International Journal of Electrical and Computer Engineering (IJECE)
Vol. 7, No. 6, December 2017, pp. 3613~3621
ISSN: 2088-8708, DOI: 10.11591/ijece.v7i6.pp3613-3621

Finding Bad Code Smells with Neural Network Models

Dong Kwan Kim
Department of Computer Engineering, Mokpo National Maritime University

Article Info

Article history:

Received May 4, 2017
Revised Jun 28, 2017
Accepted Jul 14, 2017

Keyword:

Code smells
Neural networks
Object-oriented metrics
Software maintenance

ABSTRACT

Code smell refers to any symptom introduced in design or implementation phases in the source code of a program. Such a code smell can potentially cause deeper and serious problems during software maintenance. The existing approaches to detect bad smells use detection rules or standards using a combination of different object-oriented metrics. Although a variety of software detection tools have been developed, they still have limitations and constraints in their capabilities. In this paper, a code smell detection system is presented with the neural network model that delivers the relationship between bad smells and object-oriented metrics by taking a corpus of Java projects as experimental dataset. The most well-known object-oriented metrics are considered to identify the presence of bad smells. The code smell detection system uses the twenty Java projects which are shared by many users in the GitHub repositories. The dataset of these Java projects is partitioned into mutually exclusive training and test sets. The training dataset is used to learn the network model which will predict smelly classes in this study. The optimized network model will be chosen to be evaluated on the test dataset. The experimental results show when the modelis highly trained with more dataset, the prediction outcomes are improved more and more. In addition, the accuracy of the model increases when it performs with higher epochs and many hidden layers.

Corresponding Author:

Dong Kwan Kim,
Department of Computer Engineering,
Mokpo National Maritime University,
91, Haeyangdaehak-ro, Mokpo-si, Jeollanam-do, Korea.
Email: dongkwan@gmail.com

1. INTRODUCTION

Software maintenance is the modification of a software product after delivery to correct faults, to improveperformance or other attributes, or to adapt the product to a modified environment [1]. The activities of software maintenance can be categorized into four types: adaptive, perfective, corrective, and preventive maintenances. Among them, the adaptive maintenance involves reorganizing the structure of the software systems to cope with changes in the software environmentsuch as user requirements and computing technologies. In other words, software changes are one of the inevitable characteristics in the software system. It is hard to estimate the potential consequences of a change after modifying some parts of the software system because even a minor change could increase the complexity of the system and cause unexpected problems. Sometimes such changes can introduce some code smells accidentally in the system. Code smell refers to any symptom introduced in design or implementation phases in the source code of a program. Such a code smell can potentially cause deeper and serious problems during software maintenance. According to the scope of the code smell, the code smells can be classified into three classes: application-level, class-level, and method-level smells. Many software tools have been developed to detect and eliminate code smells using object-oriented metrics and code refactoring methods [2]. The code smell can be an effective indication of whether or notsome parts of the code should be refactored. Although a variety of

software detection tools have been developed, they still have limitations and constraints in their capabilities. In this paper, a code smell detection system is presented with the neural network model that delivers the relationship between bad smells and object-oriented metrics by taking a corpus of Java projects as experimental dataset. The detection system proposed in the paper uses major object-oriented metrics to extract various features for bad code smells by calculating metric values. Many object-oriented metrics have been proposed in the past decades. The well-known metrics such as Line of Code (LOC), Depth of Inheritance Tree (DIT), and Coupling Between Objects (CBO) are applied to assess the quality of object-oriented programs at different levels. While some metrics are applied to assess the whole object-oriented systems, others are mainly used to evaluate single classes or methods. In the previous studies, the object-oriented metrics have been used to detect smelly classes by supporting a prediction model for the code smells. The automated detection tools calculate metric values over an inspected software system to identify specific characteristics of bad smells. For example, Response for aclass (RFC) and Coupling Between Objects (CBO) metrics are used to evaluate the degree of coupling between classes. Cohesion can be measured by Lack ofCohesion in Methods (LCOM) and Loose Class Cohesion (LCC).

The rest of this paper is organized as follows. Section 2 describes the proposed code smell detector in detail. Section 3 presents the experimental results by the detection system. Section 4 introduces the related work and Section 5 remarks the conclusions and future work directions.

2. CODE SMELL DETECTOR

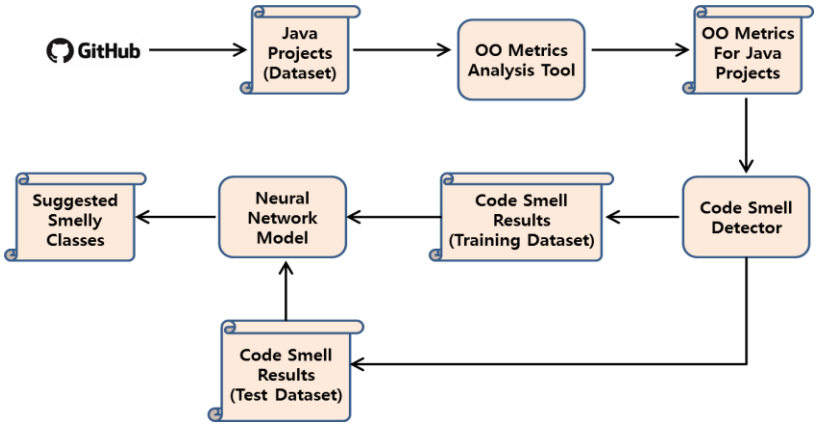

Figure 1. The Overall Workflow of the Proposed Code Smell Detection System

Figure 1 shows the overall workflow of the proposed code smell detection system to find code smells in Java programs.The proposed detection system uses the twenty Java projects which are downloaded from the GitHub repositories. The detection system consists of three main components-OO Metrics Analyzer, Code Smell Detector, and Neural Network. A commercial code analysis tool, *SciTools Understand* is used to analyze the Java projects downloaded from GitHub and to deliver comprehensive metrics values for Java. Code Smell Detector supports different criteria and thresholds of the object-oriented metrics in detecting code smells. Code Smell Detector determines whether or not the classes in the Java projects area smelly class by checking the metrics values of each class. The code smell results are separated into the training and test datasets. The neural network model in the detection system is trained and optimized by the training dataset and then is evaluated by the test dataset of the code smell results.

2.1. Software Metrics and Code Smells

SciTools Understand, a static analysis tool, can extract a wide range of metrics and generate a customizable report for the Java projects. The object-oriented metrics for this research are summarized in the following:

- Line of Code (LOC): Line of code metric is used to count the lines of the source code without considering the comment lines and blank lines. LOC is typical software metric that is used to measure the program size. Many research findings demonstrate that larger LOC values take more time to develop and maintain a program.

- Depth of Inheritance Tree (DIT): The depth of inheritance tree can be defined as the maximum length of a class in the inheritance tree. DIT counts the depth from a specific class to atop-level root class in the hierarchical structure tree. The Java programming allows for only single class inheritance. In general, as a particular class has more super-classes, the class inherits more fields and methods. Since the potential reuse of inherited members can make it more complex a software system, the DIT metric can be one of the criteriato predict the degree of the complexity of an object-oriented system.
- Coupling Between Object classes (CBO): CBOmeasures the number of other classes which a particular class is coupled. We can say that classes C1 and C2 are coupled if class C1 uses a type, data, or method from class C2. By the UML terminologies, classes C1 and C2 have class relationships such as association, aggregation, and composition. Larger CBO values can decrease the modularity and prevent reasonablereuse of a software system. Therefore, we need to keep CBO values to a minimum for the better maintainability.
- Response for a Class (RFC): RFC considers the number of accessible methods and constructors invoked by a class. When an object of that class receives a message, the RFC metric counts the number of distinct methods including inherited methods. The Java programming language determines whether or not other classes can access a particular data or invoke a particular method in a class. The RFC metric is related to the Java access modifiers such as public, package-private, protected, and private. If the RFC value for a class is unacceptably large, it means that class has potential interactions with the rest of a program.
- Weighted Methods per Class (WMC): The WMC metric is the sum of the complexities of all class methods. *SciTools Understand* calculates the sum of cyclomatic complexity of all nested functions or methods in a class, not considering inherited methods. In general, a class with a large WMC value implies that that class can be complex and harder to maintain.
- Number of Children (NOC): The NOC metric measures the number of immediate subclasses i.e., the number of classes one level down the inheritance tree from a target class.The NOC value indicates that the data and methods of a class can be reused in its subclasses. Therefore, if a class has a high NOC value, the class is more responsible in a software system.
- Cyclomatic Complexity (CC): Cyclomatic complexity, as known as McCabe's Cyclomatic Complexity, measures the number of linearly independent paths through a program module. *SciTools Understand* counts the keywords for decision points (FOR, WHILE, etc.) and then adds 1. For a switch statement, each ‘case’ is counted as 1 and the ‘switch’ itself adds oneto the final Cyclomatic Complexity count.
- Lack of cohesion in methods (LCOM): Cohesion refers to the degree of the intra-relationship between the elements in a software module such as packages and classes. It is ideal that each element has a strong relationship in the module byachieving a particular functionality. The LCOM metric indicatesa set of methods in a class is not strongly connected to other methods.

Code smellsrefer to any symptom in the source code of a program that possibly indicates a deeper problem. Bad code smells are usually placed in software systems due to poor design and implementation choices. During design and implementation phases, software developers may ignore the potential problems such as code duplication, unclear code, complicated code, and dead code. Such design and implementation issues are able to make software systems more and more messy over time.

Table 1 show the six code smells which will be considered in this study. These code smellsare major and frequently occurred bad code smells in the object-oriented systems. Some smells such as God Class and Large Class are closely related to the class structures and others are fine-grained code smells in the system like Feature Envy and Long Method. As considering the class levelas well asthe method level, we can improve the accuracy of the proposed neural network model.

Table 1. Code smells considered in the study

Code Smell	Level	Description
God Class	Class	Classes have many members and implement different behaviors
Large Class	Class	Classes do too many tasks with many methods and data members
FeatureEnvy	Class, Method	Methods access the data of another object more than its own data
Parallel Inheritance Hierarchies	Class	We need to create a subclass for another class whenever we create a subclass for a class
Data Class	Class	Classes contain only fields(data) and methods for accessing them
Lazy Class	Class	Classes do not do enough work

2.2. Neural Network Model

Figure 2 depicts a simplified neural network model that is used in the proposed code smell detection system. The network model is implemented in Tensor Flow and consists of three layers-input layer, hidden layer, and output layer. There are the eight object-oriented metrics as input value in the input layer-LOC,

DIT, CBO, RFC, WMC, NOC, CC, and LCOM. The neurons in the different layers are connected with connection weights and biases. Furthermore, the proposed network model supports multiple hidden layers to improve the performance of the model when it predicts the smelly class. In the neural network model, the tanh function is used as activation function for the hidden layers and the softmax function is used as activation function for the output layer.In this study, Adam Optimizer in Tensor Flow is used to optimize the learning rate.

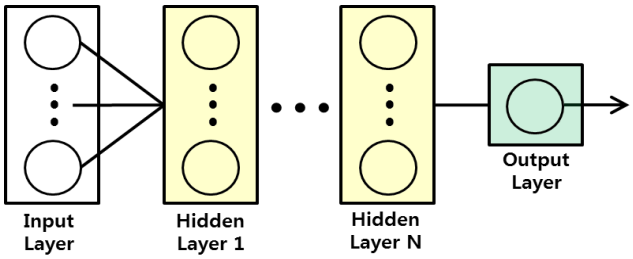

Figure 2. Neural Network Model

3. EVALUATION

Case studies have been conducted to demonstrate the effectiveness and accuracy of the proposed approach for code smell detection with a corpus of massive Java projects. During these case studies, the proposed neural network model has been trained and tested to predict the code smells in the preceding section.

3.1. Java GitHub projects

The Java programming language has continued to evolve with the rapid spread of the Internet since its release in the world. Java also has been used as the primary development language for Android which is one of the most popular mobile platforms. GitHub, a public code repository, stores and manages many Java projects in different application domains. The proposed code smell detection system aims at locating bad smells in the Java projects. While existing research has detected code smellsin relatively small-scale Java projects, the proposed approach uses open large-scale Java programs to increase the reliability and precision of the code smell detection strategies. The GitHub code repositories may include low-quality Java projects, thus requiring the ability to remove the source code for these projects. The fork system provided by the GitHub code repository can be used to effectively extract low-quality Java projects with low fork ratios. A Java project with a low fork frequency is assumed to be a low-quality project.

Table 2. Java GitHub projects

No.	Project Names	# of Classes	LOC	Descriptions
1	Android-Universal-Image-Loader	138	6,820	Android Library
2	bigbluebutton	959	59,099	Web Conferencing
3	bukkit	785	32,277	Mincraft Mod API
4	clojure	754	38,389	Programming Language
5	dropwizard	617	22,687	RESTful web server
6	elasticsearch	7,391	420,834	REST Search Engine
7	junit	1,224	253,79	Testing Framework
8	libgdx	3,224	217,775	Game Dev Framework
9	metrics	401	14,904	Metrics Framework
10	netty	2,674	161,406	Network App Framework
11	nokogiri	154	12,300	HTML/XML/CSS parser
12	okhttp	515	37,264	HTTP & HTTP/2 Client
13	platform_frameworks_base	12,959	1,374,852	Android Base Framework
14	presto	2,841	209,295	Distributed SQL engine
15	retrofit	362	9,098	REST client
16	rxjava	2,109	49,527	Reactive JVM extensions
17	spring-boot	2,666	93,347	App Framework Wrapper
18	spring-framework	11,651	484,224	Application Framework
19	storm	1,205	64,641	Distributed Computation
20	zxing	544	39,873	Barcode image processing
	Total	53,173	3,348,612	

Table 2 shows the GitHub Java corpus [3] with short descriptions that is used to train and test the proposed neural network model. The data set is picked from the top active Java GitHub projects and includes different application domains for diversity. The data set includes 20 projects, more than 53,000 Java classes, and more than 3,000,000 source code lines. Such a corpus statistics tells the data set is big enough to train the proposed model to catch bad smells in code. The whole Java corpus is split uniformly into a training and a test set.

Table 3. Identification Rulesfor Detecting Bad Smells

Bad Smell	Rules
Large Class	LOC > 300 or DIT > 5 or RFC > 20
Lazy Class	RFC == 0 or LOC < 100 or WMC<= 2
Data Class	LCOM>80 or WMC > 50
Parallel Inheritance Hierarchies	DIT > 3 or NOC> 4
God Class	WMC >= 47
Feature Envy	CBO> 5 or LCOM > 50

For the case studies, six design code smells are considered to identify the presence of bad smells in the given Java code. The given Java projects have been analyzed and tested to detect each of the six bad code smells. If a class or a method satisfies the identified thresholds shown in Table 3, it is marked as smelly class or method. Table 3 presents experimental thresholds to apply the objected-oriented metrics. Different identification rules can be defined as following:

- Large Class: Since measuring the size of a class is a simple but essential element, it has been used to evaluate the quality of software systems. The classic and common way of measuring class size is to measure the number of lines of code, i.e. LOC, or the number of attributes and operations. However, to consider only LOC is not enough to detect the large classes. The proposed neural network model will measuremore properties: Depth of Inheritance Tree (DIT) and Response for a Class (RFC). The model usesa threshold of 300 for LOC, a threshold of 5 for DIT and a threshold of 20 for RFCin order to determine if classes are structural characteristics due to the Large Class code smell.
- Lazy Class: Software developers make efforts to maintain classes in an object-oriented system over time. Since understanding and maintaining classes is a time-consuming task, they want to keep meaningful and functional classes in the software system. Extra classes can increase the complexity of the software system. In terms of software maintenance, nearly useless classes can be subject to code refactoring [2]. Lazy Class simply refers to a class that does not do enough and does not earn your attention. LOC is a basic metric to detect the Lazy Class smells. RFC and WMC will be considered for the precise measurement. The proposed code smell detection system tries to find the lazy classes with the following conditions: RFC == 0 or LOC < 100 or WMC <= 2.
- Data Class: A class can be defined as a specification of attributes and permissible operations. Therefore, a class should provide particular services using operations for other classes. However, Data Class is a class that defines variables for managing data without meaningful operations.It does not represent useful and independent behaviors in a software system. The data class is closer to a simple data structure than a responsible class. For the better maintainability, code smell detectors locate the data classes if possible and guide programmers to apply refactoring techniques such as Move Method and Extract Method. The two metrics LCOM and WMC is used to find the data classes with the following conditions: LCOM > 80 or WMC > 50.
- Parallel Inheritance Hierarchies: It is no doubt that the concept of inheritance in object-oriented systems provides many advantages such as code reuse. However, the use of misused inheritance structures not only complicates the software system, but also makessoftware maintenance harder. The parallel inheritance hierarchies smellhappens when we have two parallel inheritance hierarchies associated by composition. Every time we create a subclassof a class, we need to create a subclass for another class due to the structure problem. It is necessary to change the parallel inheritance structure to solve the problems caused by the in heritance hierarchy. In this paper, DIT and NOC are considered to determine if a software system contains the Parallel Inheritance Hierarchies problems with the following conditions: DIT > 3 or NOC > 4.
- God Class: God Class refers to a class that does too much. Many behaviors are centralized in a particular class with unmanageable attributes and operations.To address such problems due to the god class, a common and intuitive technique is to separate the god class into several smaller classes. The object-oriented metric WMC is used to find the God Class smell through the sum of the statistical complexity of all methods in a class. A class is a God Class when WMC of the class is equal to or greater than 47.Even if we can apply other metrics for God Class, WMC is one of the major measures to detect the God Class

smells.In general, God Class can be decomposed in other multiple classes by using code refactoring techniques.

- Feature Envy: After software systemsare released, their code structure can change during operation. Due to the various code refactoring, classes may be integrated or separated with other classes. The structure change of the class can cause its attributesor operationsto be placed in different classes. In this case, a specific operation needs to frequently access some data of another class. This is called Feature Envy, which may increase the complexity of the software system. The identification of Feature Envy in source code can be performed by measuring the strength of coupling that a method has to methods belonging to other classes. Also, this code smell can be detected by measuring the degree of lack of cohesion in methods. The two metrics CBO and LCOM are considered for Feature Envy with the following conditions: CBO > 5 or LCOM > 50.

3.2. Experimental Results and Discussion

The proposed neural network model has been trained and tested with a corpus of the twenty Java projects which contain more than 53,000 Java classes and more than 3,000,000 source code lines. These data has been used in previous studies and do not include a specific application domain, but a variety of domains.In this study, the dataset size is large enough to train the proposed network model and produce more accurate prediction results when the model is used to detect the code smells. Figures 3 through 8 show the experimental results according to the six code smells which are considered in this study. The proposed network model has been evaluated with different iterations when fixing the number of the hidden layers. The number of the epochs in the network model varies 200 through 50,000. It is observed that the accuracy of the network model ranges from 91 % to 99 % according to the number of the epochs. As we can see in the figures, the accuracy of the model is moving upwards when the number of the epochs is increasing.In other words, the accuracy of the model on the test data is proportional to the number of the epochs. As the model is highly trained on the same dataset, the results will be better. However, the type of the code smell does not affect the accuracy of the proposed model. The proposed network model is trained and evaluated with the different number of the hidden layers to demonstrate the effectiveness of the hidden layer. As changing the number of the hidden layers, we can observe how much the hidden layer affects the prediction result of the model. The number of the hidden layers ranges from 1 to 60. The larger the number of the hidden layers is, the better the prediction resultsare. The accuracy of the model reaches up to the 99 % when the number of the hidden layers varies regardless of the type of the code smell. It is observed that when the modelis highly trained with more dataset, the prediction outcomesareimproved more and more. In addition, the accuracy of the model increases when it performs with higher epochs and many hidden layers. The higher value of the accuracy indicates that the proposed network model effectively detects bad code smells in a program.The artificial neural network is frequently used in computer vision and has not been used for addressing software engineering problems. In particular, there is little effortto apply the neural network model into code smell detection with big enough program code.

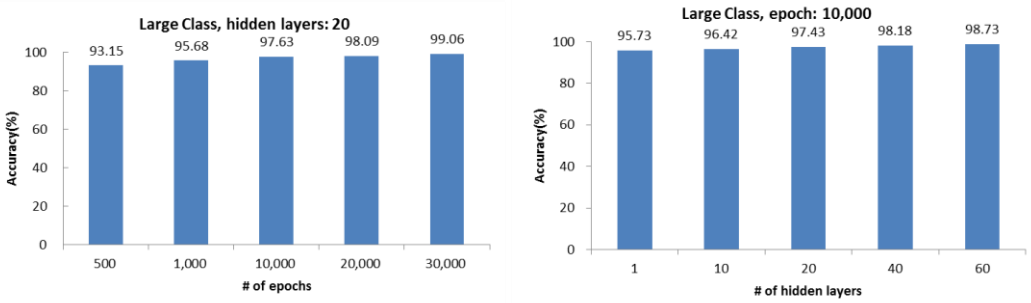

Figure 3. Accuracy of the Network Model when Detecting the Large Class Code Smell

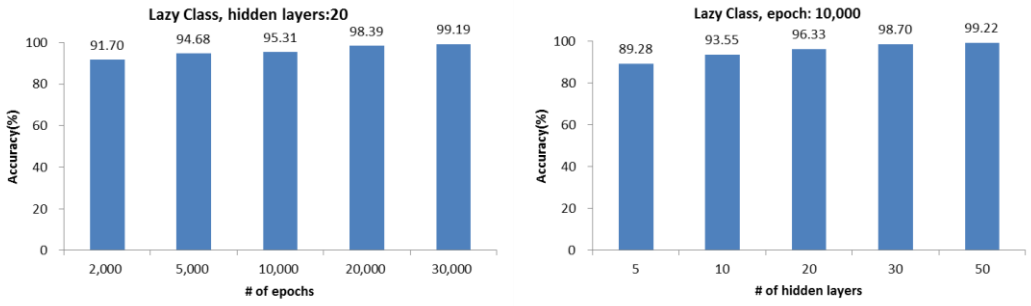

Figure 4. Accuracy of the Network Model when Detecting the Lazy Class Code Smell

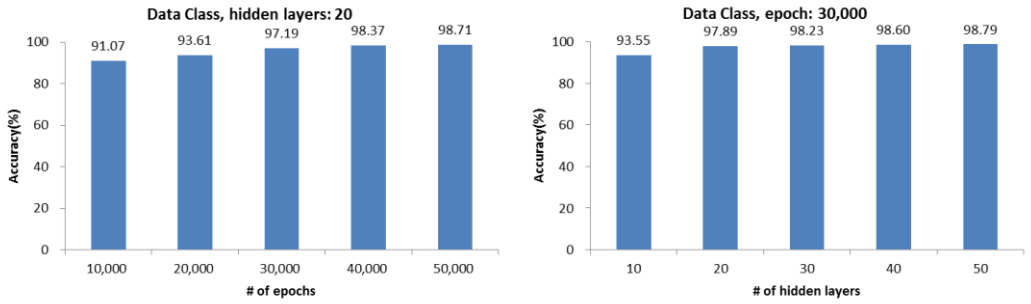

Figure 5. Accuracy of the Network Model when Detecting the Data Class Code Smell

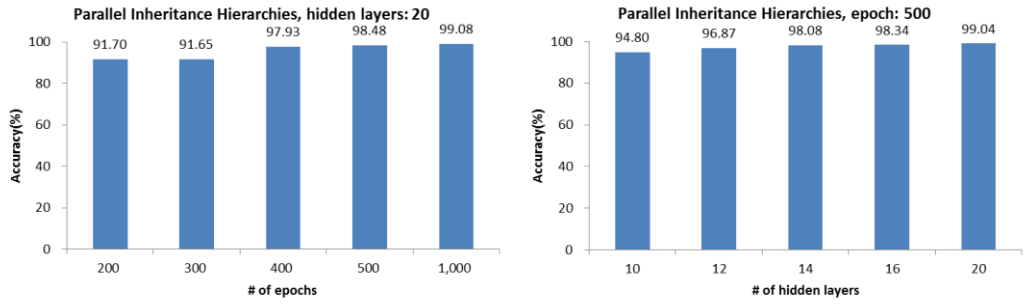

Figure 6. Accuracy of the Network Model when Detecting the Parallel Inheritance Hierarchies Code Smell

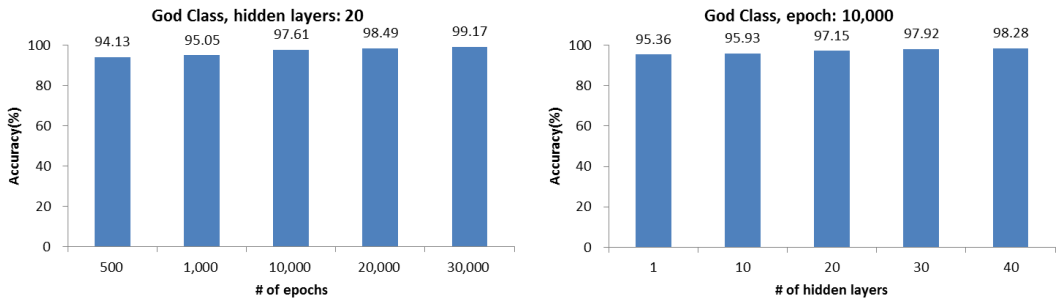

Figure 7. Accuracy of the Network Model when Detecting the God Class Code Smell

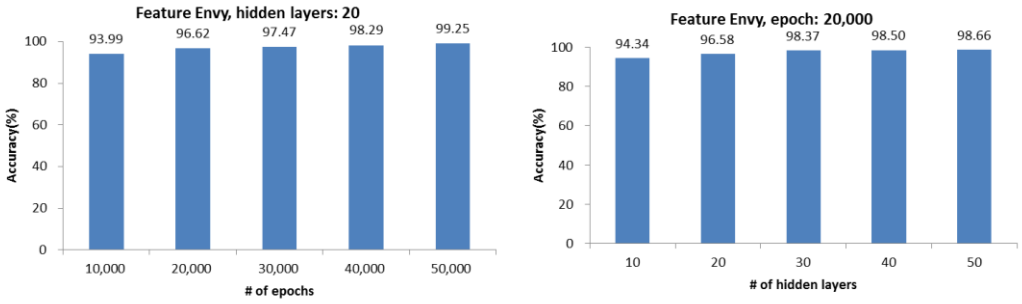

Figure 8. Accuracy of the Network Model when Detecting the Feature Envy Code Smell

4. RELATED WORK

Previous studies have presented methodologies and strategies to detect bad code smells in object-oriented software systems [4, 5]. Most approaches for code smell detection use object-oriented metrics [6, 7] to determine if a software system contains bad smells or not. They have identified empirical thresholds for object-oriented metrics by conducting case studies on object-oriented programs [8].As code size increases, automatic detection tools [9] are needed to help the developer by finding code smells systematically. Automatic software tools [10] have been introduced for visually locating code smells in source code by highlighting suspicious code snippets. They allow the developer to apply code refactoring techniques for identified code smells. Most detection systems or tools try to find primary code smell instances such as Large Class, Lazy Class, Data Class, Parallel Inheritance Hierarchies, God Class, and Feature Envy. The proposed detection system also attempts to find all of them in Java projects. Some studies have dived into evaluating code smell detection tools with their own assessment standards. They studied different code smell detection tools and applied them to find code smells againstapplications.

Neural network models have been used in various domains such as natural language processing, information retrieval, computer vision, and gene prediction [11, 12, 13, 14]. The software engineering community uses the research outcomes of neural networks to addressintractableand practically infeasible problems. In fact, popular models frequently used in computer vision could be applied for programming applications such as statistical program synthesis and code refactoring [15, 16, 17]. We also use the network model to detect and predict bad code smells in object-oriented software systems.Jaspreet kaur and et al. [18] have presented a bad smell detection methodology with a design of neural network model using object-oriented metrics. The neural network model was applied to find twelve bad smells against two different versions of a JavaScript engine called Rhino. For the better accuracy, the same model was trained and tested using different epochs.The experimental results demonstrated the relationship between bad smells and object-oriented metrics. However, their neural network model is too simple to cover much larger Java programs since it has only one hidden layer. In addition, the model was tested against only single Java application. It is a reasonable doubt that their approach mayhave a limitation on the scalability.

5. CONCLUSIONS AND FUTURE WORK

A code smell detection system has been presented to find major bad code smells effectively with an artificial neural network model that has been trained and tested with the twenty popular Java project from GitHub. For the better performance and optimization, the proposed network model has been evaluated on multiple epochs and hidden layers. The presented solution applies the neural network model for code smell detection which is one of the typical software engineering problems by validating the model with massive Java programs. The promising results from the case studies suggest that neural network models can be an essential part of code smell detection tools by identifying refactoring areas with object-oriented metrics. Furthermore, another finding of the case studies is that object-oriented metricshave a close relationship with bad code smells. As a future work, more code smells will be explored to extend the functionality of the proposed detection system with more precise thresholds of the object-oriented metrics. Based on the findings of this study, the code smell detection system will be applied for other object-oriented programming languages such as C++ and C#. The transition to other object-oriented systems may be straight forward because theycan be exposedto the same code smells. The proposed detection system will be improved by using more object-oriented metrics in terms of accuracy and effectiveness.In addition, future research might cover mathematical formalisms to represent the relationship between metrics and code smells.

REFERENCES

[1] Software Engineering Standards Committeeof the IEEE, *IEEE Standard for SoftwareMaintenance*, June 1998.
[2] M. Fowler, *et al*, Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.
[3] M. Allamanis, *et al.*, "Suggesting Accurate Method and Class Names", *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 38-49.
[4] R. Kumar, *et al.*, "An Empirical Study of Bad Smell in Codeon Maintenance Effort", *International Journal of Computer Science Engineering*, vol. 5, 2016.
[5] M. Tufano, *et al.*, "When and Why Your Code Starts to Smell Bad", *Proceedings of the 37th International Conference on Software Engineering*, May 2015.
[6] T.G.S. Filó, *et al.*, "A Catalogue of Thresholds for Object-Oriented Software Metrics", *The First International Conference on Advances and Trends in Software Engineering*, April 2015.
[7] N. Kayarvizhy, "Systematic Review of Object Oriented Metric Tools", *International Journal of Computer Applications*, vol. 135, Feb 2016.
[8] F.A. Fontana, *et al.*, "Automatic metric thresholds derivation forcode smell detection", *International Workshop on Emerging Trends in Software Metrics*, May 2015.
[9] A. Hamid, *et al.*, "A Comparative Study on Code Smell Detection Tools", *Int'l Journal of Advanced Science and Technology*, vol. 60, 2013.
[10] Di Wu, *et al.*, "A metrics-based comparative study onobject-oriented programming languages", *International Conference on Software Engineering and Knowledge Engineering*, July 2015.
[11] Chandrahas Mishra, *et al.*, "Deep Machine Learning and Neural Networks: An Overview", *International Journal of Artificial Intelligence (IJ-AI)*, vol. 6, no. 2, June 2017.
[12] Salisu Musa Borodo, *et al.*, "Big Data Platforms and Techniques", *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 1, no. 1, January 2016.
[13] P. Bielik, *et al.*, "Programming with Big Code: Lessons, Techniques and Applications", *The Inaugural Summiton Advancesin Programming Languages*, 2015
[14] Li Guoping, *et al.*, "Combination of Fault Tree and Neural Networks in Excavator Diagnosis", *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 11, no. 4, April 2013.
[15] P. Bielik, *et al.*, "PHOG: Probabilistic Model for Code", *International Conference on MachineLearning*, 2016.
[16] M. White, *et al.*, "Toward Deep Learning Software Repositories", *Int'l Conference on Mining Software Repositories*, 2015.
[17] H.K. Dam, *et al.*, "A deep language model for software code", *Workshop on Naturalness of Software*, Aug 2016.
[18] J.Kaur, *et al.*, "Neural Network based Refactoring Area Identification in Software System with Object Oriented Metrics", *Indian Journal of Science & Technology*, vol. 9, March 2016.