Random Forest for Code Smell Detection in JavaScript

Diego S. Sarafim[1], **Karina V. Delgado**[1], **Daniel Cordeiro**[1]

[1] School of Arts, Sciences and Humanities – University of São Paulo
Ermelino Matarazzo – SP – Brazil.

{d.s.sarafim, kvd, daniel.cordeiro}@usp.br

Abstract. *JavaScript has become one of the most widely used programming languages. JavaScript is a dynamic, interpreted, and weakly-typed scripting language especially suited for the development of web applications. While these characteristics allow the language to offer high levels of flexibility, they also can make JavaScript code more challenging to write, maintain and evolve. One of the risks that JavaScript and other programming languages are prone to is the presence of code smells. Code smells result from poor programming choices during source code development that negatively influence source code comprehension and maintainability in the long term. This work reports the result of an approach that uses the Random Forest algorithm to detect a set of 11 code smells based on software metrics extracted from JavaScript source code. It also reports the construction of two datasets, one for code smells that affect functions/methods, and another for code smells related to classes, both containing at least 200 labeled positive instances of each code smell and both extracted from a set of 25 open-source JavaScript projects.*

1. Introduction

JavaScript is a powerful, flexible, and widely used scripting language. The 2022 Stack Overflow Developer Survey[1] that JavaScript has been the most commonly used programming language for the last ten years.

The flexibility of JavaScript is at the same time a powerful feature and a factor that makes code more difficult to write and maintain when it is not correctly used.

The quality level attained during the implementation phase can affect the complexity of the source code and considerably increase the costs involved in the later stages of the software lifecycle [Banker et al. 1993]. Due to aspects that include lack of experience, deadline pressures, and even community factors [Tamburri et al. 2019], developers may lack the ability, discipline, time, and willingness to find suitable solutions to recurrent problems and to keep source code complexity under control [Parnas 1994].

One of the direct consequences of the lack of discipline and ability of developers over the implementation phase is the introduction of code smells, symptoms of poor choices during development activities that violate good programming practices that can negatively affect maintainability [Yamashita and Moonen 2013], and the evolution of computer programs [Abbes et al. 2011]. Previous studies have shown that code smells reduce source code comprehensibility [Abbes et al. 2011] and make it more prone to changes and faults [Khomh et al. 2012].

[1]`https://survey.stackoverflow.co/2022/` (Accessed: 15 August 2022)

To mitigate the harmful effects of code smells on software maintenance and evolution, they should ideally be detected and corrected as early as possible. Many approaches, techniques, and tools have been proposed to identify code smells, some of them are commercial products such as SonarQube[2], and some of them are a result of academic works such as JSNose [Fard and Mesbah 2013], Lacuna [Obbink et al. 2018], and SMURF [Maiga et al. 2012]. One very common approach to detecting code smells is the usage of techniques that base their search on predefined declarative rules [Marinescu 2004, Fontana et al. 2012, Padilha et al. 2014]; on the other hand, a relatively recent literature review has found that only a handful conducted studies investigated the usage and effectiveness of machine learning based techniques on the detection of code smells [Azeem et al. 2019]; moreover, the majority of those studies explored code smell detection on Java resulting in a good opportunity for further studies to examine the usage of machine learning techniques on different programming languages especially those that are prone to different and unique types of code smells which is the case of JavaScript [Fard and Mesbah 2013, Almashfi and Lu 2020].

Encouraged by the apparent lack of studies that employ machine learning techniques to detect JavaScript code smells, the fact that JavaScript can be affected by unique types of code smells and by results of previous works that demonstrated the effectiveness of decision tree-based machine learning algorithms in the task of detecting code smells [Amorim et al. 2015] we developed an approach to detect JavaScript code smells by employing Random Forest algorithm on software metrics extracted from JavaScript source code.

To the best of our knowledge, there are no standard code smell datasets for the JavaScript language in the literature. Thus, for the purposes of our study, we need to construct a dataset with metrics extracted from classes and methods/functions. Every instance of the dataset is labeled with values of true or false indicating respectively the presence or absence of each code smell in that piece of code, for example, for each method or function used to build our dataset, we extracted a set of software metrics and labeled the entry with true or false for every code smell that can affect methods and functions. This dataset was then used to train a Random Forest algorithm to correctly classify positive and negative instances of code smells.

Although the construction of the dataset was an important aspect of this study, our main contribution lies in the proposed approach for JavaScript code smell detection which we discuss in detail in Section 4.

The structure of the paper is as follows. Section 2 offers brief descriptions of the 11 code smells we focused in this work along with details of the labeling rules for each type. Section 3 presents our approach to construct the dataset used in this study and Section 4 presents our detection approach in more details. In Section 5 we discuss the results obtained. Finally, Section 6 concludes the paper.

2. Code Smells

Code smell is a term coined by Fowler and Beck [Fowler 2018] to refer to violations of programming principles and good practices. Although errors or incorrect application behavior are not direct manifestations of code smells, it has been shown before that code

[2]`https://www.sonarqube.org/` (Accessed: 15 August 2022)

smells can negatively affect maintainability [Yamashita and Moonen 2013], and the evolution of computer programs [Abbes et al. 2011] as well as make them more prone to changes and faults [Khomh et al. 2012].

This section describes 11 code smells that were the subject of this study. Part of them, namely large class, lazy class, long method, and long parameter list were proposed by Fowler [Fowler 2018] and have been constantly studied in previous works [Fard and Mesbah 2013, Almashfi and Lu 2020, Fontana et al. 2013, Maneerat and Muenchaisri 2011, Arcelli Fontana et al. 2016], those can affect programming languages that support object-oriented programming such as Java and C++. Complex method, unused variable, and undeclared variable were proposed by [Almashfi and Lu 2020]. Long message chain and long scope chaining were proposed by [Fard and Mesbah 2013]. Finally, dead code is another example of code smell that has received attention in previous works [Fard and Mesbah 2013, Obbink et al. 2018, Almashfi and Lu 2020, AlAbwaini et al. 2018].

Following we present 11 code smells along with the rules we followed to manually label their presence or absence in the dataset we constructed for this study. We start presenting those code smells that affect methods and functions and then we present those that affect classes.

2.1. Method/Function Code Smells

The majority of the code smells we focused in our study affect methods and functions.

Complex Method/Function. Complex methods are hard to understand and thus hard to maintain and evolve, the complexity of a function may indicate the need for refactoring of the source code. During the dataset build phase, the instances of this code smell were labeled with the help of SonarQube (version 8.9.2 – Community Edition). Following what was proposed by [Almashfi and Lu 2020], any method/function with cyclomatic complexity higher than 20 was labeled as a positive case of complex method/function. Cyclomatic complexity is a software quality metric that measures the number of linearly independent paths in the source code.

Long Parameter List. All instances of this code smell were labeled with the help of SonarQube, any method or function with a parameter list length that exceeds 5 parameters was labeled as a positive instance in accordance with [Fard and Mesbah 2013].

Dead Code. Dead code (code that is never called or reached) is a threat to maintainability because it can needlessly make code more difficult to understand [Almashfi and Lu 2020, Nguyen et al. 2012]. Instances of this code smell were also labeled with the help of SonarQube which can find portions of code that are not reachable when they are preceded by termination statements such as return or throw statements.

Long Method/Function. Too many lines of code in a method may be an indicator that the method aggregates too many responsibilities. All the instances of this code smell were labeled with the help of SonarQube. Any method or function with more than 105 lines of code and cyclomatic complexity higher than 9 according to [Arcelli Fontana et al. 2016] was labeled as a positive case of this code smell.

Unused Parameter. Unused parameter is any parameter present in the signature of a method that is never accessed, they needlessly make method signatures more difficult to comprehend and can be misleading since no matter which value of the argument passed to such parameters the behavior of the method will be the same. Unused parameters instances were labeled with the help of SonarQube which can detect this type of code smell.

Unused Variable. Variables that are declared and not used take up space, can be misleading, and decreases code clarity [Almashfi and Lu 2020]. Instances of this code smell were labeled with the help of SonarQube which can detect such occurrences.

Undeclared Variable. There are two ways to declare variables in JavaScript: explicit and non-explicit. Explicit declarations are preceded by one of the three keywords `var`, `let` and `const`. Non-explicit declarations omit the usage of those keywords and have the side effect of placing these variables in the global scope of the application. It is strongly recommended to declare variables using `var`, `let`, or `const` keywords and only to declare global variables when they are needed, placing those declarations outside methods to avoid confusion. Instances of this code smell were labeled with the aid of SonarQube which can detect this smell. Any non-explicit variable declaration inside methods or functions was labeled as a positive instance.

Long Message Chain. Long chaining of functions with the dot operator may result in overcomplex control flows that are hard to comprehend, maintain, and evolve. The labeling process for the instances of this code smell was fully manual, methods or functions with 4 or mode chained calls were labeled as positive instances of a long message chain in accordance with [Fard and Mesbah 2013].

Long Scope Chaining. JavaScript allows functions to be declared inside one another, these nested functions are called closures and they can be used to emulate encapsulation which is one of the key principles of object-oriented programming. One of the side effects of closures is that they produce a chain of scopes where inner functions have access to the scope of the functions they are nested in and since multiple nesting is allowed, adding too many levels of nested closures can make a function hard to comprehend and maintain. The labeling process for instances of this code smell was fully manual, following what was proposed by [Fard and Mesbah 2013] functions with more than 3 levels of chained closures were labeled as positive instances of long scope chaining.

2.2. Class Code Smells

Although JavaScript does not fully comply with object-oriented design rules, it does support the implementation of objects and is susceptible to code smells that affect classes. Our study focused on two of them, which we describe in the following.

Large Class. A class that has many responsibilities, many lines of code, or many methods is a class that does too much work and should be refactored into smaller classes that are easier to maintain and evolve. The labeling process for instances of this code smell was manual, any class declaration with at least 400 lines of code along with an attribute or method count higher than 20 or any object with more than 750 lines of code were labeled as positive instances of large class as per [Fard and Mesbah 2013, Almashfi and Lu 2020].

Lazy Class. A class that aggregates too few responsibilities and attributes is a class that may not justify its existence and should be reconstructed. The labeling process for this code smell was manual, classes with less than 40 lines of code and less than 3 properties were labeled as a positive instances of the lazy class as per [Almashfi and Lu 2020].

3. Dataset

Machine learning algorithms such as random forest rely on datasets with labeled instances so that they can be trained and learn how to correctly classify unseen and unlabeled instances they are exposed to. The goal of our study was to employ a multi-label classification of code smells so that our solution could simultaneously detect each type of code smell (if any) affecting a function or a class. To attain that goal, we constructed 2 different datasets with a very similar structure, both are composed of software metrics and indications of the presence or absence of each type of code smell. One dataset contains metrics extracted from classes as well as indications of the presence/absence of those code smells that affect classes, the other one does the same for methods and functions.

All instances included in both datasets were extracted from 25 open-source JavaScript projects cloned from GitHub repositories. Table 1 shows the list of all JavaScript projects explored during dataset construction.

Table 1. Projects used for dataset construction.

ID	Name	Resource	Commit hash
1	Ember	https://github.com/emberjs/ember.js	e2474ba66
2	Map Talks	https://github.com/maptalks/maptalks.js	d69448ee
3	A-Frame	https://github.com/aframevr/aframe	ec0d4d8b
4	Atom	https://github.com/atom/atom	07edc2b25
5	Chart.js	https://github.com/chartjs/Chart.js	6283c6f1
6	TWGL.js	https://github.com/greggman/twgl.js	aa7fe4a
7	Express	https://github.com/expressjs/express	c221b859
8	Axios	https://github.com/axios/axios	cc86c6c
9	Strapi	https://github.com/strapi/strapi	0dd74c685
10	SheetJS	https://github.com/SheetJS/sheetjs	bb99765
11	Cytoscape.js	https://github.com/cytoscape/cytoscape.js	fff9475b
12	PDF Kit	https://github.com/foliojs/pdfkit	7cd6472
13	Three.js	https://github.com/mrdoob/three.js	f31cb9ec8b
14	React	https://github.com/facebook/react	848e802d2
15	JSPaint	https://github.com/1j01/jspaint	4fdd061
16	VueJS	https://github.com/vuejs/vue	6aa11872
17	Meteor	https://github.com/meteor/meteor	cf2b6daf1
18	Xeokit-SDK	https://github.com/xeokit/xeokit-sdk	63e0f8f8
19	Play Canvas	https://github.com/playcanvas/engine	5222de3ad
20	Open Layers	https://github.com/openlayers/openlayers	7ca0aee84
21	Mark Text	https://github.com/marktext/marktext	2bb405a0
22	Apex Charts	https://github.com/apexcharts/apexcharts.js	218bda9f
23	Taro	https://github.com/Cloud9c/taro	f9874da
24	P4wm	https://github.com/douglasbagnall/p4wn	fcdda9f
25	Tudu List	https://github.com/jdubois/Tudu-Lists	d4a14ea

3.1. Labeling process

The first step in the construction of the datasets was the manual detection and labeling of approximately 200 instances of classes, methods, and functions affected by each code smell. A class or a method can be affected by multiple code smells so, every instance received positive or negative flags indicating all code smells affecting them. A portion of the instances was identified with the help of SonarQube (version 8.9.2 – Community Edition). Other code smells instances were found using a search method based on indirect evidence, such as classes with a large or a small number of lines possibly indicating instances of *large or lazy classes*, respectively. Despite the method to find the instances, all of them were manually verified and labeled based on rules described in Section 2.

3.2. Software metrics extraction process

After labeling each instance of the datasets we then computed the software metrics for each class and method/function included in both datasets.

The metric extraction process was implemented in JavaScript with the help of Babel[3] (version 7.17.3), a library that can work as a transpiler and can be used to generate and traverse abstract syntax trees (AST) from JavaScript source code. Generating and traversing an abstract syntax tree is useful as the traversal process allows us to visit all code entities, objects, properties, functions, and code blocks as well as keep track of scope. During the traversal of the AST for a given input file, our process extracts the information it needs to calculate the class metrics and the method/function metrics.

The dataset for classes is composed by 2 columns indicating the presence of the 2 class-related code smells and 19 software metrics that can be extracted from classes, namely: lines of code (LOC), lines of code excluding accessor and mutator methods (LOCNAMM), number of methods (NOM), number of public, private and static methods (NOPM, NOPVM, and NOSM respectively), number of accessor and mutator methods (NOAM), number of methods excluding accessor and mutator methods (NOMNAMM), number of attributes (NOA), number of public attributes (NPA), number of private attributes (NOPVA), number of static attributes (NOSA), weighted methods count (WMC), weighted methods count of not accessor or mutator methods (WMCNAMM), average methods weight (AMW), average methods weight of not accessor or mutator methods (AMWNAMM), weight of class (WOC), number of constructor methods (NOCM) and number of non-constructor methods (NONCM).

The dataset for methods and functions contains a set of 16 method/function-related metrics: lines of code (LOC), reachable lines of code (RLOC), reachable lines of code ratio (RLOCR), cyclomatic complexity (CYCLO), number of parameters (NOP), number of accessed parameters (NAP), accessed parameters ratio (APR), number of declared variables (NOV), number of accessed variables (NAV), accessed variables ratio (NAVR), declared global variables (NOGVD), maximum nesting level of control structures (MAXNESTING), maximum method nesting level (MMNC), maximum message chain length (MaMCL), mean message chain length (MEMCL) and number of message chain statements (NMCS). As well as 9 columns indicating the presence or absence of the 9 code smells that can affect methods and functions.

[3]`https://babeljs.io/` (Accessed: 15 August 2022)

4. Proposed approach

This section describes the proposed approach to detect the JavaScript code smells discussed in Section 2. For this work we adopted a combination of techniques explored in previous studies, that is, the extraction of software metrics through static analysis of JavaScript source code [Fard and Mesbah 2013, Almashfi and Lu 2020] (that was explained in Section 3.2) and the employment of Random Forest, a tree-based machine learning algorithm to perform a multi-label classification over the data in order to detect code smells [Amorim et al. 2015].

4.1. Random Forest for multi-label classification of code smells

Random Forest is a supervised ensemble learning model composed of decision trees, each one using a specific subset of the input features. Each input is subject to the classification process of each tree. The classification result of each tree is taken as a vote, the classification result most voted by the trees becomes the classification result of the forest. Among the advantages of random forests are their ability to generate human-readable classification rules and estimate the importance of each feature for the classification.

For the purposes of our study, we had to implement and train two Random Forest models to perform a muti-label classification. One model was responsible for classifying class-related code smells and the other handled the classification of method-related code smells, they were implemented in Python with help of scikit-learn library (version 1.1.2). The training consisted in exposing the random forests to the two datasets we constructed. The hyperparameter *number of estimators* used for classifying class-related code smells was 11 and for classifying method/function-related code smells was 25. Both values were defined after a search for the values with the best performance, the search method was to increase the number of *number of estimators* and calculate precision, recall, and f-measure using cross-validation until the point where adding more estimators would not result in performance increments.

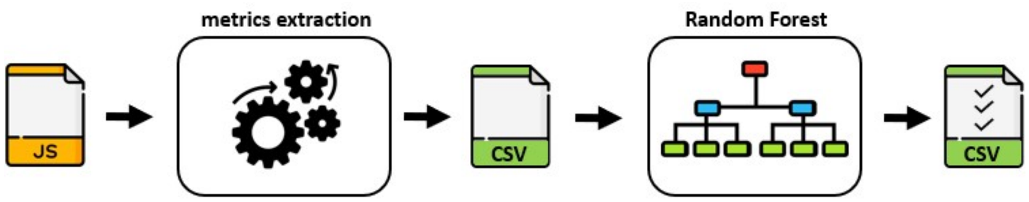
After being trained, the model is ready to be exposed to new unlabeled data and perform the classification task that ultimately indicates what instances are affected by what code smells. The generation of this new data is done by a metric extraction process explained in Section 3.2.

4.2. Testing process

As a result of the metric extraction process, two CSV files are obtained, one contains metrics extracted from classes and the other contains metrics extracted from methods/functions.

Figure 1 shows the stages of the testing process. First, the source is read from a JavaScript file. This code is then used as input into a process that extracts software metrics. The output of this process is two CSV files, one contains metrics of all methods/functions inside the code and the other contains metrics from classes. The corresponding CSV file is used as input to a random forest classifier that has been previously trained specifically for class or method/function code smells. The final result is a CSV file indicating the presence or absence of the corresponding type of code smells as per random forest classification.

Figure 1. Testing process for class or method/function code smells



5. Results

To measure the effectiveness of our approach we adopted three metrics that are broadly used in the machine learning domain to evidence the performance of techniques on classification tasks [Antunes and Vieira 2015], they are Precision, Recall, and F-Measure.

The results were assessed using 10-fold cross-validation to accurately estimate the performance of the model in the predictive task. Tables 2 and 3 present the results obtained by the random forest model for class and method/function code smells, respectively. Those results indicate that our approach demonstrates great effectiveness in detecting most of the code smells we focused on in our study, the only exception is the detection of dead code for which our model predicted no false positive instances but had problems with a high count of false negatives (smelly instances classified as non-smelly).

Table 2. Precision, Recall, and F-measure for class code smells.

Code Smell	**Precision** Avg. (Std.)	**Recall** Avg. (Std.)	**F-Measure** Avg. (Std.)
Large class	0.975 (0.023)	0.972 (0.025)	0.977 (0.023)
Lazy class	0.990 (0.013)	1.000 (0.000)	0.995 (0.004)
Avg. class code smells	0.982	0.986	0.984

Table 3. Precision, Recall, and F-measure for method/function code smells.

Code Smell	**Precision** Avg. (Std.)	**Recall** Avg. (Std.)	**F-Measure** Avg. (Std.)
Complex method	0.989 (0.005)	0.983 (0.016)	0.986 (0.009)
Long parameter list	1.000 (0.000)	0.995 (0.008)	0.997 (0.002)
Long method	0.985 (0.029)	0.962 (0.017)	0.973 (0.019)
Dead code	1.000 (0.000)	0.481 (0.035)	0.649 (0.020)
Unused parameter	0.990 (0.004)	0.903 (0.013)	0.944 (0.006)
Unused variable	0.962 (0.021)	0.807 (0.012)	0.878 (0.015)
Undeclared variable	1.000 (0,000)	0.964 (0.005)	0.982 (0.001)
Long message chain	0.990 (0.030)	0.980 (0.009)	0.985 (0.014)
Long scope chaining	0.995 (0.023)	0.995 (0.010)	0.995 (0.011)
Avg. method/function code smells	0.989	0.937	0.962

We conducted an additional test in which we compared the effectiveness of our approach with the results obtained by SonarQube, a popular platform for continuous code inspection that performs static code analysis and is capable of detecting defects, security breaches and code smells. Unfortunately, only 6 of the 11 code smells analyzed are also covered by Sonar. These 6 code smells are: *complex method*, *long parameter list*, *dead code*, *unused parameter*, *unused variable*, and *undeclared variable*.

For this test we executed a full SonarQube analysis on *Strapi* and *Tudu List* JavaScript projects followed by a full execution of our approach on the same projects, during both analyses, each tool analyzed every JavaScript file inside the projects to detect as many code smells as possible. Table 4 presents the results for each code smells in terms of Precision, Recall and F-Measure.

Table 4. SonarQube vs. our approach on *Strapi* and *Tudu List* projects.

	SonarQube			Our Approach		
Code Smell	**Precision**	**Recall**	**F-Measure**	**Precision**	**Recall**	**F-Measure**
Complex method	1.0	1.0	1.0	1.0	0.968	0.984
Long parameter list	1.0	1.0	1.0	1.0	1.0	1.0
Dead code	1.0	1.0	1.0	0.8	0.666	0.727
Unused parameter	1.0	0.818	0.899	1.0	0.972	0.986
Unused variable	1.0	1.0	1.0	0.916	0.733	0.815
Undeclared variable	1.0	1.0	1.0	1.0	0.989	0.994

As a result, both SonarQube and our approach had very similar results on complex method, long parameter list and undeclared variable code smells. SonarQube did better on the task of detecting dead code and unused variables. For the code smell unused parameter, however, our approach obtained better results when compared with SonarQube.

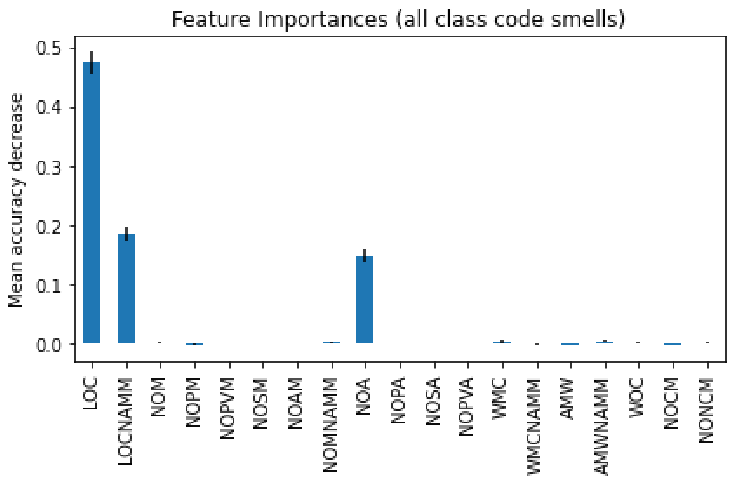
As mentioned in the previous section, random forests have two crucial advantages: (i) the classification process is based on human-readable rules, and (ii) the importance of each feature for the classification can be estimated. We explored both of those advantages next.

Figure 2 shows feature importance for all class and method/function-related code smells classification. The features that most affect the classification of class-related code smells are lines of code (LOC), lines of code excluding accessor, mutator methods (LOCNAMM), and the number of attributes (NOA), while feature importance for method/function-related code smells is more diverse, with cyclomatic complexity (CYCLO) and the number of parameters (NOP) standing out.

Figure 3 depicts one of the constituent trees for the model while classifying *long method* code smell. It is an example of how feature importance plays an important role in the classification task and shows in a human-readable manner one of the rules discovered by the model to classify this type of code smell. In this case, cyclomatic complexity (CYCLO) and lines of code (LOC) were decisive in the classification. Any method with cyclomatic complexity bigger than 16 and more than 106 lines of code would be classified as a *long method* by this tree.

By exploring those abilities from Random Forest, we found that most of the rules generated by the models to classify code smells are identical or very close to the ones used

Figure 2. Feature importance for class and method/function related code smells.



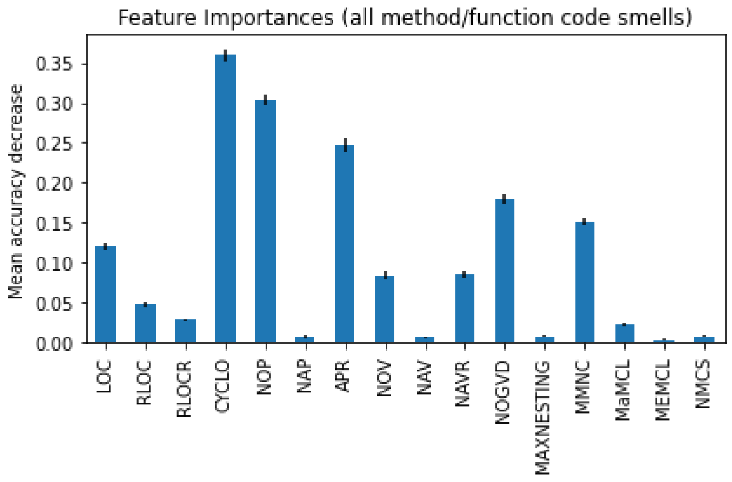
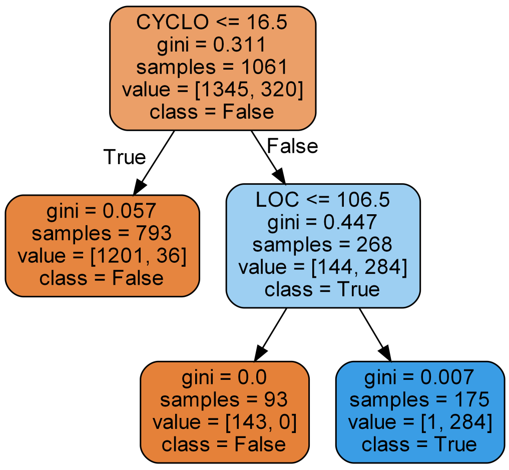


Figure 3. Detection rule for the *long method* code smell.



during the labeling process of our dataset. This indicates that the algorithm is effective in discovering classification rules for code smells and also suggests they could be explored to become the ones that primarily determine the detection rules and the metrics to be used.

6. Conclusion

In this paper, we proposed an approach to detect a set of 11 different JavaScript code smells, using software metrics extracted from static code analysis and two random forest machine learning algorithms to detect classes and methods/functions affected by code smells. Our approach can be used in any phase of the development lifecycle to help mitigate the known bad consequences of code smell presence in programs.

Our empirical evaluation demonstrated that our approach is highly effective in detecting the code smells we focused our study on, except for *dead code*. Such results support previous studies that showed the effectiveness of tree-based algorithms to detect code smells, and it also indicates this can be done to JavaScript source code with its own code smells and particularities as previous works that used machine learning focused on Java. For future work, we would like to improve the classification of dead code and increase the variety of extracted metrics and detected code smells.

References

Abbes, M., Khomh, F., Gueheneuc, Y.-G., and Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th european conference on software maintenance and reengineering*, pages 181–190. IEEE.

AlAbwaini, N., Aldaaje, A., Jaber, T., Abdallah, M., and Tamimi, A. (2018). Using program slicing to detect the dead code. In *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, pages 230–233. IEEE.

Almashfi, N. and Lu, L. (2020). Code smell detection tool for Java Script programs. In *2020 5th International Conference on Computer and Communication Systems (IC-CCS)*, pages 172–176. IEEE.

Amorim, L., Costa, E., Antunes, N., Fonseca, B., and Ribeiro, M. (2015). Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*, pages 261–269. IEEE.

Antunes, N. and Vieira, M. (2015). On the metrics for benchmarking vulnerability detection tools. In *2015 45th Annual IEEE/IFIP international conference on dependable systems and networks*, pages 505–516. IEEE.

Arcelli Fontana, F., Mäntylä, M. V., Zanoni, M., and Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191.

Azeem, M. I., Palomba, F., Shi, L., and Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138.

Banker, R. D., Datar, S. M., Kemerer, C. F., and Zweig, D. (1993). Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–95.

Fard, A. M. and Mesbah, A. (2013). Jsnose: Detecting Javascript code smells. In *2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE.

Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1.

Fontana, F. A., Zanoni, M., Marino, A., and Mäntylä, M. V. (2013). Code smell detection: Towards a machine learning-based approach. In *2013 IEEE international conference on software maintenance*, pages 396–399. IEEE.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Khomh, F., Penta, M. D., Guéhéneuc, Y.-G., and Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243–275.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Guéhéneuc, Y.-G., and Aimeur, E. (2012). Smurf: A svm-based incremental anti-pattern detection approach. In *2012 19th Working Conference on Reverse Engineering*, pages 466–475. IEEE.

Maneerat, N. and Muenchaisri, P. (2011). Bad-smell prediction from software design model using machine learning techniques. In *2011 Eighth international joint conference on computer science and software engineering (JCSSE)*, pages 331–336. IEEE.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359. IEEE.

Nguyen, H. V., Nguyen, H. A., Nguyen, T. T., Nguyen, A. T., and Nguyen, T. N. (2012). Detection of embedded code smells in dynamic web applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 282–285. IEEE.

Obbink, N. G., Malavolta, I., Scoccia, G. L., and Lago, P. (2018). An extensible approach for taming the challenges of Javascript dead code elimination. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 291–401. IEEE.

Padilha, J., Pereira, J., Figueiredo, E., Almeida, J., Garcia, A., and Sant'Anna, C. (2014). On the effectiveness of concern metrics to detect code smells: An empirical study. In *International Conference on Advanced Information Systems Engineering*, pages 656–671. Springer.

Parnas, D. L. (1994). Software aging. In *Proceedings of 16th International Conference on Software Engineering*, pages 279–287. IEEE.

Tamburri, D. A., Palomba, F., Serebrenik, A., and Zaidman, A. (2019). Discovering community patterns in open-source: a systematic approach and its evaluation. *Empirical Software Engineering*, 24(3):1369–1417.

Yamashita, A. and Moonen, L. (2013). Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 682–691. IEEE.