IJCNN 2019. International Joint Conference on Neural Networks. Budapest, Hungary. 14-19 July 2019

Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder

Mouna Hadj-Kacem
Mir@cl Laboratory, Sfax University
Sfax, Tunisia
mouna.hadjkacem@gmail.com

Nadia Bouassida
Mir@cl Laboratory, Sfax University
Sfax, Tunisia
nadia.bouassida@isimsf.rnu.tn

***Abstract*—Detecting code smells is an important research problem in the software maintenance. It assists the subsequent steps of the refactoring process so as to improve the quality of the software system. However, most of existing approaches have been limited to the use of structural information. There have been few researches to detect code smells using semantic information although its proven effectiveness in many software engineering problems. In addition, they do not capture entirely the semantic embedded in the source code. This paper attempts to fill this gap by proposing a semantic-based approach that detects bad smells which are scattered at different levels of granularity in the source code. To this end, we use an Abstract Syntax Tree with a Variational Auto-Encoder in the detection of three code smells. The code smells are Blob, Feature Envy and Long Method. We have performed our experimental evaluation on nine open-source projects and the results have achieved a considerable overall accuracy. To further evaluate the performance of our approach, we compare our results with a state-of-the-art method on the same publicly available dataset.**

I. Introduction

Because of the significant growth of the software size and complexity, the maintenance costs become more challenging to manage. The continuous changes and the quality constraints are the most important reasons for this high rise in complexity. In such case, an update operation that is intended to enhance or alter a program's functionality could indirectly produce a design problem. In the literature, this situation of poor implementation is known as a code smell [1].

According to Fowler et al. [1], a code smell is defined as 'a surface indication that usually corresponds to a deeper problem in the system'. In other words, it is an indicator of a design problem in the source code. Many researchers have empirically examined the impact of code smells on the quality of the system [2, 3, 4, 5]. They have found that the system becomes more fault-prone and difficult to maintain [6]. Consequently, this influences the quality and causes its deterioration. Fowler et al. [1] have defined the refactoring as an effective solution with low cost to the code smells. By definition, refactoring is a maintenance activity that enhances the software quality. It reconstructs the internal software structure without affecting its external behaviour [1].

Many code smells detection approaches have been proposed in the literature [7, 8, 9, 10, 11, 12, 13, 14]. Most of them rely on the structural analysis [15, 16, 17, 18]. However, the semantic information is less considered despite its proven effectiveness in solving different software engineering problems. Due to this lack of interest, we attempt in this paper to take advantage of the source code's syntax as a first step towards extracting automatically its semantic content. To deal with, we need to apply a specific process that could be insured by means of deep learning algorithms [19].

Recent advances in deep learning [20] have been widely proved in many research fields, including image classification, speech recognition and pattern recognition. Inspired by the promising results reached so far in different software engineering tasks [21, 22, 23], we propose in this paper a semantic-based approach using a deep learning algorithm. Starting from the syntax of the source code, we intend to extract semantic features using a variational auto-encoder [24]. The variational auto-encoder is a generative probabilistic algorithm that is able to encode a latent representation using a Bayesian inference method.

In our approach, the main objective is to extract automatically the semantic features. After parsing the source code with an Abstract Syntax Tree, a transformation is applied to change the obtained trees into a vector representation [25]. Once this step is completed, a variational auto-encoder [24] will be used to generate deep representation that embeds the needed semantic features. Afterwards, these learned semantic features will be fed into a Logistic Regression to determine whether it is a code smell or not.

In the present paper, we focus on three types of code smells belonging to different granularity levels. At class level, we will detect the Blob, while at method level, we will find the Feature Envy and the Long Method. Fowler et al. [1] have defined these code smells as follows:

- Blob is a large class that implements the most of the system's functionalities. For this reason, it is characterized by containing a huge number of attributes and methods that are depending on other classes.
- Feature Envy is a method that is more interested in using the data in classes other than its own. This smell is characterized by a high degree of coupling with the other classes.
- Long Method is a method that dominates the main functionality of the class. Consequently, it is characterized

978-1-7281-2009-6/$31.00 paper N-20433.pdf

by its complexity because it tends to encompass a large number of data.

The evaluation of our approach will be performed empirically on nine open source systems (Apache Xerces, Apache Lucene, Apache Ant, Apache Hive, Apache Pig, Apache Qpid, Apache Ivy, aTunes and Eclipse Core). Based on a public dataset [26], we will compare our method to a prior related work [16]. This latter detects code smells based on textual elements from the source code by defining heuristics to each type.

The main contributions of the present paper are listed as follows:

- We detect code smells based on the semantic information that is extracted from the source code using deep learning algorithm.
- We have conducted our experiments on public available dataset in order to validate its effectiveness and compare its performance against another method.

The rest of this paper is organized as follows. Section II reviews the related work on code smells detection. In Section III, we introduce our proposed detection approach. The performance results of our experiments are reported in Section IV. In Section V, we discuss the threats to validity of our work. Conclusion and future works are given in Section VI.

II. Related Work

According to our three-dimensional taxonomy proposed in [18], the detection approaches are classified by different criteria. For instance, the first dimension in this taxonomy deals with the definition of the abstraction level, the automation level and the used techniques. In the abstraction level, we distinguish between code [7, 9, 13, 14, 27] and model [10, 11] levels.

In this paper, we focus on the second taxonomy's dimension that presents the details of the detection analysis. Specifically, we are interested in the use of the properties types at code level, i.e. structural, historical and semantic properties. We are motivated by the gap found in the literature, where there is an extensive interest devoted to the use of structural information compared to other types, like the historical and particularly the semantic information.

In the following, we will briefly review the detection approaches according to the information types. Then, we will delve into the use of semantic properties, as it is the main source of information in our approach.

A. Structural Information

A majority of existing approaches in the literature rely on the structural information for the detection task. They exploit different metrics that describe the structure of the systems, including the McCabe metrics [28] and CK metrics [29]. Using this type of information, different techniques have been used to detect code smells, where most of them are rule-based, search-based and machine learning-based.

Moha et al. [12] have proposed DECOR (DEtection & CORrection) that automatically generates the detection algorithms. It is a rule-based approach that is able to detect 4 smells, i.e. Blob, Spaghetti Code, Functional Decomposition and Swisss Army Knife. In [30], detection strategies have been defined to formulate metric-based heuristics. The code smells are identified by rules that combine metrics with threshold values.

Sahin et al. [7] have proposed an approach for generating code smells detection rules as a Bi-Level Optimization Problems. They have formulated the detection task as a search-based optimization problem. Their experiments have been conducted on 9 systems to detect 7 types of code smells. Kessentini et al. [31] have conducted a detection approach as a distributed optimization problem where Parallel Evolutionary algorithms were used to detect 8 code smells.

In [8], the authors have presented BDTEX (Bayesian Detection Expert), a semi-automatic tool which is based on the GQM (Goal Question Metric) methodology to build the BBNs (Bayesian Belief Networks) from the definition of anti-patterns. Fontana et al. [9, 13] have applied 16 different machine learning algorithms to detect 4 code smells. They have performed their experimentation over 74 open source systems. In [14], a hybrid detection approach is proposed to detect 4 code smells using deep learning. A deep auto-encoder is trained to extract a reduced feature representation, that subsequently has been learned by an Artificial Neural Network.

B. Historical Information

The historical information resulting from changes occurred to the software projects is also an important source of information to detect code smells. In the literature, some researches have been conducted using this type of information [27, 32, 33].

Palomba et al. [27, 32] have proposed HIST (Historical Information for Smell deTection) that detects 5 code smells via the historical information. This type of information is basically extracted from version control system. The authors have used the association rule mining algorithm followed by heuristics to detect the code smells. Similarly, Fu and Shen [33] presented their approach by exploiting the evolutionary history of projects that is mined from the revision control system. In their approach, three code smells have been detected from 5 projects.

For the approaches that are purely based on historical information, it is worth noting that without enough change history data, some instances of code smells could be missed. This limitation may be performed in the case of newly introduced software projects.

C. Semantic Information

Despite the importance of semantic information, there have been few researches to detect code smells using this type of information. Palomba et al. [16] have introduced TACO (Textual Analysis for Code Smell Detection) that uses textual

TABLE I
CODE SMELLS DETECTION APPROACHES

		Palomba et al. [16]	Liu et al. [17]	Our Approach
Source of the Information		Identifiers + Comments	Identifiers + Code Metrics	AST of the Source Code *
The used Technique		Heuristics	Deep Learning	Deep Learning
Code Smells	Blob	x		x
	Feature Envy	x	x	x
	Long Method	x		x
	Promiscuous Package	x		
	Misplaced Class	x		
Number of Projects		10	7	9

* Without including comments

content from selected parts of the source code to detect code smells. The textual elements needed in their approach are identifiers and comments. After having been extracted, the textual elements receive an Information Retrieval normalization process. Then, for each type of code smell, an heuristic is defined using Latent Semantic Indexing. However, it is difficult to define the perfect heuristic for each type and it becomes harder to generalize for all the code smells in the literature. For the evaluation, TACO is able to detect five code smells at method, class and package levels. It has been evaluated on 10 open source projects based on LandFill dataset [26].

Liu et al. [17] have proposed a detection approach based on deep learning using as input textual features and code metrics. For the textual input, the authors have been limited to the use of one source of information that is the identifier names. They used word2vector together with the Convolutional Neural Network (CNN) to detect the Feature Envy.

Our work differs from the previous mentioned studies in that we will not be restricted to some textual elements, rather we will consider the entire source code because it is semantically richer. As shown in the summary Table I, we will extract the semantic information using deep learning from the source code. It is important to note that the source code does not include comments.

III. PROPOSED APPROACH

The main steps of our approach can be summarized as follows. First, we parse the source code into Abstract Syntax Trees. Afterwards, each tree will be transformed to a vector representation that will be fed into the variational auto-encoder in order to generate the semantic information. Finally, a Logistic Regression classifier is applied to determine whether it is a code smell or not. These steps are illustrated in Figure 1. In the following, we will describe the approach in more detail.

A. Abstract Syntax Tree

The syntax of the program's source code is an important source of semantic information [34]. It can be completely encoded to a code representation. As stated by [25], the representation of the code can be analysed at different granularities.

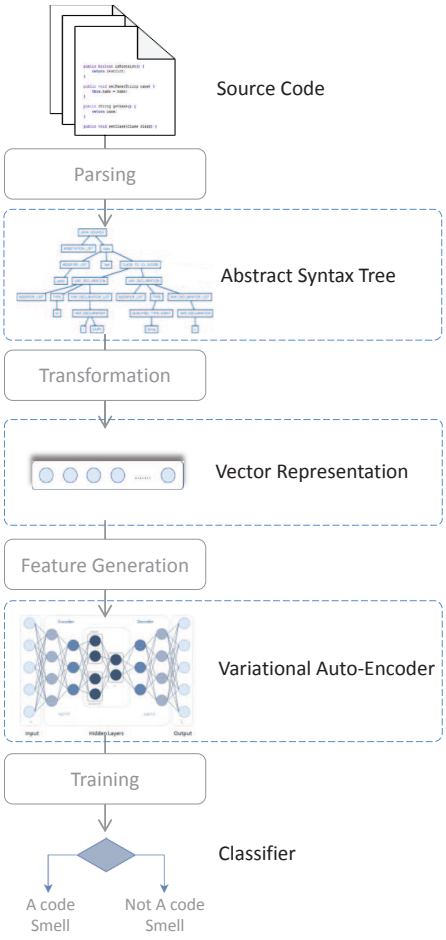


Fig. 1. Overview of the proposed approach

They are defined from fine-to-coarse levels; character-level, token-level, nodes in Abstract Syntax Trees, statement-level and higher.

In this paper, we parse the source code using the Abstract Syntax Tree (AST) as we need the granularity of nodes. This level is the most suitable representation that faithfully translates the syntactic information in the code [25]. Further-

more, it is commonly used in different software engineering tasks [21, 25]. The AST has been used as the intermediate representation that can retain the embedded semantic in the source code [19].

It is important to note that the ASTs are dependent on the programming language specification. In our case, we use the Java AST as our experimental evaluation is performed on Java programs.

B. Vector Representation

Once the ASTs nodes are obtained, they have to be pre-processed. Since the input of the variational auto-encoder must be of integer form, we are required to transform the ASTs nodes into numerical vectors. To deal with this step, we adopted the method proposed by Peng et al. [25] that uses the so-called "coding criterion" in building the program representation. This method serves as a pre-training step in the program analysis. Each AST node is mapped to a numerical vector.

According to this method [25], a non-leaf node p in AST is represented by its n children $\{c_i\}_{1\leq i\leq n}$ denoted as $\{vec(c_i)\}_{1\leq i\leq n}$. It is formulated as:

$$vec(p) \approx tanh\left(\sum_{i=1}^{n} l_i W_i.vec(c_i) + b\right) \quad (1)$$

Then, considering that W_l and W_r are two weight matrices where the number of leaves under $\{c_i\}$ is $l_i = \frac{\#leaves\ in\ c_i}{\#leaves\ in\ p}$, the W_i is defined as follows:

$$W_i = \frac{n-i}{n-1} W_l + \frac{i-1}{n-1} W_r \quad (2)$$

Once the weight W_i of the nodes is calculated, the closeness distance d is measured using the Euclidean distance square, as follows:

$$d = \left\| vec(p) - tanh\left(\sum_{i=1}^{n} l_i W_i.vec(c_i) + b\right)\right\|_2^2 \quad (3)$$

Afterwards, a negative sampling has been applied to minimize the d value.

$$J(d^{(i)}, d_c^{(i)}) = max\left\{0, \Delta + d^{(i)} - d_c^{(i)}\right\} \quad (4)$$

C. Variational Auto-Encoder

A variational auto-encoder [24] is a probabilistic version of a typical auto-encoder. It is a deep generative model that estimates the latent representation using a Bayesian inference method.

As shown in Figure 2, the variational auto-encoder inherits the architecture of a conventional auto-encoder [35]. It consists of encoder and decoder neural networks. The encoder generates a latent representation (z) from the input (x). Then, using the extracted latent vector, the decoder reconstructs the original input data ($\hat{x}$). The joint distribution of the model is defined as follows:

$$p_\theta(x,z) = p_\theta(x \mid z)\, p_\theta(z) \quad (5)$$

Considering that the encoder is $q_\phi(z|x)$ where ϕ denote its parameters (weights and biases). The output of the encoder is an approximation of the mean and variance variables of the Gaussian distribution. By sampling the latent variables with the obtained mean and variance, a latent vector (z) is generated. This latter will be reconstructed through the decoder $p_\theta(x|z)$ where θ are the parameters of this step. The marginal likelihood is:

$$log\, p_\theta(x^{(1)}, ..., x^{(N)}) = \sum_{i=1}^{N} log\, p_\theta(x^{(i)}) \quad (6)$$

This marginal likelihood is the sum of the marginal likelihood of each individual data point. It is defined as follows:

$$log\, p_\theta(x^{(i)}) = D_{KL}(q_\phi(z|x)||p_\theta(z)) + \mathcal{L}(\theta, \phi; x^{(i)}) \quad (7)$$

Where KL is the Kullback-Leibler divergence between the posterior and prior distributions.

Once the semantic features are generated by the variational auto-encoder, a Logistic Regression classifier is used to determine whether it is a code smell or not.

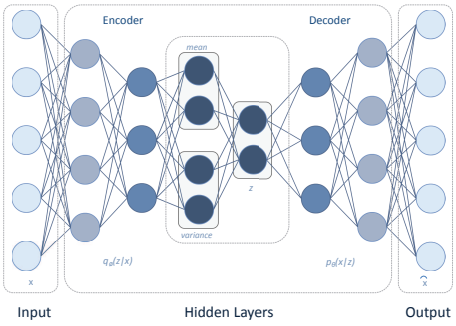


Fig. 2. Illustration of a Variational Auto-Encoder structure

IV. Experiments

In this section, we will evaluate the performance of our approach. We first describe the used dataset and explain the measurement metrics. Then, we analyse our experimental results by studying the posed research questions. To further evaluate our approach, we make a comparison with a prior work.

A. Dataset Description

To evaluate our proposed approach, we have used a publicly available dataset called LandFill [26] as a reference benchmark. LandFill [26] is a web-based platform that contains instances of five bad smells mined from 20 open source systems. It includes two types of bad smells, that are three code smells (i.e. Blob, Feature Envy and Long Method) and two test smells (i.e. General Fixture and Eager Test).

For the purpose of comparability, we have selected nine open-source projects from this dataset. These projects have been used in this previous state-of-the-art work [16]. Additionally, they belong to different scopes and sizes. They are

downloaded from Apache[1], Github[2] and SourceForge[3]. Table II shows the details of the selected projects.

TABLE II
EVALUATED PROJECTS

Project	Version	Classes	KLOC
Apache Xerces	2.3.0	736	201
Apache Lucene	3.6	2246	466
Apache Ant	1.8.0	813	204
Apache Hive	0.9	1115	204
Apache Pig	0.8	922	184
Apache Qpid	0.18	922	193
Apache Ivy	2.1.0	349	58
aTunes	2.0.0	655	106
Eclipse Core	3.6.1	1181	441

B. Evaluation Metrics

To measure the performance of our proposed detection method, three evaluation metrics have been used. These include Precision, Recall, and F-measure. They are defined as follows:

$$precision = \frac{TP}{TP + FP} \tag{8}$$

$$recall = \frac{TP}{TP + FN} \tag{9}$$

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{10}$$

Where TP (True Positive) and TN (True Negative) stand for the number of correctly classified positive and negative instances of code smells. While FP (False Positive) and FN (False Negative) denote, respectively, the number of misclassified positive and negative instances of code smells. Overall, the F-measure is the harmonic mean of the precision and recall metrics, it represents a balance between their values.

C. Experimental Results

Our experiments are designed to answer the following research questions (***RQ***):

- ***RQ1*** : Does the granularity of the code smell impact the accuracy results?
- ***RQ2*** : To what extent does our approach prove its effectiveness when it is compared to other detection approach based on textual content?

In order to train our resulting model, we use k-fold cross-validation technique [36] to divide the training and testing data. In our experiment, we apply 5-fold cross-validation where the dataset is randomly split into 5 partitions. At each time, one partition is used for testing and the other partitions are used for training. Then, the mean value of each evaluation metric is considered.

At class level, Table III shows the detection results of the Blob smell. The precision ranges between 67.01% and 79.2%, the recall ranges between 70.02% and 82.93%. Overall the nine software projects, the average of F-measure is 76.57%. At method level, Table IV and Table V report the accuracy results of, respectively, Feature Envy and Long Method code smells. On average, the F-measures for both of them are 77.03% and 82.97%. As an answer to the ***RQ1***, it is observed from the reported results that the model performs slightly better at method level than at class level. Most probably, this is due to the fact that the dataset does not treat other types of class-level smells.

TABLE III
THE ACCURACY RESULTS OF BLOB

Project	Precision	Recall	F-measure
Apache Xerces	75.79%	81.47%	78.52%
Apache Lucene	78.8%	80.76%	79.76%
Apache Ant	75.52%	82.54%	78.87%
Apache Hive	77.51%	78.88%	78.18%
Apache Pig	67.01%	70.02%	68.48%
Apache Qpid	77.98%	82.93%	80.37%
Apache Ivy	72.66%	74.73%	73.68%
aTunes	69.79%	71.54%	70.65%
Eclipse Core	79.2%	82.17%	80.65%
Average	74.91%	78.33%	76.57%

TABLE IV
THE ACCURACY RESULTS OF FEATURE ENVY

Project	Precision	Recall	F-measure
Apache Xerces	78.42%	81.69%	80.02%
Apache Lucene	79.43%	86.65%	82.88%
Apache Ant	73.65%	85.44%	79.10%
Apache Hive	79.89%	87.76%	83.64%
Apache Pig	72.53%	76.22%	74.32%
Apache Qpid	78.46%	84.51%	81.37%
Apache Ivy	79.59%	85.77%	82.56%
aTunes	62.47%	65.03%	63.72%
Eclipse Core	63.33%	68.11%	65.63%
Average	74.19%	80.13%	77.03%

However, it is not possible to confirm that the granularity does really affect the accuracy because we treated one smell at class level vs. two smells at method level. We will further investigate this issue as a part of our future work by expanding the dataset to include other code smells at class level, i.e. Data Class.

D. Comparison with a state-of-the-art method

In order to answer ***RQ2***, we have compared our approach to a state-of-the-art method [16] that (i) is based on textual elements from the source code, (ii) and uses the same comparative baseline which is LandFill [26]. Table VI illustrates the results of both methods for each code smell and for each software project. We choose to compare the results by using the F-measure metric as it is a harmonic mean between precision and recall measurements.

Figures 3 and 4 show the comparative results for each code smell using the F-measure. As an example, for the Apache Ivy project, we have achieved 73.68% against 80% in the case of

[1]https://www.apache.org
[2]https://github.com
[3]https://sourceforge.net

TABLE V
THE ACCURACY RESULTS OF LONG METHOD

Project	Precision	Recall	F-measure
Apache Xerces	76.51%	83.47%	79.83%
Apache Lucene	84.31%	86.13%	85.21%
Apache Ant	82.73%	85.51%	84.09%
Apache Hive	83.33%	87.65%	85.43%
Apache Pig	82.73%	84.52%	83.61%
Apache Qpid	85.98%	86.98%	86.47%
Apache Ivy	77.36%	81.37%	79.31%
aTunes	75.03%	80.28%	77.56%
Eclipse Core	84.39%	86.09%	85.23%
Average	81.37%	84.66%	82.97%

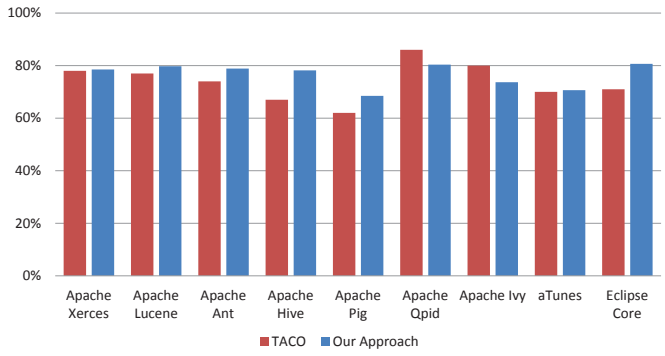


Fig. 3. The comparative F-measure between TACO and our approach at class level (Blob)

Blob. This is interpreted by the fact that the number of Blob instances affects our model because it was very low in this project. Nevertheless, for the same treated code smell, but in a different project, i.e. Eclipse Core, we have reached 80.65% against 71% as the number of smelly instances was higher. Even for the largest project, i.e. Apache Lucene (see Table II), our results outperform the state-of-the-art results by an average of 10.61% for all the types of code smells.

Regardless the granularity levels and across the different evaluated projects, we have achieved considerable overall F-measure. Generally, our performance outperformed the state-of-the-art results by reporting an improvement in 77% of cases. This leads us to conclude that the deep representation learnt from the source code has generated useful semantic information for the detection task. In the literature, there are 22 code smells belonging to different granularities. It would be very hard and tedious to define a perfect heuristic for each code smell, however in our case we do not need to treat distinctly the different types of code smells as we rely on the deep representation learning.

V. THREATS TO VALIDITY

In this section, we discuss the threats that may affect the validity of our performance results.

- **Internal Validity** is related to the consistency of the obtained results. The main threat to internal validity of our experiment concerns the public dataset [26]. As it is labelled manually, it is possible that it could suffer from an inherent subjectivity in the definition of code smells. However, we considered it because it is publicly available and it suits our purpose to conduct a less biased comparison that evaluates our approach to a prior work.
- **External Validity** refers to the relevance of the results and their generalization. With regards to the programming language, we conduct our experiments on Java software projects. Consequently, we cannot claim generalizability of our approach to projects belonging to different programming languages.
  Another threat to external validity arises from the choice of the dataset availability in our evaluation. In our experiment, we have used only the open source systems. So, we do not know if it is applicable in an industrial context. Additional research is required to investigate this issue.

VI. CONCLUSION

In this paper, we have proposed a code smells detection approach based on deep learning. A variational auto-encoder is implemented to generate a deep representation that captures the semantic information hidden in the source code.

As a benchmark for our experimentation, we have used the LandFill dataset to evaluate the accuracy of the code smells detection. In summary, we have reached considerable experimental results. Our findings have highlighted the effectiveness of the generative algorithm, i.e. variational auto-encoder, in improving the results of the code smells detection. Also, we have demonstrated that our method can outperform the results of previous related work.

In our future work, we plan to extend our approach by including the comments with the source code in the learning of the deep representation. Our hypothesis supposes that the textual information in the comments will enrich the semantic embedded in the source code. Considering that the comment's syntax differs with the source code's syntax, we will pre-train it otherwise. In this way, our performance results may be more improved. Additionally, it would be interesting to apply our method to other programming languages.

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

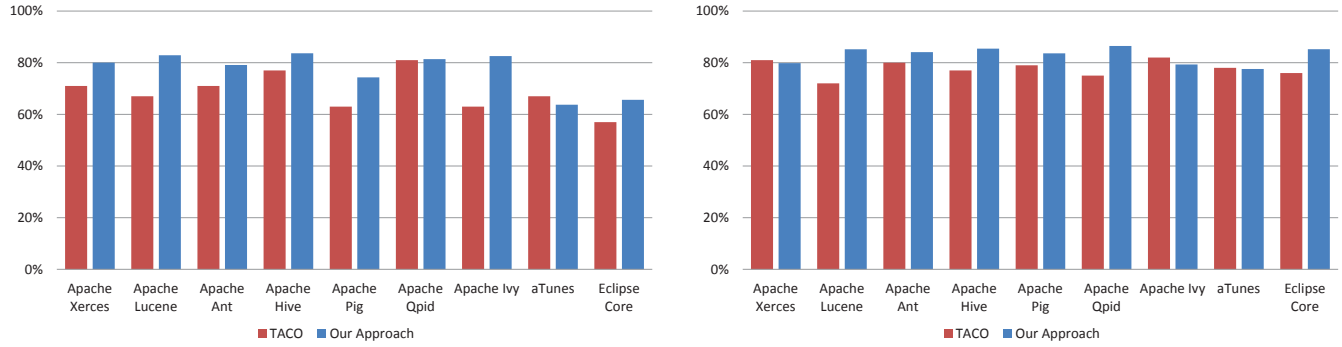
[2] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, Jun 2012.

[3] Z. Soh, A. Yamashita, F. Khomh, and Y. G. Guhneuc, "Do Code Smells Impact the Effort of Different Maintenance Programming Activities?" in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1, March 2016, pp. 393–402.

[4] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When

TABLE VI
COMPARISON RESULTS WITH THE STATE-OF-THE-ART METHOD

Project	Approach	Blob	Feature Envy	Long Method
Apache Xerces	Palomba et al.	78%	71%	81%
	Our Approach	78.52%	80.02%	79.83%
Apache Lucene	Palomba et al.	77%	67%	72%
	Our Approach	79.76%	82.88%	85.21%
Apache Ant	Palomba et al.	74%	71%	80%
	Our Approach	78.87%	79.10%	84.09%
Apache Hive	Palomba et al.	67%	77%	77%
	Our Approach	78.18%	83.64%	85.43%
Apache Pig	Palomba et al.	62%	63%	79%
	Our Approach	68.48%	74.32%	83.61%
Apache Qpid	Palomba et al.	86%	81%	75%
	Our Approach	80.37%	81.37%	86.47%
Apache Ivy	Palomba et al.	80%	63%	82%
	Our Approach	73.68%	82.56%	79.31%
aTunes	Palomba et al.	70%	67%	78%
	Our Approach	70.65%	63.72%	77.56%
Eclipse Core	Palomba et al.	71%	57%	76%
	Our Approach	80.65%	65.63%	85.23%



(a) Feature Envy

(b) Long Method

Fig. 4. The comparative F-measure between TACO and our approach at method level

and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. IEEE Press, 2015, pp. 403–414.

[5] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[6] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb 2004.

[7] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 6:1–6:44, Oct. 2014.

[8] F. Khomh, S. Vaucher, G. Yann-Gaël, and H. Sahraoui, "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559 – 572, 2011.

[9] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.

[10] G. Saranya, H. K. Nehemiah, A. Kannan, and V. Nithya, "Model level code smell detection using egapso based on similarity measures," vol. 57, no. 3, 2018, pp. 1631 – 1642.

[11] A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, vol. 24, no. 4, pp. 947–965, Dec 2016.

[12] N. Moha, Y. G. Gueheneuc, L. Duchien, and A. F. L. Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, Jan 2010.

[13] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, Jun 2016.

[14] M. Hadj-Kacem and N. Bouassida, "A Hybrid Approach To Detect Code Smells using Deep Learning," in *Proceedings of the 13th International Conference on Eval-*

uation of Novel Approaches to Software Engineering. SciTePress, 2018, pp. 137–146.

[15] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[16] F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for Smell Detection," in *IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[17] H. Liu, Z. Xu, and Y. Zou, "Deep Learning Based Feature Envy Detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 385–396.

[18] M. Hadj-Kacem and N. Bouassida, "Towards a Taxonomy of Bad Smells Detection Approaches," in *Proceedings of the 13th International Conference on Software Technologies*. SciTePress, 2018, pp. 164–175.

[19] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 334–345.

[20] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[21] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 297–308.

[22] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning*, 2014, pp. 649–657.

[23] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 763–773.

[24] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[25] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Knowledge Science, Engineering and Management*. Springer International Publishing, 2015, pp. 547–553.

[26] F. Palomba, D. D. Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Landfill: An Open Dataset of Code Smells with Public Evaluation," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 482–485.

[27] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Mining Version Histories for Detecting Code Smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, May 2015.

[28] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[29] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[30] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, Sep. 2004, pp. 350–359.

[31] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[32] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 268–278.

[33] S. Fu and B. Shen, "Code bad smell detection through evolutionary data mining," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct 2015, pp. 1–9.

[34] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[35] Y. Bengio *et al.*, "Learning deep architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[36] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann Publishers Inc., 2011.