Detecting Code Smells in JavaScript: An Annotated Dataset for Software Quality Analysis

Diego S. Sarafim
Karina V. Delgado
Daniel Cordeiro
d.s.sarafim@usp.br
kvd@usp.br
daniel.cordeiro@usp.br
School of Arts, Sciences and Humanities — University of São Paulo
São Paulo, São Paulo, Brazil

ABSTRACT

The source code quality level attained during the development phase is an important factor in increasing costs in later stages of software development. Among the most detrimental quality problems are code smells, which are violations of both programming principles and good practices that negatively affect the maintainability and evolution of computer programs. Much effort has been put into creating tools for code smell detection over the last decades. A promising approach relies on machine learning (ML) algorithms for automated smell detection. Those algorithms usually need datasets with labeled instances pointing to the presence/absence of smells in programming constructs such as classes and methods. Despite a good number of studies using ML for code smell detection, there is a lack of studies adopting this approach for programming languages other than Java. Even widely popular languages like JavaScript have few or no studies covering the usage of ML models for smell detection despite lexical, structural, and paradigm differences when compared to Java. A symptom of the lack of such studies in JavaScript is the absence of standard code smell datasets for this language in the literature. This work presents a new dataset for code smell detection in JavaScript software focused on detecting God Class and Long Method, two of the most prevalent and harmful code smells. We describe the strategy used for the dataset construction, its characteristics, and a few preliminary experiments using our dataset, along with ML models for code smell detection.

KEYWORDS

dataset, code smells, JavaScript, machine learning, classification

1 INTRODUCTION

One important factor of cost increasing in the later stages of the software development life cycle is the source code quality level attained during the development phase [6]. In many situations, for reasons such as lack of experience, tight deadlines, and even community factors [27], programming professionals may lack the ability, time, discipline, or willingness to find suitable solutions to common problems and to keep source code complexity under control [26].

As a consequence of the inability developers may face to keep code complexity and quality under control, many projects can be affected by the introduction of *code smells*. Code smells are symptoms of poor choices during development activities that violate good programming practices that negatively affect maintainability [30] and the evolution of computer programs [1]. A key common outcome of the presence of code smells is a decrease in source code comprehensibility [1] that makes software more prone to changes and faults [17].

Ideally, code smells should be detected and fixed as early as possible in the software development life cycle in order to mitigate their negative effects. Over the years, tools for code smell detection have been produced both for commercial SonarQube[1] and academic purposes such as SMURF [20], JSNose [11] and more recently Lacuna [24]. When it comes to academic research, a handful of studies employed machine learning techniques to detect either the presence or the severity degree of code smells, finding promising results for both tasks [5]. Although many studies explored interesting techniques and strategies with machine learning, mostly all focus on detecting smells in Java programming language. JavaScript offers good research opportunities, three important reasons for that are (I) it is one of the most popular languages worldwide, (II) to the best of our knowledge was not explored for smell detection with machine learning, and (III) when compared to Java, it has structural, paradigm, and syntactical differences that can affect how code smells are perceived, introduced, or detected [2, 11]. We describe those differences and how they can affect code smell introduction or perception in Section 3.1 of this paper.

Something that is both a consequence and probably part of the reason for the lack of studies that focus on the employment of machine learning in the task of detecting code smells on software JavaScript is the fact that, to the best of our knowledge, there are no standard code smell datasets for the JavaScript language in the literature.

Encouraged by the apparent lack of publicly available datasets for code smell on JavaScript, we set out to construct one comprised of human-labeled positive and negative instances of two of the most pervasive and harmful types of code smells, God Class and Long Method. This dataset is the main contribution of this study and can be used for code smell presence binary detection.

The structure of the paper is comprised of the following sections: In Section 2, we introduce related works on code smell data set construction as well as studies on code smell detection based on machine learning techniques. Section 3 provides an overview of code smells, some of the challenges to detect them, it also presents

[1]https://www.sonarqube.org/ (Accessed: 12 March 2024)

the code smells included in our data set and describes some of the main aspects that distinguish JavaScript in terms of how code smells can be introduced, detected and perceived when compared to Java which is by far the language most explored by studies looking into detection of code smells based on machine learning. Section 4 describes the approach we followed to construct the dataset and Section 5 presents the details of the finished dataset. In Section 6, we present the results obtained employing different machine learning techniques on our finished data set. In Section 7, we discuss limitations and threats to the validity of our work. Finally, Section 8 concludes the paper.

2 RELATED WORKS

To the best of our knowledge, there are no standard code smell datasets for the JavaScript language in the literature. Therefore, in this section, we are limited to describing related works within the scope of different programming languages.

Approaches to the creation of datasets for code smell detection can vary. Those differences include the criteria adopted for project selection, the type or number of code smells included, the features used to represent each instance (i.e., software metrics or word embedding), the goal of the classification—i.e., smell presence binary classification or smell severity classification—and finally the instance labeling strategy.

When it comes to the labeling approach, an option is to fully rely on existing automated tools for code smell detection such as the dataset presented by Lenarduzzi et al. [18]. This approach makes the labeling process much more efficient. Still, it has some major drawbacks: (I) the dataset is limited by the list of smells the tool can detect, (II) whatever model trained with such dataset will very likely learn the underlining detection rules of the automated tool, which may limit reaching the detection via the discovery of new rules or approaches, and (III) it takes real developers with professional experience out of the annotation loop and that may reduce the final quality and precision of the annotations.

Among the approaches that rely on humans conducting the labeling process, we can find the works by Fontana et al. [13] that introduced a data set with 420 instances for each of the four code smells—*Data Class, God Class, Feature Envy, and Long Method*—all of them extracted from Java projects. Their dataset was used for both smell severity classification [4] and smell binary classification [13]. The approach taken by the authors inspired most of the approach we used in our construction process, the most important aspects of their approach are (I) the usage of automated tools to select candidate instances for further human analysis and annotation and (II) a human-centered annotation process conducted by trained evaluators and carefully designed to avoid a known sensibility of code smell detection based on the opinion of a single developer [21]. Three evaluators in total worked on the labeling process and each instance was labeled by all three evaluators. One drawback of this specific dataset is the fact that the projects used are taken from Qualitas Corpus [28] that, despite being constantly used in software engineering research, contains projects as old as 2002, way before the introduction of many recent techniques and features, which can make them less relevant for recent code smell detection [19].

Guggulothu and Moiz [16], Di Nucci et al. [10], and Mhawish and Gupta [22] created new datasets that are modified versions of the dataset created by Fontana et al. [13]. The authors of those three studies either (I) employed feature selection techniques to sort out redundant and less representative features, (II) merged class and method-wise instances to produce multi-class datasets, or (III) reduced the imbalance between positive and negative instances. To the best of our knowledge at this point—8th of April 2024—among them, only Guggulothu and Moiz [16] have made their dataset publicly available.

In a more recent work, Madeyski and Lewowski [19] presented a data set with nearly 1500 instances focusing on four code smells—*Data Class, God Class, Feature Envy, and Long Method*—all extracted from recent Java projects. While they conducted a purely human-centered annotation process with multiple evaluators per instance, they did not include a rechecking process like Fontana et al. [13] did in their approach. The lack of a rechecking process in case of divergence between human evaluators can lead to noise and incorrect labels in case of human error during manual annotation. In this study 20 evaluators worked on the annotation process with an uneven distribution of evaluators per instance. They also did not include any features for the instances in their data set, such as software metrics, leaving this task to interested researchers. Although software metrics are not very complex to obtain, their lack leaves the dataset in a not readily usable state. One detail that distinguishes this dataset from other human-centered annotated smell sets is that the authors included an auxiliary dataset comprised of data extracted from an extensive survey of developers involved in the study to open new research opportunities.

Although the majority of studies that covered code smell detection using machine learning did so with datasets from Java systems, a few studies focused on other languages such as C# [29] and C [31]. Both studies focused on *Duplicated Code*, an important type of code smell. Wang et al. [29] extracted their instances from two projects without names disclosed due to regulation questions leading to replication barriers, and the study by Yang et al. [31] focused on proposing a new classification model. To the best of our knowledge, their dataset is not publicly available.

Our dataset construction approach takes advantage of Fontana et al. [13] by conducting a human-centered annotation process with multiple evaluators with an enforced rechecking process for instances where the initial agreement was not achieved. Like Madeyski and Lewowski [19], we selected more recent projects to extract the instances from. Finally, unlike studies that described the creation of datasets without making them publicly available, our dataset is available to the public[2].

3 CODE SMELLS

Code smell is a term coined by Fowler and Beck [15] 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 smells can negatively affect maintainability [30], the evolution of computer programs [1], as well as make them more prone to changes and faults [17].

[2]https://www.dataset.com/

One commonly discussed aspect of code smells is their objectiveness, which results in differences in their evaluation by developers [21], scarce agreement in results among detectors [12], and even differences in the way they are perceived by different communities of developers[27].

To construct the data set described in this paper, we selected two code smells that pose a considerable negative impact on the software quality [25] and are among the most pervasive code smells [32]. Additionally, following the methodology used by Fontana et al. [13], we focused on two smells that have detection rules defined in the literature; one is at the class level, and the other is at the function/method level. For these reasons, we decided to focus on the code smells *Long Method* and *God Class*.

Long Method. The increase in reading complexity, the existence of too many lines of code or too many statements, and the usage of too many control flow structures in a method may indicate that the method aggregates too many responsibilities. A Long Method tends to be complex, difficult to understand, uses more attributes from other classes than from its own, and tends to centralize the functionality of a class [13].

God 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. A God Class tends to be complex, to have too much code, to implement several different functionalities, and in more extreme cases, it tends to centralize the system's intelligence [13].

3.1 Code Smells in JavaScript

JavaScript is a flexible and widely used programming language. The 2023 edition of the Stack Overflow Developer Survey[3] found that for the eleventh consecutive year, JavaScript was the most commonly used programming language.

Although it is a formidable feature and a major component of what makes the language so powerful, the flexibility of JavaScript is a factor that can make code more challenging to write, read, and maintain when not correctly used.

JavaScript is a prototype-based language. It proposes a class-free style of object-oriented programming that allows objects to inherit properties from other objects directly. It also allows prototypes to be redefined at runtime, leading to immediate redefinition of all the objects referring to them. In JavaScript, object properties including their methods can be created, changed or deleted at runtime, this is a major difference between JavaScript and Java as Java is more rigid and class-based not allowing class redefinition at runtime.

Another aspect that differentiates JavaScript from Java is that JavaScript functions are first-class values which means they can contain nested functions and properties, they can be stored in variables and be passed as arguments to other functions. Functions in JavaScript can even be objects themselves.

Such dynamism not only poses difficulties to common software analysis techniques such as static analysis and even manual code inspection but also can change how code smells are introduced and perceived. For instance, a prototype can contain an acceptable number of lines, number of methods, number of attributes, overall methods complexity in the place it is first defined just to be redefined by lines of code in a different place. Additionally, functions can have nested functions and attributes that can be redefined at runtime that in turn can have other functions or objects assigned to them.

Due to those differences and all this dynamism, JavaScript source code static analysis is more challenging, manual analysis is more time-consuming and code smell detection is more error-prone especially in large code bases [11].

4 DATASET CONSTRUCTION PROCESS

Supervised machine learning algorithms 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. Our study aimed to construct a data set that can be used on the classification of code smells so that our solution could simultaneously detect each type of code smell (if any) affecting a method or a class. To attain that goal, we constructed two different datasets with a very similar structure, both of which 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.

As mentioned in Section 3, we selected two of the most common and harmful types of code smells to focus on during the construction of the data set: God Class and Long Method. The composition of each data set and the strategy we used to construct them were inspired by previous studies [4, 13] that achieved good results on detecting code smells using machine learning models. The whole construction process for the dataset can be divided into three main phases: (I) the construction, setup, and execution of an automated advisor to help us conduct a stratified sampling, using it as an initial filter that could provide us with positive and negative candidate instances of God Class and Long Method, for this phase we extracted a reduced set of software metrics to be used by the automated advisor; (II) a manual evaluation process conducted by MSc students with programming experience and software developer professionals to produce the ground truth labels for the positive and negative smell instances; and finally (III) the extraction of a broader set of software metrics to compose the dataset as the defining features of each instance of class or method. Figure 1 illustrates the three phases and main activities executed during the construction of the dataset. The activities are numbered in the order in which they occurred and are listed within the sections that describe each phase.

Most of the construction process, which is presented in more detail in Sections 4.1, 4.2, and 4.3, was inspired by a strategy employed by Fontana et al. [13] for the construction of a data set for code smell detection on Java software. Finally, Section 5 presents details of the finished dataset.

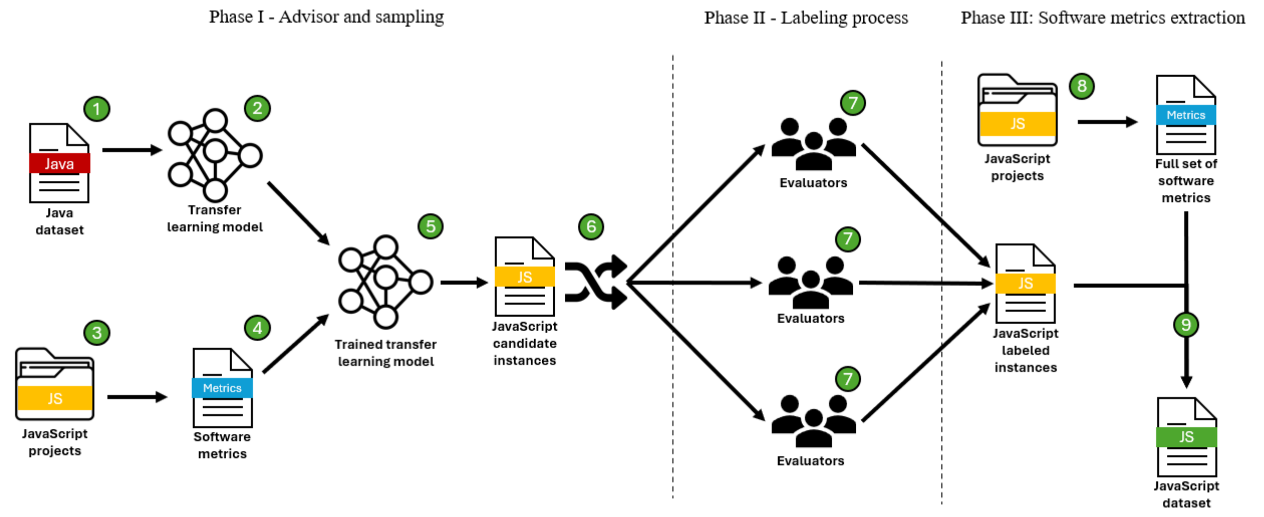
4.1 Phase I: Advisor and Sampling

In this study, we refer as an *advisor* to any tool, method, or technique capable of detecting the presence of code smells with at least some expected—even if low—success rate.

The low density of code smells in source code poses a big drawback if the selection of instances for a dataset is achieved by pure random sampling. Such selection will, in most cases, produce very

[3]https://survey.stackoverflow.co/2023/, (Accessed: 12 March 2024)

Figure 1: Data set construction phases.


Phase I - Advisor and sampling
Phase II - Labeling process
Phase III: Software metrics extraction
1
Java
Java dataset
2
Transfer learning model
3
JS
JavaScript projects
4
Metrics
Software metrics
5
Trained transfer learning model
JS
JavaScript candidate instances
6
7
Evaluators
7
Evaluators
7
Evaluators
JS
JavaScript labeled instances
8
JS
JavaScript projects
Metrics
Full set of software metrics
9
JS
JavaScript dataset

imbalanced sets that lack enough positive instances—those affected by code smells—and machine learning models trained with such imbalanced sets are more likely to produce bad and imprecise results. Moreover, manually evaluating thousands of source code files from several projects to gather a sufficient amount of positive instances may require much effort that is too big to handle when human resources are limited.

One way to deal with this problem is using a stratified sampling that relies on indications made by one or more advisors whose purpose is to provide hints at the presence or absence of code smells in classes or methods.

As illustrated in Figure 1 and described in this section, the main activities in phase I are:

(1) Acquisition of a Java code smell dataset;
(2) Training of a transfer learning model using Java dataset;
(3) Selection and cloning of open source JavaScript projects;
(4) Extraction of some software metrics from those projects;
(5) Using transfer learning model to detect candidate instances; and
(6) Stratified random sampling of candidate instances.

Fontana et al. [13] selected as advisors, external tools that detect code smells using deterministic rules. For this work, we choose to construct our own advisors using transfer learning, a technique that utilizes a trained model for a specific task that is re-purposed to execute a different but related task. In our case, we trained a model using an existing Java code smell data set and used it to make predictions about the presence or absence of the same code smells on JavaScript code. Although some error rate was expected due to the differences between Java and JavaScript outlined in Section 3.1, by using our advisor's classification as a starting point, we could greatly reduce the number of instances to be manually analyzed when compared to the universe of instances contained in all selected projects. For our work, we used two advisor models, one to classify instances of classes to detect the presence of God Class and another to predict the presence of Long Method on method instances.

The Java code smell data set we used to train our two advisor models is comprised of software metrics calculated from classes and methods. Unlike Java, JavaScript does not possess proper object-oriented programming (OOP) capabilities. This implies that some metrics calculated following OOP principles cannot be calculated for JavaScript. Due to those differences affecting metrics calculation, we reduced the number of features we used from the original data set to train the advisors.

Following a common practice for previous works that used data sets for code smell detection using machine learning models, all instances that were classified by our advisor and later included in our data set were extracted from a set of open-source JavaScript projects cloned from GitHub repositories. Table 1 shows the list of all the projects we used.

Table 1: Projects used for dataset construction.

Name	Resource	Hash
Map Talks	github.com/maptalks/maptalks.js	d69448ee
Atom	github.com/atom/atom	07edc2b25
Chart.js	github.com/chartjs/Chart.js	6283c6f1
JSPaint	github.com/1j01/jspaint	4fdd061
Xeokit-SDK	github.com/xeokit/xeokit-sdk	63e0f8f8
Play Canvas	github.com/playcanvas/engine	5222de3ad
Open Layers	github.com/openlayers/openlayers	7ca0aee84
Apex Charts	github.com/apexcharts/apexcharts.js	218bda9f

After training both advisor models, we employed a metric calculation process to extract from JavaScript classes/methods the same set of software metrics the models were trained on. The set of metrics for both methods and classes extracted for the first phase to use with the advisors is reduced when compared to the set of metrics calculated for the final phase of the construction process.

The class and method metrics calculated in the first phase to be used by the automated advisors are listed on Tables 2 and 3. The broader set of metrics extracted for the composition of the final data set, as well as how the metrics calculation process was executed, are presented in Section 4.3.

Table 2: Initial class related metrics.

Metric	Description
(LOC)	lines of code
(LOCNAMM)	lines of code excluding accessor and mutator methods
(WMC)	weighted methods count
(WMCNAMM)	weighted methods count of not accessor or mutator methods
(NOM)	number of methods
(NOMNAMM)	number of methods excluding accessor and mutator methods

Table 3: Initial method related metrics.

Metric	Description
(LOC)	lines of code
(CYCLO)	cyclomatic complexity
(NOP)	number of parameters
(MAXNESTING)	maximum nesting level of control structures

We applied the metric extraction process to calculate metrics for all classes and methods contained on each of the 8 open-source projects listed on Table 1, which resulted in approximately 2,000 class instances and 16,000 method instances upon which we conducted a stratified random sampling to gather roughly 200 positive candidate instances and 400 negative candidate instances of both God Class and Long Method. The process resulted in two sets with 600 instances each, with an imbalance of 1/3 positive and 2/3 negative candidate instances.

4.2 Phase II: Labeling Process

This labeling process was carefully designed and conducted to reduce the bias given by the sensibility of code smell detection based on a single developer opinion, as pointed by Mäntylä et al. [23].

As illustrated in Figure 1 and described in this section, the main activity in phase II is:

(7) Each selected instance is evaluated by three human evaluators.

Our main objective was to create one data set with 420 instances for each code smell, each set should have 140 positive instances and 280 negative instances to keep the 1/3 to 2/3 imbalance. To achieve this goal, we had to conduct a manual evaluation to label each instance until we reached the desired number of positive and negative instances among the 1,200 instances. The evaluation was performed by MSc students with software development experience and by professional developers, all of them were specifically trained for the task. The labeling process was conducted by 13 evaluators, and each instance was labeled by exactly three evaluators. The training consisted of presentations and discussions regarding the nature of code smells and the usual characteristics of both God Class and Long Method smells. The presentations and discussions considered both the dynamic nature of JavaScript and a set of guidelines for God Class and Long Method code smell detection proposed by Fontana et al. [13], which are:

God Class:

- God classes tend to access many attributes from many other classes; the number of attributes contained in other classes that are used from the class, considering also attributes accessed using accessors methods, tends to be high;
- God classes usually contain large and complex methods;
- God classes are large;
- God classes usually expose a large number of methods.

Long Method:

- Long methods tend to be complex;
- Long methods tend to access many attributes;
- Long methods contain many lines of code;
- Long methods tend to have many parameters.

During the manual labeling phase, each instance was individually labeled by three different evaluators. During the manual evaluations, all participants should not communicate with each other and could only use a source code editor with code highlighting as a tool. No plugins or options to detect any problem in the code were allowed. No metrics were exposed, calculated, or considered, and neither did the participants know the classification results given by the advisor models.

In the first round of labeling, each instance was given a label that indicates the severity degree for the specific code smell, God Class for class instances and Long Method for method instances. The initial severity labeling followed an ordinal scale:
0 — *no smell*: the class (or method) is not affected by the smell;
1 — *light*: the class (or method) is only partially affected by the smell;
2 — *moderate*: the smell characteristics are all present in the class or method;
3 — *severe*: the smell is present and has high values of size, complexity, or coupling.

When the labeling was finished, each instance label was collapsed to a binary label: {0}⟶NEGATIVE, {1, 2, 3}⟶POSITIVE. For cases when there was conflict on the binary labels (total agreement regarding the presence or absence of the code smell in a particular instance was not met), the participants reached an agreement to decide which label to apply. The process was repeated until we reached two sets: (I) a set with labels for 420 classes, of which 140 are positive and 280 are negative instances of God Class, and (II) a set with labels for 420 methods, of which 140 are positive and 280 are negative instances of Long Method.

4.3 Phase III: Software Metrics Extraction

As illustrated in Figure 1 and described in this section, the main activities in phase III are:

(8) Increased set of metrics is extracted from the projects;

(9) Labeled instances and metrics combined compose the dataset.

As a final step of the construction process, we computed software metrics for all 420 classes and 420 methods manually annotated in the labeling phase. Our decision to rely on software metrics for constructing the data set is attributed to good results from previous works that applied machine learning to detect code smells using such metrics as defining features [5].

Table 4 contains names and descriptions of all metrics computed from all labeled classes, and Table 5 contains names and descriptions of all metrics computed from all labeled methods.

Table 4: Final class related metrics list.

Metric	Description
(LOC)	lines of code
(LOCNAMM)	lines of code excluding accessor and mutator methods
(NOA)	number of attributes
(NOM)	number of methods
(NOAM)	number of accessor and mutator methods
(NOMNAMM)	number of methods excluding accessor and mutator methods
(WOC)	weight of class
(WMC)	weighted methods count
(AMW)	average methods weight
(WMCNAMM)	weighted methods count of not accessor or mutator methods
(AMWNAMM)	average methods weight of not accessor or mutator methods

Table 5: Final method related metrics list.

Metric	Description
(LOC)	lines of code
(CYCLO)	cyclomatic complexity
(MAXNESTING)	maximum nesting level of control structures
(NOAV)	number of accessed variables
(NOLV)	number of local variables
(NOP)	number of parameters
(MaMCL)	maximum message chain length
(MeMCL)	mean message chain length

The metric extraction process was implemented in JavaScript with the help of Babel[4] (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 class and method metrics.

[4]https://babeljs.io/ (Accessed: 12 March 2024)

Besides calculating software metrics, we also extracted additional information to help us uniquely identify each instance of method and class contained in our data set. Table 6 describes the identification data we extracted from instances of classes and methods.

Table 6: Additional information extracted from classes and methods.

Name	Description
(ID)	unique identifier of the class or method instance
(Project)	project that the class or method belongs to
(File)	path to the file that contains the method or class starting from project root directory
(Start)	line number where the class or method starts
(End)	line number where the class or method ends

5 DATASET DESCRIPTION

The finished dataset comprises two comma-separated value (CSV) files, one containing the set of class instances and the other containing the set of method instances. Figures 2 and 3 illustrate the final format of each of those sets, class and method-related, respectively.

Both sets contain information for class/method instance identification comprised of a unique ID, the project that contains the method or class, the path to the file that contains the method or class relative to the root directory of the project as well as the start and end lines of the respective class/method.

After the identification data, comes the software metrics calculated for the class or method in question followed by a binary label indicating the presence or absence of Long Method for method instances or God Class for class instances.

Figure 2: Class set schema.

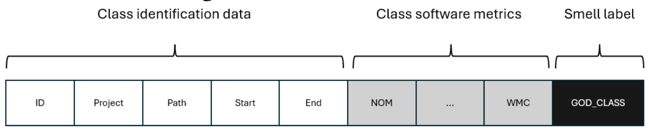
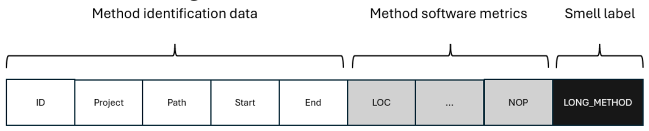


Figure 3: Method set schema.



The finished dataset comprises 420 instances of classes and 420 of methods, each having 140 positive instances—instances affected by the specific smell—and 280 negative instances. Table 7 indicates how many instances were included in the final dataset from each project.

Table 7: Instances per project.

Project	Class Instances	Method Instances	Total
Map Talks	72	43	115
Atom	51	62	113
Chart.js	29	34	63
JSPaint	29	37	66
Xeokit-SDK	67	55	122
Play Canvas	58	72	130
Open Layers	77	52	129
Apex Charts	37	65	102

As a result of having the finished dataset, we could assess two important questions regarding our construction approach: (I) What was the overall performance of the transfer learning advisor we used during the first phase? and (II) How well machine learning models could perform using our data set? We present the answer to the first question in this section and the answer for the second one in Section 6, where we discuss some preliminary experiments we conducted with our finished data set.

Table 8 indicates the performance of our model advisor when compared to the ground truth data obtained from the manual annotation process for the instances included on the finished dataset.

Table 8: Advisor performance.

Code Smell	Evaluated	Correct	Incorrect
God class	420	329	91
Long method	420	356	64

6 EXPERIMENTS AND RESULTS

Supervised machine learning algorithms have shown promising results in the task of code smell detection, as pointed out by Azeem et al. [5]. The learning process of such algorithms—when used for classification tasks—relies on training datasets comprised of *features* that adequately represent the object of the classification—i.e., software metrics, demographics data, etc.—and *targets* representing the correct outcome of the classification—i.e., labels indicating the presence of code smells, the price value of an item or the content of an image. A useful dataset for a classification task—i.e., classification indicating the presence/absence of code smells—should be comprised of trustworthy targets and features that adequately represent characteristics of the classification subject that possess relationships with the target.

To confirm that our dataset is adequate to support the detection of God Class and Long Method in JavaScript code using machine learning techniques, we conducted preliminary experiments with three different algorithms. In this section, we present those algorithms and their results.

6.1 Algorithms

Results from a recent literature review on the usage of machine learning techniques for code smell detection [5] showed that two of the best-performing algorithms for such tasks were JRip and Random Forest. Additionally, Support Vector Machine (SVM) also achieved good performance in code smell prediction as found by Fontana et al. [13] and Fontana et al. [14]. Due to the promising results achieved in previous studies, we decided to explore these three algorithms as a preliminary experiment on the data set to assess their effectiveness. Next, we briefly introduce each algorithm, JRip, Random Forest, and Support Vector Machine, respectively.

JRip is an implementation of RIPPER [8] (Repeated Incremental Pruning to Produce Error Reduction) and works as a classification and rule induction algorithm that uses a rule-based approach to generate classification rules from datasets. JRip operates by building and refining rule sets through rule generation, testing, and pruning iteratively.

Random Forest [7] works by forming a forest composed of random decision trees, each using a specific subset of the input features. Each tree performs a classification that is taken as a vote, and the algorithm then chooses the classification that has the most votes among all the trees in the forest.

Support vector machine (SVM) [9] works by finding a hyperplane in the data space that produces the largest minimum distance (called margin) between the samples that belong to different classes. This hyperplane is called the maximum margin hyperplane, and SVM uses the instances on the edges of the margin (called support vectors) to classify every instance.

6.2 Experimental setup

We conducted experiments for both types of code smell using our finished dataset and the three algorithms.

We trained two models of each algorithm, one model for each type of code smells.

To measure the performance of the machine learning algorithms, we adopted three metrics that are broadly used in the machine learning domain [3], which are Precision, Recall, and F-Measure. The results were assessed using 10-fold cross-validation to accurately estimate the performance of each model in the predictive task.

In our experiments with JRip we used JRip implementation available in Waikato Environment Knowledge Analysis (WEKA)[5]. For our tests with Random Forest and Support Vector Machine models, we used their respective implementations available in the Python library scikit-learn[6] (version 1.4.1).

Hyperparameters tuning is an important aspect of the machine learning training process because it has an impact on the predictive performance of models. During our experiments, we employed a manual search as a hyperparameter tuning strategy and were able to achieve good results with just a few iterations.

During God Class classification experiments, we kept most of the hyperparameters with the default values as they are proposed by the libraries we used to implement them.

For JRip, we only changed the value of *optimizations* from 2 to 10. For Random Forest, we changed the parameters *n_estimators* (which controls the number of trees in the forest) from 100 to 10 and *random_state* (that controls the randomness of the process that

[5]https://www.cs.waikato.ac.nz/ml/weka/ (Accessed: 12 March 2024)
[6]https://scikit-learn.org/ (Accessed: 12 March 2024)

is responsible for building the threes) from *None* to 0. Finally, for SVM, we did not change any hyperparameter default value.

During Long Method classification experiments, similarly, as with God Class, we kept most hyperparameters unchanged. For JRip, we only changed the value of *optimizations* to 95. For Random Forest, we changed the parameters *n_estimators* to 25 and the *random_state* to 0. Finally, for SVM, we did not change the default value of any hyperparameter.

Sections 6.3 and 6.4 present the performance achieved by all three algorithms on the code smell classification task. They also describe how we explored features of those algorithms that provide hints on the learning process and rules they devised for the classification of both types of code smells.

6.3 God Class Results

Table 9 presents the performance achieved by the three algorithms for the God Class classification task. Because we used a cross-validation test to estimate model performance, we present Precision, Recall and F-Measure using their average values and standard deviation, except for JRip for which we were not able to extract standard deviation using WEKA's implementation. All models achieved good and very similar results with a minor margin in favor of Random Forest. Those results are consistent with previous studies that pointed to a good performance for these algorithms for code smell classification tasks.

Table 9: Model performance for God Class classification.

Algorithm	**Precision** Avg. (Std.)	**Recall** Avg. (Std.)	**F-Measure** Avg. (Std.)
JRip	0.93 (—)	0.94 (—)	0.93 (—)
Random Forest	0.94 (0.038)	0.94 (0.034)	0.94 (0.036)
SVM	0.94 (0.032)	0.93 (0.033)	0.94 (0.032)

Two of the algorithms we used during our experiments (JRip and Random Forest) have interesting features that can be explored to provide us hints about each model's classification process.

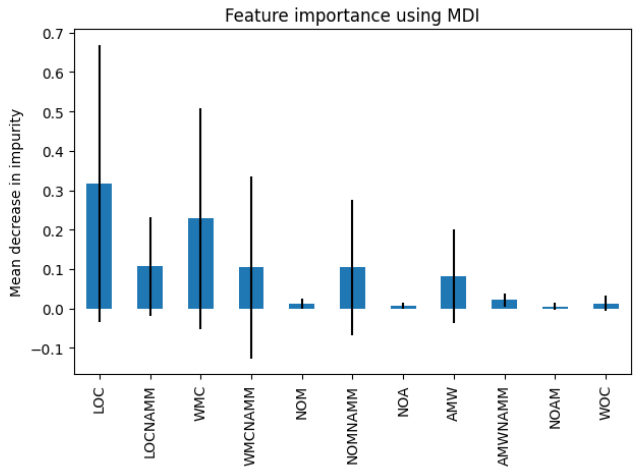
JRip displays the precise detection rules discovered and used by the algorithm as part of its results. During the classification task, for God Class classification, the rule produced by JRip was:

(LOC > 236 and AMW > 2.63) or (LOC > 276) or (WMC > 47)

For Random Forest, it is possible to export the importance of each feature—software metrics in our case—to the classification process. Figure 4 shows the feature importance for God Class classification. The features that most affect are lines of code (LOC), followed by weighted methods count (WMC), lines of code excluding accessor, mutator methods (LOCNAMM), weighted methods count of not accessor or mutator methods (WMCNAMM), number of methods excluding accessor and mutator methods (NOMNAM), and finally average methods weight (AMW).

Those results indicate that Random Forest considered a broader set of metrics during the classification process when compared to JRip. Since both algorithms achieved very good and similar results, this difference may indicate that either Random Forest is more suited to better represent the complex nature of this code smell or that JRip did a better job discovering a simpler generalization rule. Such discussion is out of the context of this work and can be better explored in future research.

Figure 4: Feature importance for God Class classification.



Finally, we compare our results for JavaScript with the results obtained by Fontana et al. [13] for a Java-based dataset for the God Class. The rule found by Fontana et al. [13] with JRip is:

WMCNAMM > 47 .

This detection rule is similar to the third part of the rule for JavaScript generated by JRip (**WMC > 47**) both in terms of value and the metrics involved. The metric WMCNAMM (weighted methods count of not accessor or mutator methods) is closely related to the metric WMC (weighted methods count) in the JavaScript rule since WMCNAMM has the same calculation base excluding getter and setter methods. Additionally, despite WMCNAMM not appearing in the rule generated by JRip for JavaScript, it is among the most important features for Random Forest classification, as shown in Figure 4.

6.4 Long Method Results

Table 10 presents the performance achieved by the three algorithms for Long Method classification. Because we used a cross-validation test to estimate model performance, we present Precision, Recall and F-Measure using their average values and standard deviation, except for JRip for which we were not able to extract standard deviation using WEKA's implementation. All models achieved good and very similar results, but this time JRip had a slightly better performance compared to the others. Those results, once again, are consistent with previous studies that pointed to a good performance for these algorithms for code smell classification tasks.

We also exported JRip detection rules and Random Forest feature importance. JRip produced the following detection rule:

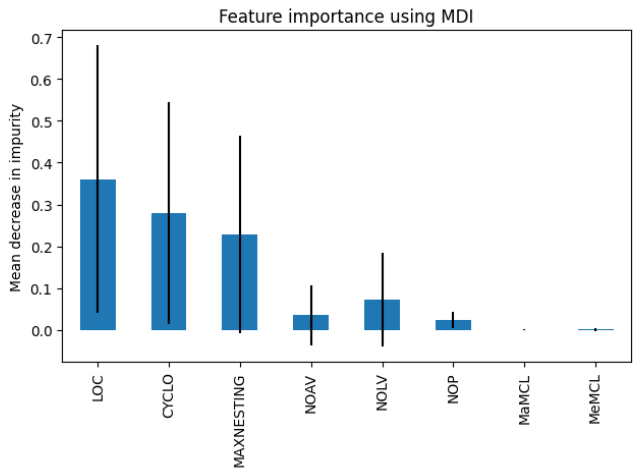
(CYCLO > 9 and LOC > 40) or (LOC > 32 and MAXNESTING > 5)

Figure 5 shows that the most important features for classification of Long Method for Random Forest are lines of code (LOC), cyclomatic complexity (CYCLO), and maximum nesting level of control structures (MAXNESTING).

Table 10: Model performance for Long Method classification.

Algorithm	**Precision** Avg. (Std.)	**Recall** Avg. (Std.)	**F-Measure** Avg. (Std.)
JRip	0.95 (—)	0.95 (—)	0.95 (—)
Random Forest	0.95 (0.027)	0.94 (0.038)	0.94 (0.033)
SVM	0.93 (0.037)	0.92 (0.041)	0.93 (0.039)

Figure 5: Feature importance for Long Method classification.



Results here indicate that both Random Forest and JRip models produced detection rules that mostly relied on the same three metrics: lines of code (LOC), cyclomatic complexity (CYCLO), and maximum nesting level of control structures (MAXNESTING).

Again, we compare our results for JavaScript with the results obtained by Fontana et al. [13] for a Java-based dataset for the Long Method. The rule found by Fontana et al. [13] with JRip for the Long Method is:

$$\textbf{CYCLO} > \textbf{8 and LOC} > \textbf{79} \quad .$$

This detection rule is similar to the first part of the rule for JavaScript generated by JRip (**(CYCLO > 9 and LOC > 40)**) in terms of the metric involved. For CYCLO the rules have very similar values. It is worth noting that LOC and CYCLO appear in both rules and are the two most important features for Random Forest classification, as shown in Figure 5.

7 THREATS TO VALIDITY

One important threat to any study that focuses on code smell detection is the inherent subjectivity they hold. Even experienced developers can disagree with one another on whether a code smell candidate is indeed a real smell. The manual evaluation is, therefore, subject to a certain degree of error, distortion, and bias, especially when conducted by a single developer. There is no easy way to avoid this threat completely, but we believe our approach of aggregating the opinions of three evaluators' opinions considerably alleviates this threat.

A very important threat specific to our study is bias and generalization. All candidate instances later randomly selected for human analysis were gathered according to the results of an advisor trained to detect code smells with a Java dataset. This approach has one major advantage and one major drawback. The advantage is that it drastically reduces the time needed for us to detect potential smelly instances. The drawback is that it can cause distortions and generalization problems. Since it relies on an advisor trained with Java data, it introduces a higher probability that selected smelly instances are similar to the ones found in Java, and since code smells can affect different languages in different ways, as explained in Section 3.1. This may pose a special threat to producing a training set that represents the entire code smell domain in terms of JavaScript programming language.

The lack of other JavaScript code smell datasets hinders our ability to perform benchmark comparisons and learn from previous mistakes, which are powerful tools that certainly help during the planning and execution of such work.

8 CONCLUSION

This paper presented a dataset for code smell detection in JavaScript. The construction approach we used in our dataset, which focuses on the God Class and Long Method (two of the most pervasive and harmful code smells described in the literature), was designed to avoid known bias problems and to mitigate the risks posed by code smell subjectivity.

We described the whole approach we took to construct this data set, starting with the employment of a transfer learning technique to produce an advisor model that we used to help us identify candidate instances, followed by a stratified random sampling of the candidate instances that were then labeled in a human-centered process with enforced multi-person analysis and agreement of each instance to produce labels. These labels were, finally, combined with software metrics extracted from each instance to compose the finished data set. Moreover, we discussed the advantages and drawbacks of other works, including the ones that inspired our construction strategy.

We also described the results of preliminary experiments with three machine learning algorithms that delivered good results in code smell classification in previous studies. The results of our experiments were promising and helped indicate our dataset's usefulness when used as a training set for machine learning classification models.

ARTIFACT AVAILABILITY

As described in Section 5 our dataset is divided into two CSV files, one with class and one with method instances. Both files can be found at:
https://github.com/d-sarafim/js-code-smells-dataset.

REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 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*. IEEE, 181–190.

[2] Nabil Almashfi and Lunjin Lu. 2020. Code smell detection tool for Java Script programs. In *2020 5th International Conference on Computer and Communication Systems (ICCCS)*. IEEE, 172–176.

[3] Nuno Antunes and Marco Vieira. 2015. On the metrics for benchmarking vulnerability detection tools. In *2015 45th Annual IEEE/IFIP international conference on dependable systems and networks*. IEEE, 505–516.

[4] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems* 128 (2017), 43–58. https://doi.org/10.1016/j.knosys.2017.04.014

[5] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138.

[6] Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. 1993. Software complexity and maintenance costs. *Commun. ACM* 36, 11 (1993), 81–95.

[7] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (01 Oct 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[8] William W. Cohen. 1995. Fast Effective Rule Induction. In *Machine Learning Proceedings 1995*, Armand Prieditis and Stuart Russell (Eds.). Morgan Kaufmann, San Francisco (CA), 115–123. https://doi.org/10.1016/B978-1-55860-377-6.50023-2

[9] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20, 3 (01 Sep 1995), 273–297. https://doi.org/10.1007/BF00994018

[10] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 612–621. https://doi.org/10.1109/SANER.2018.8330266

[11] Amin Milani Fard and Ali Mesbah. 2013. Jsnose: Detecting Javascript code smells. In *2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 116–125.

[12] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.* 11, 2 (2012), 5–1.

[13] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (2016), 1143–1191.

[14] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mäntylä. 2013. Code smell detection: Towards a machine learning-based approach. In *2013 IEEE international conference on software maintenance*. IEEE, 396–399.

[15] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[16] Thirupathi Guggulothu and Salman Abdul Moiz. 2020. Code smell detection using multi-label classification approach. *Software Quality Journal* 28, 3 (01 Sep 2020), 1063–1086. https://doi.org/10.1007/s11219-020-09498-y

[17] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.

[18] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. 2019. The Technical Debt Dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering* (Recife, Brazil) *(PROMISE'19)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/3345629.3345630

[19] Lech Madeyski and Tomasz Lewowski. 2020. MLCQ: Industry-Relevant Code Smell Data Set. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering* (Trondheim, Norway) *(EASE '20)*. Association for Computing Machinery, New York, NY, USA, 342–347. https://doi.org/10.1145/3383219.3383264

[20] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabane, Yann-Gaël Guéhéneuc, and Esma Aimeur. 2012. Smurf: A svm-based incremental anti-pattern detection approach. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 466–475.

[21] Mika V. Mäntylä and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering* 11, 3 (01 Sep 2006), 395–431. https://doi.org/10.1007/s10664-006-9002-8

[22] Mohammad Mhawish and Manjari Gupta. 2019. Generating Code-Smell Prediction Rules Using Decision Tree Algorithm and Software Metrics. *International Journal of Computer Sciences and Engineering* 7 (05 2019), 41–48. https://doi.org/10.26438/ijcse/v7i5.4148

[23] M.V. Mäntylä, J. Vanhanen, and C. Lassenius. 2004. Bad smells - humans as code critics. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 399–408. https://doi.org/10.1109/ICSM.2004.1357825

[24] Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia, and Patricia Lago. 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)*. IEEE, 291–401.

[25] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I.K. Sjøberg. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *2010 IEEE International Conference on Software Maintenance*. 1–10. https://doi.org/10.1109/ICSM.2010.5609564

[26] David Lorge Parnas. 1994. Software aging. In *Proceedings of 16th International Conference on Software Engineering*. IEEE, 279–287.

[27] Damian A Tamburri, Fabio Palomba, Alexander Serebrenik, and Andy Zaidman. 2019. Discovering community patterns in open-source: a systematic approach and its evaluation. *Empirical Software Engineering* 24, 3 (2019), 1369–1417.

[28] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference*. 336–345. https://doi.org/10.1109/APSEC.2010.46

[29] Xiaoyin Wang, Yingnong Dang, Lu Zhang, Dongmei Zhang, Erica Lan, and Hong Mei. 2012. Can I clone this piece of code here?. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 170–179. https://doi.org/10.1145/2351676.2351701

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

[31] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2015. Classification model for code clones based on machine learning. *Empirical Software Engineering* 20, 4 (01 Aug 2015), 1095–1125. https://doi.org/10.1007/s10664-014-9316-x

[32] Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code Bad Smells: a review of current knowledge. *J. Softw. Maint. Evol.* 23, 3 (apr 2011), 179–202. https://doi.org/10.1002/smr.521