Scientific Publications

Universal Journal of Computer Sciences and Communications, 2024, 3, 949
www.scipublications.org/journal/index.php/ujcsc
DOI: 10.31586/ujcsc.2024.949

(Sheikh Umar Farooq, 2011)Technical Note

Achieving Maintainability, Readability & Understandability of Software Projects using Code Smell Prediction

Linda Susan Amos [*], **Eng Tirivangani Magadza**

School of Information Science & Technology (SIST)-Harare Institute of Technology, Zimbabwe

*Correspondence: Linda Susan Amos (ladylee.amos@gmail.com)

Abstract: Maintenance of large-scale software is difficult due to large size and high complexity of code.80% of software development is on maintenance and the other 60% is on trying to understand the code. The severity of the code smells must be measured as well as fairness on it because it will help the developers especially in large scale source code projects. Code smell is not a bug in the system as it doesn't prevent the program from functioning but it may increase the risk of software failure or performance slowdown. Therefore, this paper seeks to help developers with early prediction of severity of code smells and test the level of fairness on the predictions especially in large scale source code projects. Data is the collection of facts and observations in terms of events, it is continuously growing, getting denser and more varied by the minute across different disciplines or fields. Hence, Big Data emerged and is evolving rapidly, the various types of data being processed are huge, but no one has ever thought of where this data resides, we therefore noticed this data resides in software's and the codebases of the software's are increasingly growing that is the size of the modules, functionalities, the size of the classes etc. Since data is growing so rapidly it also mean the codebases of software's or code are also growing as well. Therefore, this paper seeks to discuss the 5V's of big data in the context of software code and how to optimize or manage the big code. When we talk of "Big Code for Big Software's," we are referring to the specific challenges and considerations involved in developing, managing, and maintaining of code in large-scale software systems.

Keywords: Severity, Fairness Code Smells, Source Code, Big Code, Big Software Projects

How to cite this paper:
Amos, L. S., & Magadza, E. T. (2024). Achieving Maintainability, Readability &amp; Understandability of Software Projects using Code Smell Prediction. *Universal Journal of Computer Sciences and Communications*, *4*(2), 54–62. Retrieved from https://www.scipublications.com/journal/index.php/ujcsc/article/view/949

Received: March 28, 2024
Revised: May 19, 2024
Accepted: July 8, 2024
Published: July 10, 2024

1. Introduction

"Big Code" is the large-scale software systems that involves a significant amount of code, often consisting of thousands or even millions of lines of code. These codebases are typically complex, with multiple modules, dependencies, and interactions between different components. **"Big Software**" refers to software applications or systems that are extensive and complex in nature. This could include enterprise-level software solutions, operating systems, large-scale web applications, databases, or any software that requires substantial resources and infrastructure to develop, deploy, and maintain. When we talk about "Big Code for Big Software's," we are referring to the specific challenges and considerations involved in developing, managing, and maintaining large-scale software systems (Saeed, 2021) [1]. Some of the key aspects that developers and teams need to address when working with big code and big software include:

Scalability: Designing the software architecture to handle large amounts of data, high user loads, and growing demands over time (Jyothi, 2015) [2]. **Modularity**: Breaking down the codebase into manageable modules or components to improve maintainability, reusability, and collaboration among developers. **Performance**: Optimizing the software to ensure efficient execution and response times, considering factors like algorithmic complexity, data structures, and system resources. **Testing and Quality Assurance**:

Implementing robust testing strategies, including unit testing, integration testing, and performance testing, to ensure the reliability and stability of the software. **Documentation**: Providing comprehensive and up-to-date documentation to aid in understanding and maintaining the codebase, as well as facilitating collaboration between team members. **Version Control and Collaboration**: Utilizing version control systems, such as Git, to manage the codebase, track changes, and enable collaboration among multiple developers or teams. **Deployment and Infrastructure**: Considering the deployment strategies, infrastructure requirements, and scalability options to ensure smooth deployment and operation of the software in production environments. These are just a few considerations when dealing with large-scale software systems. The specific challenges and approaches may vary depending on the nature of the software and the technologies involved (Bachmann, 2005) [3].

The complexity of software is growing fast and software companies are required to continuously update their source code (Lehman, 1980) [4]. The continuous change that occurs to the software often result into the so called *technical debt* and this technical debt can lead into *code smells* (Dario Di Nucci, 2018) [5]. During software maintenance and evolution, software systems need to be continuously changed by developers to (i)implement new requirements (ii)enhance existing features (iii) fix important bugs. Due to time constraints and pressure that most developers normally experience they do not have the time or willingness to keep the complexity of the software under control and find good design solutions before applying modifications. As a consequence, the development activities are often performed in an undisciplined manner, and have the effect to erode the original design of the system by introducing the so called '*technical-debt*'. Code smells is the symptom of the presence of poor design or implementation choices in the source code. Code smells has a quality compromise which when ignored can lead to effects on maintainability, reliability, readability, understandability and testability of the software. Complexity of the software is increasingly growing continuously nowadays due to complex requirements, increased number of modules and code smells in the developed software. Complex requirements are difficult to analyse and understand and due to this reason development becomes difficult and the maintainability of the complex software becomes low. In the software development process, functional and non-functional values both are essential for designers to follow and they guarantee software quality. Functional requirements are only emphasised by developers whereas on functional requirements e.g comprehensibility, verifiability, evolution, maintainability and re-usability are neglected. The lack of non-functional quality leads to decline in the quality of the software and increases the maintenance of the software. This non-functional quality is mostly associated with code smells. Code smell prediction involves using machine learning techniques to automatically identify and predict the severity of the smells using software metrics as features. Fairness mitigation in code smell prediction is the process of identifying and addressing biases and unfairness in the prediction models. This is important to enhance trust and avoiding reinforcement of bias as unfair predictions can reinforce existing biases and stereotypes. Fairness mitigation helps to break this cycle.

2. Research Objectives

To design and develop machine learning model to detect code smells severity in big software programs using software metrics.

To compare & evaluate the level of the severity on the detection based on two different classifiers on different code smell types.

To determine which software metric is most influential to the detection of bad smells.

To determine the level of fairness in the detection of severity.

3. Research Questions

Number	Research Question
RQ1	How can we term the 5'Vs of big data in the context of big code for big Software's
RQ2	How does feature selection method affect the performance of the prediction of code smells?
RQ3	How to determine the level of fairness in code detection model?
RQ4	How can A.I improve the maintenance of software's

4. Methodology

The researcher started by defining the research problem such as code smell detection and fairness mitigation in the software development. Determining the specific aspects of code smells on fairness that we want to focus on. Then literature review followed by conducting a comprehensive study to understand the existing work and approaches related to code smell detection and fairness in software development. Identifying all the gaps, limitations and potential areas for improvement in the existing literature. The next phase was to collect the data for the research work and the data was taken from (Francesca Arcelli Fontana) [6] which consist of four code smell data sets namely god class, feature envy, long methods and data class. On the fairness mitigation phase the researcher wants to develop the strategies or interventions to mitigate fairness issues related to code smells. Propose the techniques or guidelines to promote fairness in code development practices, code reviews or software development processes. For the model development the author followed the research design methodology.

5. Data Set Description

The datasets used in this study are provided by (Francesca Arcelli Fontana) [6], and it is defined as benchmarked data. The datasets belong to different domains and are composed of 76 systems of different sizes written in java. A large set of object-oriented metrics were computed on the 76 systems of the qualities corpus and it covers method, class, package and project level. Some of these metrics were defined according to aspects of software quality like complexity, size and coupling. The code smells have been defined at level of method or class. In method level we have feature envy and long method whereas at class level, we have a data class and god class (As shown in Table 1 and Table 2).

Table 1. Considered Metrics within the Dataset

Size	Complexity	Cohesion	Coupling	Encapsulation	Inheritance
LOC	CYCLO	LCOM5	FANOUT	LAA	DIT
LOCNAMM*	WMC	TCC	ATFD	NOAM	NOI
NOM	WMCNAMM*		FDP	NOPA	NOC
NOPK	AMWNAMM*		RFC		NMO
NOCS	AMW		CBO		NIM
NOMNAMM*	MAXNESTING		CFNAMM*		NOII
NOA	WOC		CINT		
	CLNAMM		CDISP		
	NOP		MaMCL§		
	NOAV		MeMCL§		
	ATLD*		NMCS§		
	NOLV		CC		
			CM		

Table 2. List of Considered Metrics

Quality dimension	Abbreviation	Name of Metric
Complexity	WMC	Weight Method Count
	AMW	Average Method Weight
	CYCLO	Cyclomatic Complexity
	ATED	Access to Foreign Data
	FDB	Foreign Data Providers
	NOP	Number of Parameters
	ATLD	Access
	WMCNAMM	Weight Methods Count of Not Accessor or Mutator Methods
Size	NOPK	Number of Packages
	NOM	Number of Methods
	LOC	Lines of Code
Coupling	CBO	Coupling Between Objects
	FDB	Foreign Data Providers
	NOAM	Number of Accessor Methods
	CM	Changing Method
	ATFD	Access to Foreign Data
	RFC	Response for a Class
	CC	Changing Classes
	CFNAMM	Called Foreign Not Accessor or Mutator Methods
Inheritance	NOI	Number of Interfaces
	NMO	Number of Methods
	NOC	Number of Children
	DIT	Depth of Inheritance Tree
Encapsulation	LAA	Locality of Attribute Access
	NOPA	Number of Public Attribute
	NOI	Number of Interfaces

6. Results and Findings

The main aim of the research study is to develop a machine learning algorithm solution to predict the severity of the software metrics on each code smell type and determine the level of fairness on each prediction using random forest and logistic regression.

This chapter focuses on the discussion of the results obtained during the study and other insights obtained from the research study. The data insights from the discussion are of paramount importance in finding the software metrics that has a positive influence or negative influence on the prediction of a code smell being of certain type for example god class, feature envy, long method or data class.

Random Forest and Logistic Regression models were applied on four bench marked datasets for code smells namely:

- God Class.csv
- Feature Envy.csv
- Long Method.csv
- Data Class.csv

which means four models were developed for the datasets. The research finding will be presented using the following outline.

- Data loading and Preprocessing
- Exploratory Data Analysis
- Feature Engineering
- Model Training
- Model Evaluation
- Model Interpretation
- Fairness evaluation

Data Loading and Preprocessing: The script starts by loading the four datasets named dataset-god-class.csv using pandas. It then performs some preprocessing tasks such as dropping unwanted columns, extracting labels and splitting the data into categorical, integer and float features.

Feature Engineering: The process separates the input features into three types: float, categorical and integer. It iterates over the columns of the training DataFrame and categorizes the columns into the respective types based on their datatypes. Float columns are stored in train_float_features, categorical columns in train_cat_features and integer columns in train_int_features for the god class model. The scripts extracts the target variable or label, which is the severity column from the training data frame. It converts the label into 1-dimensional array using np.ravel()

Model Training: Two models are trained which are the RandomForestClassifier and a Logistic Regression model. The data is split into a training set and a validation set, and the models are trained on the training set.

Model Evaluation: The models are evaluated using the validation set.

Model Interpretation: The models use LIME (Local Interpretable Model Agnostic Explanations) to explain the predictions of the models. It generates explanations for specific instances in the validation set. The training data uses 80% and the test set uses 20% of the data for validation test (As shown in Figure 1, Figure 2, Figure 3, Figure 4, Figure 5, Figure 6, Figure 7 and Figure 8).

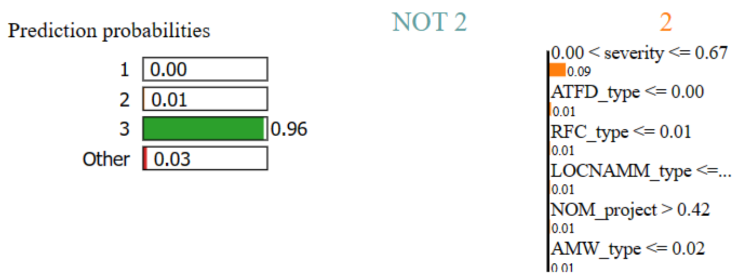

Figure 1. God Class Model 1-Random Forest Classifier

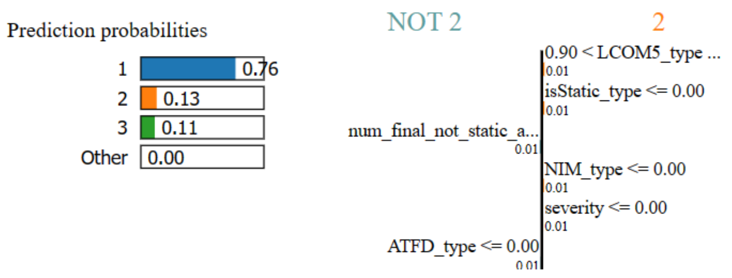

Figure 2. God Class Model 2-Logistic Classifier

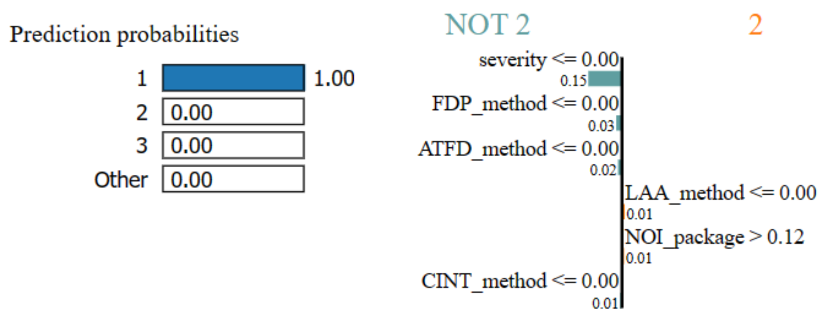

Figure 3. Feature-Envy Model 1-RandomForest Classifier

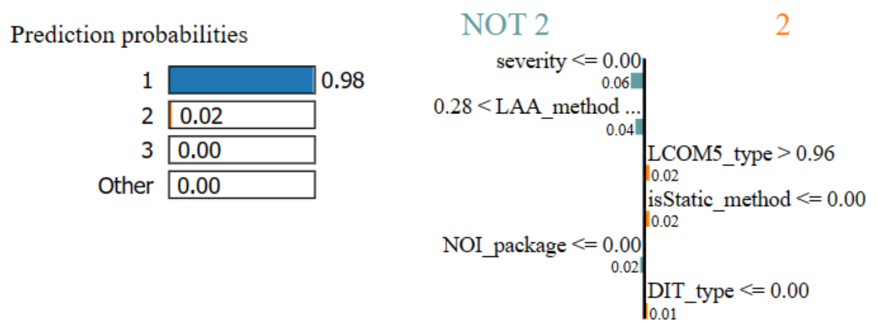

Figure 4. Feature-Envy Model 2-Logistic Regression

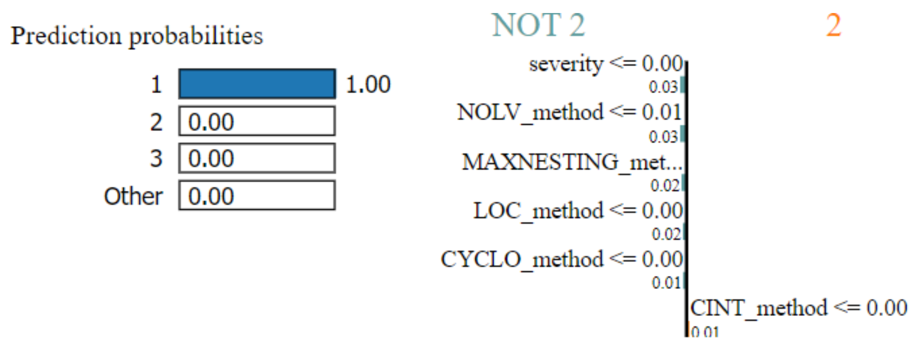

Figure 5. Long Methods Model 1-RandomClassifier

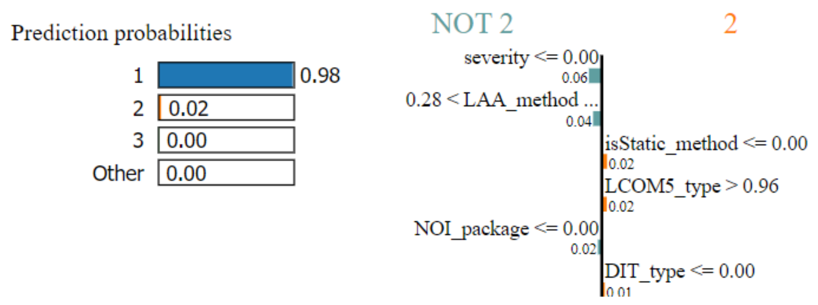

Figure 6. Long Method Model 2-Logistic Regression

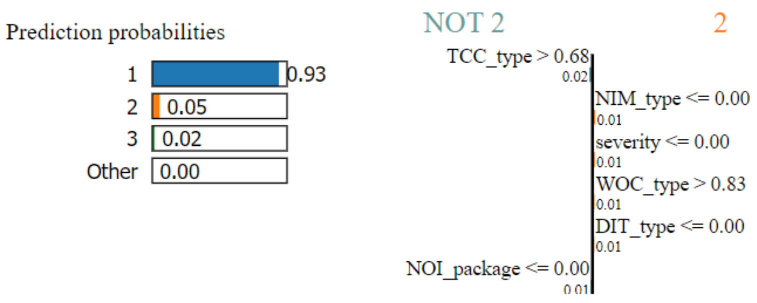

Figure 7. Data Class Model 1-RandomClassifier

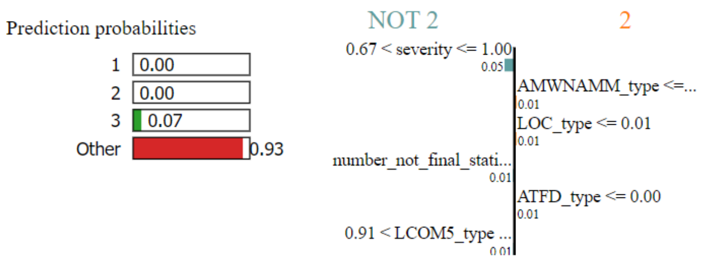

Figure 8. Data Class Model 2-LogisticClassifier

RQ1-How can we term the 5'Vs of big data in the context of big code for big Software's

The five Vs of big data Volume, Velocity, Variety, Veracity, and Value can be adapted and applied to the context of big code for big software projects. The following is how we can term them:

- **Volume:** In the context of big code, Volume refers to the sheer size and scale of the codebase. Big software projects often have extensive codebases with a large number of files, classes, functions, and lines of code. The Volume of the code can present challenges in terms of understanding, maintenance, and scalability.
- **Velocity:** Velocity in the context of big code for big software refers to the speed at which code changes and updates occur. Big software projects typically involve multiple developers working simultaneously on different features or components. The Velocity of code changes can be high, requiring efficient collaboration, version control, and continuous integration practices.
- **Variety:** Variety refers to the different types of code and technologies present in a big software project. Large-scale projects often involve a variety of programming languages, frameworks, libraries, and modules. The Variety of code requires developers to have a diverse skill set and the ability to work with different technologies seamlessly.
- **Veracity:** Veracity in the context of big code relates to the accuracy, reliability, and quality of the codebase. Big software projects demand adherence to coding standards, best practices, and code quality guidelines. Ensuring Veracity involves code reviews, testing, and continuous quality assurance processes to maintain a high level of code integrity.
- **Value:** Value refers to the ultimate benefit or worth that the big software project delivers. A big codebase should contribute to the value proposition of the software by enabling the desired functionalities, performance, and user experience. Value is realized when the software meets the needs of its users, achieves business objectives, and provides a competitive

advantage. Considering these five Vs which are Volume, Velocity, Variety, Veracity, and Value. Developers and project teams can better understand and address the challenges associated with big code in big software projects. This understanding can help guide decisions related to code management, collaboration, quality control, and overall project success.

RQ2- How does feature selection method affect the performance of the prediction of code smells?

Feature selection affects the performance of the predictions by reducing the number of features(dimensions) used for predictions. Feature selection enhances model accuracy, precision, recall and other performance metrics.

RQ3-How to determine the level of fairness in code detection model?

To determine the level of fairness in all the models we used the demographic parity by comparing the positive prediction rates across different demographic groups. The positive predictions rates were calculating the rate of positive predictions for each demographic group. Comparisons were done across the different groups. If the rates are close to 0.0 it means they satisfy demographic parity.

RQ4-How can A.I improve the maintenance of software's?

A.I enables predictive maintenance by analyzing data to predict when components might fail or anomalies occur. Predictive helps schedule maintenance before breakdowns, reducing downtime and costs.

7. Discussion and Conclusion

Software metrics play a crucial role in code smell predictions as they provide valuable information about the structural characteristics of the code. These metrics capture various aspects of the code, such as size, complexity, coupling, cohesion, and other design-related properties. The influence of software metrics on code smell predictions can include identification of smelly code since the software metrics help in identifying potential code smells by quantifying different aspects of the code. For example, metrics like cyclomatic complexity, class size, and method length can indicate the presence of code smells such as complex code or long methods. Feature Importance: Software metrics can be used as features in machine learning models to predict code smells. By analyzing the feature importance, we can understand which specific metrics have a stronger influence on the prediction of code smells. This information helps in prioritizing efforts for code refactoring and maintenance. Threshold Setting: Thresholds for software metrics can be defined to determine the presence or severity of code smells. For instance, a specific metric value exceeding a threshold may indicate the presence of a particular code smell. By defining appropriate thresholds, we can establish guidelines for code quality and identify areas that require improvement. Selection of threshold strongly influence accuracy of the results. Refactoring Guidance: Software metrics provide insights into the areas of code that might benefit from refactoring. For example, metrics like coupling, cohesion, and inheritance depth can guide refactoring efforts to improve code modularity and maintainability, reducing the occurrence of code smells. Continuous Monitoring: Software metrics enable the continuous monitoring of code quality and the detection of potential code smells. By regularly analyzing these metrics, developers and teams can proactively identify and address code smells, reducing technical debt and improving the overall maintainability and scalability of the codebase.

References

[1] Saeed, N. &. (2021). Big data characteristics (V’s) in industry. Iraqi Journal of Industrial Research. 8(1), 1-9.
[2] Jyothi, B. S. (2015). A study on big data modelling techniques. International Journal of Computer Networking, Wireless and Mobile Communications (IJCNWMC), 5(6), 19-26.
[3] Bachmann, F. B. (2005). Designing software architectures to achieve quality attribute requirements. IEE Proceedings-Software,, 152(4), 153-165.
[4] Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68, number 9, 1060-1076.
[5] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik and A. De Lucia. (2018). "Detecting code smells using machine learning techniques: Are we there yet?," 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, pp. 612-621, doi: 10.1109/SANER.2018.8330266.
[6] Francesca Arcelli Fontana, M. V. (n.d.). Comparing and Experimenting Machine Learning Techniques For Code Smell Detection. Empirical Software Engineering.
[7] Sheikh Umar Farooq,Software measurements and metrics, 2011.