for Java and JavaScript systems; and median recall of 88% and 89%, again for Java and JavaScript). We conclude by presenting related work (Section V), by discussing practical applications of our results (Section VI) and by proposing a future research agenda (Section VII).

II. Preliminary Exploratory Study

We conducted a first study to check two central assumptions in our research:

- *Assumption #1 (Research Problem):* There are utility functions that are not implemented in *util* libraries.
- *Assumption #2 (Availability of Training Data):* Most functions in *util* libraries are indeed utility functions.

TABLE I
SYSTEMS USED IN THE EXPLORATORY STUDY

System	Description	Language	# Devs.
A	Rocket Location Visualization	JavaScript	7
B	Rocket Location Data Analysis	Java	17

TABLE II
EXPLORATORY STUDY RESULTS

System	LOC	NOF	NOUF	FP	FN
System A	12,212	1,334	199	11	16
System B	60,184	6,905	388	17	14

The first assumption is central to convince that our research problem indeed *happens* in practice, *i.e.*, that developers sometimes implement util functions in modules that are not util libraries. By contrast, the second assumption is central to show that we have chances to *solve* this research problem, by training a classifier on the util functions implemented in util libraries and then testing its ability to identify similar functions that are implemented in other modules.

To check these assumptions the first author of this paper manually inspected all functions from two proprietary aerospace systems implemented in JavaScript and Java, as described in Table I.[1] This author is an expert developer on these systems (three years of experience). Therefore, she has the required expertise to classify their functions, as util and non-util functions. Moreover, she has five years of professional experience in Java and three years in JavaScript.

Table II reports the results of this manual validation. For each system, the table shows (a) the number of lines of code (LOC); (b) the total number of functions (NOF);[2] (c) the total number of functions implemented in util libraries (NOUF); (d) the number of false positives, *i.e.*, functions implemented in util libraries that are not utility functions; (e) the number of false negatives, *i.e.*, utility functions not implemented in util libraries. We can see that both assumptions hold in these systems. First, System A and System B have 16 (8% of the total number of utility functions) and 14 (3%) utility functions that are not implemented in util libraries, respectively (Assumption #1). Second, most functions in util libraries are indeed utility functions, *i.e.*, 188 functions ($199 - 11$; 94%) and 371 functions (95%), respectively (Assumption #2).

Limitations and Threats to Validity: This study is a first step towards convincing that the problem we tackle happens in practice and that at the same time we have reliable data to train a classifier of utility functions. However, we plan to extend this study, by considering more systems. We also acknowledge that our manual classification of utility functions is subjected to failures, although it was performed by an expert developer on both systems considered in this first exploratory study.

[1]The real systems' names are omitted due to a non-disclosure agreement.

[2]To unify the terms for Java and JavaScript, we use the term function to denote Java's method, throughout this paper.

III. STUDY DESIGN

A. Machine Learning Algorithm

We use a Random Forest classifier [6] because it is known to have several advantages, such as being robust to noise and outliers [7], [8]. In addition, Random Forest is widely used in software engineering research, with promising results [8]–[13]. In future work, we plain to test other algorithms.

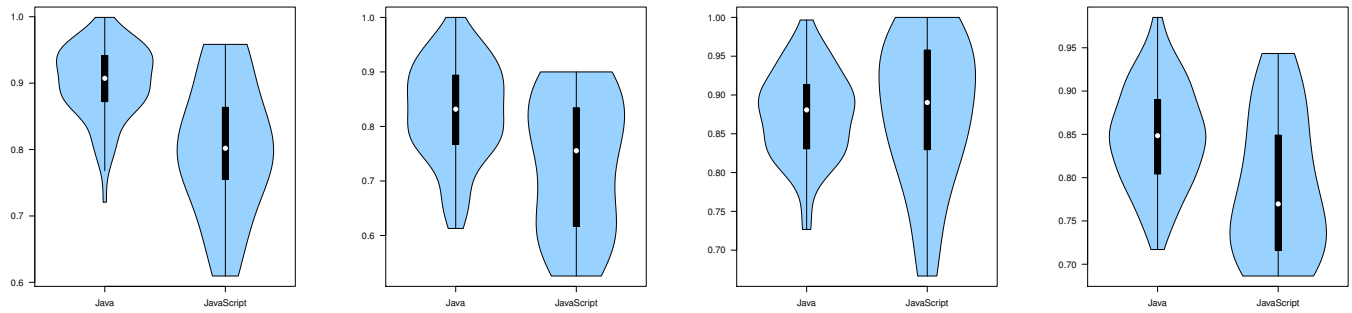
B. Predictors

For Java, as input to Random Forest we use 23 predictors calculated for methods (the type of the predictor's value is showed between parenthesis): cyclomatic complexity (int), LOC (int), number of formal parameters (int), returns a non-void type (boolean), is static (boolean), method's modifier (private/protected/public/default), is final (boolean), is abstract (boolean), number of references to this (int), is a get/set method (boolean), is implemented in a subclass (boolean), overwrites a method of the superclass (boolean), throws an exception (boolean), number of references to static variables (int), number of references to final variables (int), number of references to field variables (int), number of calls to project's methods (int), number of calls to Java API or other library methods (int), number of references to project's classes (int), number of references to Java API or library classes (int), is implemented in an inner class (boolean), number of outcoming calls (int), number of other classes calling the method (int). Basically, these predictors were selected to (a) include basic metrics (such as LOC and cyclomatic complexity); (b) data directly provided by Java ASTs (such as, is static and number of parameters); (c) simple coupling metrics to the Java API or other external libraries, since utility functions usually do not depend on a project's type but depend on other libraries types.

For JavaScript, we essentially select predictors following the same criteria used for Java. We use the following 20 predictors: cyclomatic complexity (int), LOC (int), number of parameters (int), includes a return statement (boolean), uses prototype (boolean), number of DOM uses (int), uses *arguments* variable (boolean), is nested in anonymous function (boolean), has an empty body (boolean), only returns a boolean constant (boolean), number of references to this (int), number of `getById` calls using a constant string as argument (int), number of references to global variables (int), number of references to native variables (*e.g.*, `Math`) (int), number of function calls (int), number of nested function definitions (int), number of qualified function calls (*e.g.*, `o.f()`) (int), number of local function calls (int), number of native function calls (*e.g.*, `Math.sin()`) (int), number of function calls that are not covered by the previous three predictors values (int).

C. Dataset

For Java, we initially consider 106 systems from the Qualitas.class Corpus [14]. We assume that utility libraries are implemented in files that contain the word `util` in their path. In 95 systems, we found utility libraries, according to this criterion. We then select 84 systems for our study, which are the ones that contain at least 25 methods in the detected utility



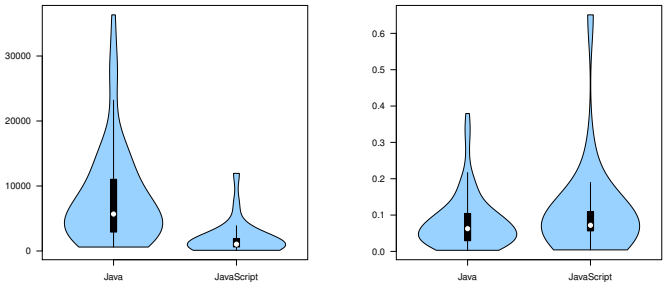
(a) AUC (b) Precision (c) Recall (d) F-Measure

Fig. 2. Accuracy measures

libraries. This dataset includes well-known Java systems, such as JHOTDRAW, SPRING FRAMEWORK, and HIBERNATE.

For JavaScript, we initially considered the top-100 GitHub projects, ordered by number of stars. Using the same criterion to detect utility libraries in Java, we found such libraries in 42 systems. We then select 22 systems with more than 25 utility functions. We used GitHub's Linguist tool to remove third-partly libraries, which are often included in JavaScript repositories [15].[3] The final dataset includes systems such as BRACKETS, PDF.JS, and LEAFLET.

Figure 3 shows violin plots with the number of functions and the ratio of util functions in the datasets. The systems in Java have more functions than in JavaScript (5,687 and 1,049 functions, median values). However, in relative terms, JavaScript systems have more utility functions (6.25% and 7.19%, median values, respectively).



(a) Number of Functions (b) % of Util Functions

Fig. 3. (a) Number of functions, (b) percentage of util functions

D. Training and Steps

For each system s with u_s utility functions, we compute the defined predictors for: set (a) with u_s utility functions; and set (b) with u_s randomly selected non-utility functions. It is important to highlight that we usually have noisy data in both sets. On one hand, in set (a) we might have functions that are not utility functions; on the other hand, in set (b) we might have have functions that are indeed utility functions. However, as evidenced in the exploratory study (Section II), both kinds of functions do not dominate the respective sets. Moreover, random forests classifiers are known to tolerate some level of noisy data [6].

Using this data, we then rely on 10-fold cross validation to train and test a Random Forest classifier for each system separately.[4] We build scripts that randomly divide a given system's data in ten folds and perform ten training/testing cycles. In each cycle, nine folds are used for training and the remaining one for testing. After ten cycles, the accuracy results are aggregated under four measures: AUC (area under the curve), precision, recall, and F-measure. AUC is a commonly used measure for binary classification problems and it refers to the area under the Receiver Operating Characteristic (ROC) curve. AUC $\geq$ 0.70 is considered reasonably good [8], [16]. In fact, in the Software Engineering domain, many accepted classifiers have AUC values between 0.70–0.80 [8], [9], [12].

IV. RESULTS

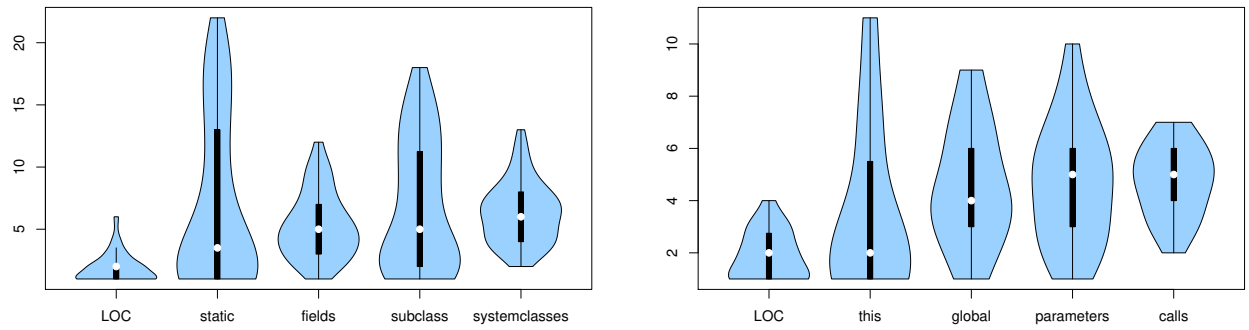
A. Accuracy

Figure 2 shows the values of the accuracy measures, for each classifier (one classifier per system and programming language). The results are summarized as follows:

- *AUC:* For Java, AUC results for 1st, 2nd, and 3rd quartiles are 0.87, 0.90, and 0.94, respectively. APACHE COMMONS COLLECTIONS is the system with the highest AUC measure (0.99) and FREEMIND is the system with the lowest measure (0.72). For JavaScript, AUC results for 1st, 2nd, and 3rd quartiles are 0.75, 0.80, and 0.86, respectively. TYPEAHEAD.JS is the system with the highest AUC measure (0.95) and MOCHA is the system with the lowest measure (0.60).
- *Precision:* For Java, precision results for 1st, 2nd, and 3rd quartiles are 0.76, 0.83, and 0.89, respectively. CHECKSTYLE is the system with the highest precision measure (1.0) and FREEMIND is the system with the lowest measure (0.61). For JavaScript, precision results for 1st, 2nd, and 3rd quartiles are 0.61, 0.75, and 0.83, respectively. REACT NATIVE is the system with the highest precision

[3]https://github.com/github/linguist

[4]We decide to generate a classifier per project, instead of evaluating cross-project predictors, because we presume that the main properties of utility functions may change significantly from a project to another. For example, they depend on different libraries, depending on the project.



(a) Java

(b) JavaScript

Fig. 4. Top-5 best predictors (Java and JavaScript)

measure (0.9) and REACT is the system with the lowest measure (0.52).

- *Recall:* For Java, recall results for 1st, 2nd, and 3rd quartiles are 0.83, 0.88, and 0.91, respectively. APACHE COMMONS COLLECTIONS is the system with the highest recall measure (0.99) and AZUREUS is the system with the lowest measure (0.72). For JavaScript, recall results for 1st, 2nd, and 3rd quartiles are 0.82, 0.89, and 0.95, respectively. REACT and TYPEAHEAD.JS are the system with the highest recall measure (1.0) and REACT NATIVE is the system with the lowest measure (0.66).
- *F-Measure:* For Java, F-Measure results for 1st, 2nd, and 3rd quartiles are 0.80, 0.84, and 0.88, respectively. APACHE COMMONS COLLECTIONS is the system with the highest F-Measure measure (0.98) and FREEMIND is the system with the lowest measure (0.71). For JavaScript, F-Measure results for 1st, 2nd, and 3rd quartiles are 0.71, 0.76, and 0.84, respectively. TYPEAHEAD.JS is the system with the highest F-Measure measure (0.94) and MATERIAL is the system with the lowest measure (0.68).

Therefore, random Forest classifiers can identify utility functions with very high accuracy. However, results for JavaScript are slightly lower than for Java (median F-measure equals to 0.84 and 0.76, for Java and JavaScript, respectively).

B. Most Influential Predictors

For each run of the 10-fold cross validation, we collect a measure of the importance of each predictor. Using this measure, we generate a rank of the most important predictors (on each run). For each predictor, we then calculate its average rank position, by averaging its position in the fold ranks. Figure 4 shows the five best predictors, for Java and JavaScript. The violin plots show the distribution of the rank positions of a given predictor, considering all systems in our dataset. By considering the median rank positions, the best predictors for Java are: LOC, is static, number of references to field variables, is implemented in a subclass, and number of references to Java API or library classes. For JavaScript, the best predictors are: LOC, number of references to this, number of references to global variables, number of parameters, and number of function calls.

Figure 4 also presents a relevant dispersion in the rank positions of the best predictors, depending of the system the classifiers refer to. For example, the rank positions of *is static* for Java ranges from 1 (for 22 systems in our dataset) to 22 (which is found for a single system, APACHE JAMES). LOC is the predictor with the smallest dispersion, both in Java and in JavaScript. These results somehow suggest that classifiers for utility function should be generated for each system and that cross-project predictors probably will present a much lower accuracy than the classifiers we evaluate in this section. In Section VI we discuss how in-project predictors can be used to build recommenders of *Move Utility Function* refactorings.

C. Threats to Validity

Both for Java and JavaScript, we assume that utility libraries have `util` in their names or in their paths. However, we might have utility libraries that do not match this criteria. For example, in BRACKETS we found one utility library called `src/LiveDevelopment/Agents/DOMHelpers.js`. However, by considering a more precise criteria to select utility libraries probably will improve our results.

V. RELATED WORK

To some extent, utility functions implemented in incorrect modules can be seen as instances of the Feature Envy bad smell. JDeodorant is a well-known system to detect this kind of code smell [17]. The system defines an Entity Placement metric to evaluate the quality of a possible Move Method recommendation. This metric is used to evaluate whether a recommendation reduces a system-wide measurement of coupling and at the same time improves a system-wide measurement of cohesion. MethodBook is another system to recommend Move Methods, which considers both structural (*e.g.*, method calls) and conceptual relationships (*e.g.*, comments) with other methods [18]. Finally, JMove recommends Move Method refactorings using structural dependencies [19]. In future work, we plain to compare our results with the ones produced by these systems. However, we can already mention that all of them are implemented to Java and heavily depend on static types. Therefore, it is not straightforward to adapt these systems to dynamic languages, like JavaScript. By contrast, we showed in this paper that Random Forest classifiers provide

high accuracy even for JavaScript. Finally, the precision and recall results of these systems are usually much lower than the ones we reported in this paper.

Random Forest has been used to support several tasks in Software Engineering. For example, some studies are concerned with software defects or fault prediction [9]–[11]. Other studies focus on the support of software development practices, for instance, by using Random Forest to predict delays in issue integration [12] or to help untangling fine-grained code changes (*e.g.*, by separating commits related to bug fix and refactoring) [13]. Finally, another usage example is to support the detection of application characteristics, for instance, the difference between high-rated and low-rated applications in Android [8]. Notice that the list of related work using Random Forest is not exhaustive, but it shows the diversity of its application.

VI. Practical Applications

Tool Support: Utility functions represent less than 10% of the functions in a system (at least, in our dataset of 84 Java systems and 22 JavaScript systems). Therefore, we propose to train machine learning classifiers by considering all utility functions in a system and a sample of the same size of non-utility functions. Typically, this training data will include at most 20% of the functions from a system. Therefore, by relying on the produced classifiers, we can build recommenders of *Move Utility Function* refactorings for the remaining 80% of the system functions.

Application on Other Types of Functions: In theory, the proposed technique can work on any kind of functions that are implemented in well-known modules. As examples, we have database-related functions (normally implemented in Data Access Objects (DAOs) [3], [4]), user-interface functions (normally implemented in `View` components) or even functions related to a domain concept (*e.g.*, functions that manipulate `Loan` related concepts are usually confined in a well-defined component, in banking applications [1]).

VII. Conclusion

In this paper, we showed that Random Forest classifiers can identify with high accuracy utility functions not implemented in correct modules. We achieved with these classifiers median F-measures equal to 0.84 and 0.76, for Java and JavaScript systems, respectively. Implementing utility functions in utility libraries is important to promote their visibility to other developers and therefore increase their chances of reuse. As further work, we first plan to extend the exploratory study, by considering other systems. We also plan to investigate the use of machine learning classifiers to identify other kinds of functions, such as database access functions.

Acknowledgments

This research is supported by CNPq and FAPEMIG.

References

[1] S. Sarkar, S. Ramachandran, G. S. Kumar, M. K. Iyengar, K. Rangarajan, and S. Sivagnanam, "Modularization of a large-scale business application: A case study," *IEEE Software*, vol. 26, pp. 28–35, 2009.

[2] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A curated collection of Java code for empirical studies," in *17th Asia Pacific Software Engineering Conference (APSEC)*, 2010, pp. 336–345.

[3] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, "A recommendation system for repairing violations detected by static architecture conformance checking," *Software: Practice and Experience*, vol. 45, no. 3, pp. 315–342, 2015.

[4] R. Terra, M. T. Valente, K. Czarnecki, and R. Bigonha, "Recommending refactorings to reverse software architecture erosion," in *16th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 335–340.

[5] A. Serebrenik, T. Schrijvers, and B. Demoen, "Improving Prolog programs: Refactoring for Prolog," *TPLP*, vol. 8, no. 2, pp. 201–215, 2008.

[6] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[7] F. Provost and T. Fawcett, "Robust classification for imprecise environments," *Machine learning*, vol. 42, no. 3, pp. 203–231, 2001.

[8] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? a case study on free android applications," in *International Conference on Software Maintenance and Evolution*, 2015.

[9] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *Transactions on Software Engineering*, vol. 34, no. 4, 2008.

[10] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Guéhéneuc, "Can lexicon bad smells improve fault prediction?" in *Working Conference on Reverse Engineering*, 2012.

[11] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *Working Conference on Mining Software Repositories*, 2013.

[12] D. Alencar da Costa, S. L. Abebe, S. McIntosh, U. Kulesza, and A. E. Hassan, "An empirical study of delays in the integration of addressed issues," in *International Conference on Software Maintenance and Evolution*, 2014.

[13] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *International Conference on Software Analysis, Evolution and Reengineering*, 2015.

[14] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha, "Qualitas.class Corpus: A compiled version of the Qualitas Corpus," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 1–4, 2013.

[15] L. Silva, M. Ramos, M. T. Valente, N. Anquetil, and A. Bergel, "Does JavaScript software embrace classes?" in *22nd International Conference on Software Analysis, Evolution and Reengineering*, 2015, pp. 73–82.

[16] D. W. Hosmer Jr and S. Lemeshow, *Applied logistic regression*. John Wiley & Sons, 2004.

[17] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 99, pp. 347–367, 2009.

[18] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 99, pp. 1–10, 2014.

[19] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending move method refactorings using dependency sets," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 232–241.