A Hybrid Approach To Detect Code Smells using Deep Learning

Mouna Hadj-Kacem and Nadia Bouassida
Mir@cl Laboratory, Sfax University, Tunisia

Keywords: Hybrid Approach, Deep Learning, Auto-encoder, Artificial Neural Networks, Code Smell Detection.

Abstract: The detection of code smells is a fundamental prerequisite for guiding the subsequent steps in the refactoring process. The more the detection results are accurate, the more the performance of the refactoring on the software is improved. Given its influential role in the software maintenance, this challenging research topic has so far attracted an increasing interest. However, the lack of consensus about the definition of code smells in the literature has led to a considerable diversity of the existing results. To reduce the confusion associated with this lack of consensus, there is a real need to achieve a deep and consistent representation of the code smells. Recently, the advance of deep learning has demonstrated an undeniable contribution in many research fields including the pattern recognition issues. In this paper, we propose a hybrid detection approach based on deep Auto-encoder and Artificial Neural Network algorithms. Four code smells (God Class, Data Class, Feature Envy and Long Method) are the focus of our experiment on four adopted datasets that are extracted from 74 open source systems. The values of recall and precision measurements have demonstrated high accuracy results.

1 INTRODUCTION

Due to the ever-increasing software complexity, the maintenance has become arduous, time-consuming and hence more costly. More than two-third of the total project budget is dedicated to the maintenance activities, mainly because of the continuous changes (Erlikh, 2000; April and Abran, 2012). Very often, these changes could be improperly treated by developers who may, unwittingly, commit violations of software design principles. In such case, possible design problems may appear in the software. These problems are known in the literature as code smells or by other designations like anti-patterns (Brown et al., 1998), anomalies, bad smells, design flaws, design defects, etc. According to (Fowler et al., 1999), a code smell is defined as 'a surface indication that usually corresponds to a deeper problem in the system'.

Many research studies have empirically investigated the impact of code smells on the software quality (Khomh et al., 2012; Soh et al., 2016). They have pointed out that the code smells make the software more fault-prone and difficult to maintain, which subsequently leads to the deterioration of its quality. In order to mitigate the problems caused by the code smells, a refactoring process should be followed (Fowler et al., 1999). By definition, refactoring is a popular maintenance activity, devoted to enhance the software quality with economical costs (Mens and Tourwe, 2004). It reconstructs the internal software structure without affecting its external behaviour (Fowler et al., 1999). According to (Mens and Tourwe, 2004), the refactoring process consists of three consecutive steps: (i) detection of code smells, (ii) identification of the adequate refactoring operations and (iii) evaluation of the preservation of the software quality.

The earlier the code smells are detected, the less will be the cost of refactoring and the better the software quality will be. Therefore, the detection step plays a decisive role in improving the results of the other steps and thereby on the performance of the software refactoring. As a result, several approaches have been proposed to deal with this research topic by using different techniques. However, the lack of consensus regarding the definition of code smells in research studies results in great difficulties in interpreting or making meaningful comparisons of detection results. This was explicitly stated by (Mäntylä and Lassenius, 2006), who confirmed that the conflicting perceptions of developers conduct to subjective code smell interpretations, which in turn lead to the variances in the performance evaluation. In this context, the use of machine learning techniques has been considered as a suitable way to deal with the confusion surrounding this lack of consensus. On the one hand,

Hadj-Kacem, M. and Bouassida, N.
A Hybrid Approach To Detect Code Smells using Deep Learning.
DOI: 10.5220/0006709801370146
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 137-146
ISBN: 978-989-758-300-1

it can provide additional objectivity to face this issue by reducing the developers' subjective cognitive interpretations (Arcelli Fontana et al., 2016). On the other hand, it proves more efficiency when dealing with large-scale systems and more ability to interact with newly acquired data.

Recently, deep learning (LeCun et al., 2015), an emerging branch of machine learning, has spread widely over many research fields including the pattern recognition issues. The architectures of deep learning techniques are based on multilayer neural networks that are intended to effectively model high-dimensional data. According to (Bengio et al., 2013), the deep learning algorithms aim to provide multiple levels of abstraction of the data ranging from low to high levels. They are different from conventional machine learning techniques that are built on shallow structure.

In this paper, we propose a hybrid learning approach to detect four object-oriented code smells. Our approach consists of an unsupervised learning phase followed by a supervised learning phase. In the first phase, we perform a dimensionality reduction of the input feature space by using a deep auto-encoder (Hinton and Salakhutdinov, 2006) that extracts the most relevant features. The output of this phase is a reduced representation of newly extracted features that is used as a basis for the supervised classification in the second phase. The selected algorithm for this task is the Artificial Neural Networks (ANN) classifier.

In our study, the four selected code smells are God Class, Data Class, Feature Envy and Long Method. The two first ones belong to the class-level and the two latter ones belong to the method-level. According to (Fowler et al., 1999), their corresponding definitions are as follows:

- God Class is a large class that dominates a great part of the main system behaviour by implementing almost all the system functionalities. It is distinguished by its complexity and by encompassing a high number of instance variables and methods.
- Data Class is a class that is merely composed of data without complex functionalities. This class is free of responsibility; it is designated to be manipulated by other classes.
- Feature Envy is a method-level smell characterized by its excessive use of variables and/or operations belonging to other classes more than using its own ones. Thus, this method tends to make so many calls to use the data of the other classes.
- Long Method refers to a large method in terms of its size, which dominates the implementation of several functionalities.

To evaluate the performance of our approach, we conducted our experiments on four adopted datasets (Arcelli Fontana et al., 2016) that are generated from 74 open source systems. Furthermore, to evaluate the effectiveness of our approach, we conduct a comparison between our hybrid detection method and the basic classifier. The reported experimental results underline the importance of reducing the dimensionality in improving the accuracy of the detection.

In summary, the main contributions of this paper are two-fold:

- We propose the first hybrid detection approach based on combining two learning techniques (deep auto-encoder and ANN) for the detection of code smells.
- We show that the detection performance can be significantly improved when exploiting a reduced number of extracted features by means of a deep learning algorithm.

The rest of this paper is organized as follows. Section 2 overviews the related work dealing specifically with the machine learning-based detection approaches. Section 3 describes the proposed approach. The experimental setup and results are described in Section 4. Section 5 discusses the threats to validity of our study. Finally, Section 6 concludes the paper and outlines future work.

2 RELATED WORK

Several approaches have been proposed to detect code smells. Based on the used techniques, we divide these approaches into five broad categories: rule-based (Moha et al., 2010a), search-based (Sahin et al., 2014), visualization-based (Murphy-Hill and Black, 2010), logic-based (Stoianov and Şora, 2010) and machine learning-based approaches (Kreimer, 2005; Hassaine et al., 2010; Oliveto et al., 2010; Khomh et al., 2011; Maiga et al., 2012a; Fu and Shen, 2015; Palomba et al., 2015; Arcelli Fontana et al., 2016).

Since our approach belongs to the latter category, we focus the related work only on the approaches falling in the partially or fully machine learning-based.

(Kreimer, 2005) proposed a detection approach based on decision trees. The selected technique was applied to detect two design flaws that are Long Method and Large Class. The experimental result was based on two projects that are IYC system and WEKA package.

Table 1: Machine learning-based detection approaches.

	Learning Nature			ML Algorithm	Number of Systems	Features Type
	Supervised	Unsupervised	Hybrid			
(Kreimer, 2005)	x			Decision Tree	2	Structural
(Khomh et al., 2009)	x			Bayesian Belief Networks (BBNs)	2	Structural + Lexical (From the rule cards (Moha et al., 2010b))
(Hassaine et al., 2010)	x			Artificial Immune System Algorithm	2	Structural
(Oliveto et al., 2010)	x			B-Splines	2	Structural
(Khomh et al., 2011)	x			BBNs based on the Goal Question Metric (GQM)	2	Structural
(Maiga et al., 2012b)	x			Support Vector Machines (SVM)	3	Structural
(Maiga et al., 2012a)	x			Support Vector Machines (SVM)	3	Structural (Renforced with the feedback of users)
(Palomba et al., 2013)		x		Association Rule Mining	8	Historical
(Fu and Shen, 2015)		x		Association Rule Mining	5	Structural + Historical
(Palomba et al., 2015)		x		Association Rule Mining	20	Historical
(Arcelli Fontana et al., 2016)	x			16 Algorithms*	74	Structural
Our Approach			x	Auto-Encoder + Artificial Neural Networks (ANN)	4 Adopted Datasets**	Structural + Generated Features

*J48 (with pruned, unpruned and reduced error pruning), JRip, Random Forest, Naïve Bayes, SMO (with Radial Basis Function and Polynomial kernels) and LibSVM (with the two algorithms *C*-SVC and ν-SVC in combination with Linear, Polynomial, RBF and Sigmoid kernels)
** See Section 4.1

(Khomh et al., 2009) proposed an extension of the DECOR (DEtection & CORrection) approach (Moha et al., 2010b) in order to support uncertainty in the detection of smells. In this work, the authors transformed the specifications which are in form of rule cards into BBNs (Bayesian Belief Networks). However, the expressiveness of the rule cards was limited and their composition can lead to the emergence of many intermediate nodes in the BBNs. As a solution, the authors (Khomh et al., 2011) suggested BDTEX (Bayesian Detection Expert) that is based on the GQM (Goal Question Metric) methodology to extract information from the anti-pattern definition. Thus, the BBNs can be systematically built without relying on the rule cards. The experiment was conducted on GanttProject and Xerces to detect the occurrences of Blob, Functional Decomposition and the Spaghetti Code.

(Hassaine et al., 2010) have drawn a parallel between the human's immune system and the detection approach. They have applied the artificial immune systems algorithms to identify the occurrences of Blob, Functional Decomposition and the Spaghetti Code on two open source systems that are GanttProject and Xerces.

(Oliveto et al., 2010) proposed an approach, called ABS (Anti-pattern identification using B-Splines). The signature of the anti-pattern is learned and is performed through an interpolation curve that is generated by a set of metrics and their values. Then, based on the distance between this signature and the signature of a given class, the similarity value is deducted. Thus, the higher the similarity value, the higher the likelihood that the given class is affected. ABS has been tested with Blob on two medium size Java systems.

(Maiga et al., 2012b) proposed a support vector machine-based approach to detect the occurrences of Blob, Functional Decomposition and the Spaghetti Code. Later, the authors (Maiga et al., 2012a) extended their previous work by suggesting an approach called SMURF, which takes into account the feedback of practitioners. Both approaches were experimented on the same three open source systems which are ArgoUML, Azureus and Xerces.

(Palomba et al., 2013; Palomba et al., 2015) proposed an approach named HIST (Historical Information for Smell deTection) to detect five types of bad smells. Only the historical information extracted from version control systems was used by the association rule mining algorithm. Then, they defined heuristics which were applied to identify each one of the considered code smells.

Similarly, (Fu and Shen, 2015) proposed a detection approach by mining the evolutionary history of projects extracted from revision control system. Three code smells are chosen to be detected from 5 projects, whose the duration of the evolutionary history vary from 5 to 13 years. The two latter works used historical properties. The common limitation

shared by them is the lack of availability of versions from a given system in order to have historical changes that are fed into the association rule mining algorithm.

Recently, (Arcelli Fontana et al., 2016) conducted an experimentation of 16 machine learning algorithms on four code smells that are Data Class, Large Class, Feature Envy and Long Method. The selected algorithms are as follows: J48 (with pruned, unpruned and reduced error pruning), JRip, Random Forest, Naïve Bayes, SMO (with Radial Basis Function and Polynomial kernels) and LibSVM (with the two algorithms *C*-SVC and ν-SVC in combination with Linear, Polynomial, RBF and Sigmoid kernels). Additionally, these algorithms have been combined with a boosting technique. The experiment is based on a large heterogeneous repository composed of 74 software systems belonging to the Qualitas Corpus (Tempero et al., 2010).

In contrast to the previous approaches, we exploit in this paper two learning algorithms for the tasks of dimensionality reduction and classification. Also, the type of the trained features is the extracted ones from the deep auto-encoder because they can significantly help the learner to reduce the computational overhead and yield high accuracy results. More details about the previous works and ours are tabulated in Table 1. They are listed according to the nature of learning (i.e., supervised, unsupervised or hybrid), the machine learning (ML) algorithm, the number of systems and the type of the used features. For the datasets, we adopt them from (Arcelli Fontana et al., 2016) because their oracle was built by means of advisors and raters (see Section 4.1), in difference to other works that are applied on fully manually constructed oracles.

3 PROPOSED APPROACH

In this study, we propose a hybrid learning-based approach consisting of two main phases. The first phase performs an unsupervised learning procedure based on the use of a deep auto-encoder. The auto-encoder is applied on an unlabelled data in order to reduce its dimensionality and to extract a new feature representation. Then, based on the output of the first phase, the second phase targets a supervised learning classification by using an Artificial Neural Networks (ANN).

The structure of the proposed approach is outlined in Figure 1. More details of the approach are given in the next subsections.

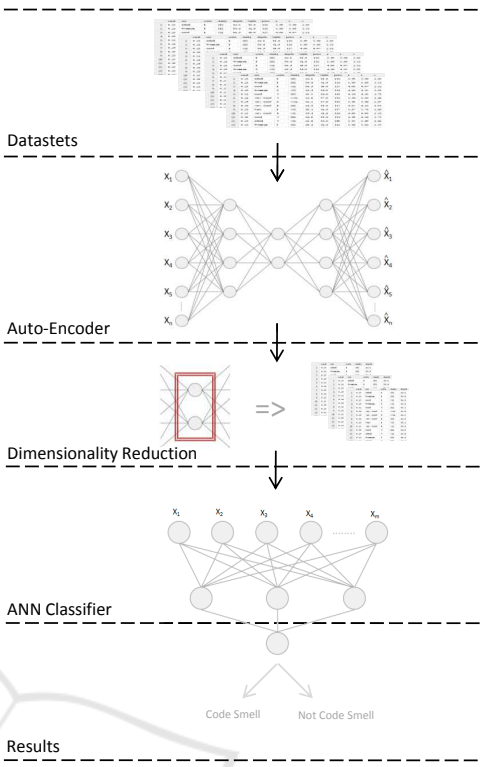


Figure 1: Overview of the proposed approach.

3.1 Phase 1: Unsupervised Feature Learning

The presence of a large number of features may contain redundancy and noise that can affect the performance of a learner and subsequently increase the complexity of the generated model. To tackle these problems, a dimensionality reduction should be applied. Given an input of unlabelled data, a feature extraction is applied to perform the needed transformations that generate the most significant features (Han et al., 2011). As a consequence, the generated low-dimensional data will be easily classified and also makes it possible for the classifier to improve its performance results.

An auto-encoder (Bengio et al., 2009) is an unsupervised neural network that is trained with feedforward and backpropagation algorithms. It mainly aims at the reduction of its original input and reproducing it as an equivalent reconstructed output. The structure of the auto-encoder consists of an input and output layers of equal sizes, and one or more hidden layers. It is composed of an encoder $f : x \rightarrow h$ that maps the input to the hidden layer h, and a decoder $g : h \rightarrow \widehat{x}$ that maps back to the original input. The principle of the encoding procedure is as follows:
Given an input data $\{x(1), x(2), ..., x(n)\}$, the encoder

network compresses these n inputs into a lower dimensional feature subspace. The hidden layer is given by:

$$h = f(x) = tanh(Wx + b_h) \quad (1)$$

where W is the weight and b_h is the bias of the hidden layer. Afterwards, the decoder network reproduces the output $\{\widehat{x}(1), \widehat{x}(2), ..., \widehat{x}(n)\}$ that is computed as follows:

$$\widehat{x} = g(h) = tanh(W^T h + b_{\widehat{x}}) \quad (2)$$

where $b_{\widehat{x}}$ is the bias of the output. The auto-encoder is trained to minimize the reconstruction error between the input and the output:

$$Err = \sqrt{\sum_{i=1}^{n} (x_i - \widehat{x}_i)^2} \quad (3)$$

Figure 2 illustrates an example of an auto-encoder containing 3 hidden layers that are respectively composed of 4, 2, 4 neurons.

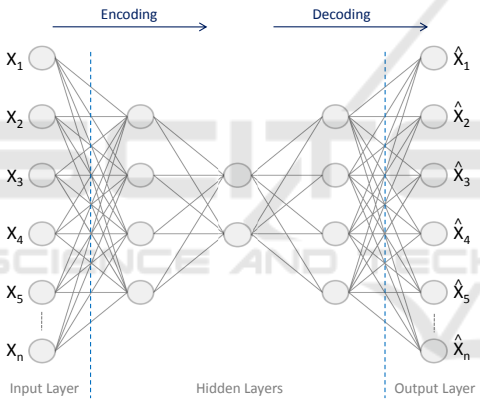


Figure 2: An auto-encoder structure.

3.2 Phase 2: Supervised Classification

In this phase, the new extracted feature space will be combined with a feature vector from the original dataset. This feature vector is the response class that indicates if the sample is affected or not by one of the studied code smells. Thus, we obtain a reduced labelled training set to be fed as an input of the selected classifier.

There exists a variety of supervised learning algorithms, like the probabilistic classifiers (i.e., BBNs), classification rules (i.e., JRip), decision trees (i.e., Random Forest) and neural networks (i.e., ANN). In our study, we select a supervised classifier belonging to the family of neural networks that is the Artificial Neural Networks (ANN). The ANN consist of input, hidden and output layers, in which the neurons are connected, and for each connection, weights are set. As it is a binary classification, the output neuron will decide whether it is a code smell or not, according to its type.

4 EXPERIMENTS

In this section, we describe the used datasets as well as the experimental settings. Then, the results are evaluated according to the standard measures of performance.

4.1 Datasets

As stated before, we experiment our approach using the datasets proposed in (Arcelli Fontana et al., 2016). In brief, the authors selected 74 open source systems from Qualitas Corpus (Tempero et al., 2010). The systems are heterogeneous; they belong to different application domains and vary in their sizes. Then, the metrics are computed using the DFMC4J (Design Features and Metrics for Java) tool (Ferme, 2013). Afterward, the authors achieved the labelling by means of advisors which are existing code smell detection tools (iPlasma (Marinescu et al., 2005), PMD[1], Fluid Tool (Nongpong, 2012), AntiPattern Scanner (Wieman, 2011)) and rules (Marinescu, 2005), followed by a manual validation made by 3 raters. Once the labelling process is done, a balanced dataset is generated for each type of code smells. Each dataset contains 1/3 smelly samples and the rest 2/3 are non smelly samples.

As shown in Table 2, the two first datasets concern the code smells at class level where the number of features is 62. While at method level, the number of features rises because of the difference in granularity where there is more fine-grained features. The list of all the used features is shown in Figure 3. The detailed definitions of these features are available in the appendix in (Arcelli Fontana et al., 2016).

Table 2: Datasets description.

	Dataset	Code Smell	# Samples	# Features
Class level	DS1	God Class	420	62
	DS2	Data Class	420	62
Method level	DS3	Feature Envy	420	83
	DS4	Long Method	420	83

[1]https://pmd.github.io/

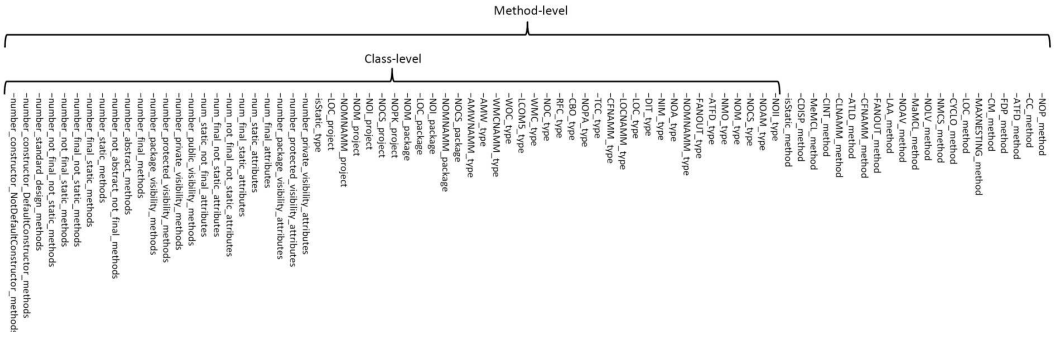


Figure 3: Original features at class and method levels.

4.2 Experimental Setup

Our experiments are performed in the R programming environment (R Core Team, 2014).

In order to tune the parameters of the auto-encoder, we use the metric of mean squared error (MSE) to minimize the rate of the reconstruction of the input. The chosen auto-encoder in this experiment is a five-layers network composed of input and output layers with three hidden ones. The activation function is hyperbolic tangent (tanh). The rest of the selected parameters are summarized in Table 3. For the ANN classifier, the parameter optimization was made by the use of a grid search approach.

Table 3: The parameters of the auto-encoder.

Parameters	Values
Activation function	Hyperbolic Tangent (Tanh)
Number of hidden layers	3
Neurons in each hidden layer	20 . 10 . 20
Number of epochs	200

4.3 Experimental Results

The experimental results are first discussed, then are followed by a comparison with a conventional detection.

4.3.1 Performance Measurements

The performance of our approach is evaluated based on the precision, recall and F-measure metrics. They are defined as follows:

- Precision: the percentage of correctly identified code smells by the number of detected code smell candidates.

$$precision = \frac{TP}{TP + FP} \quad (4)$$

- Recall: the percentage of correctly identified code smells by the total number of actual code smells.

$$recall = \frac{TP}{TP + FN} \quad (5)$$

- F-measure: the harmonic mean of the precision and recall metrics, it represents a balance between their values.

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (6)$$

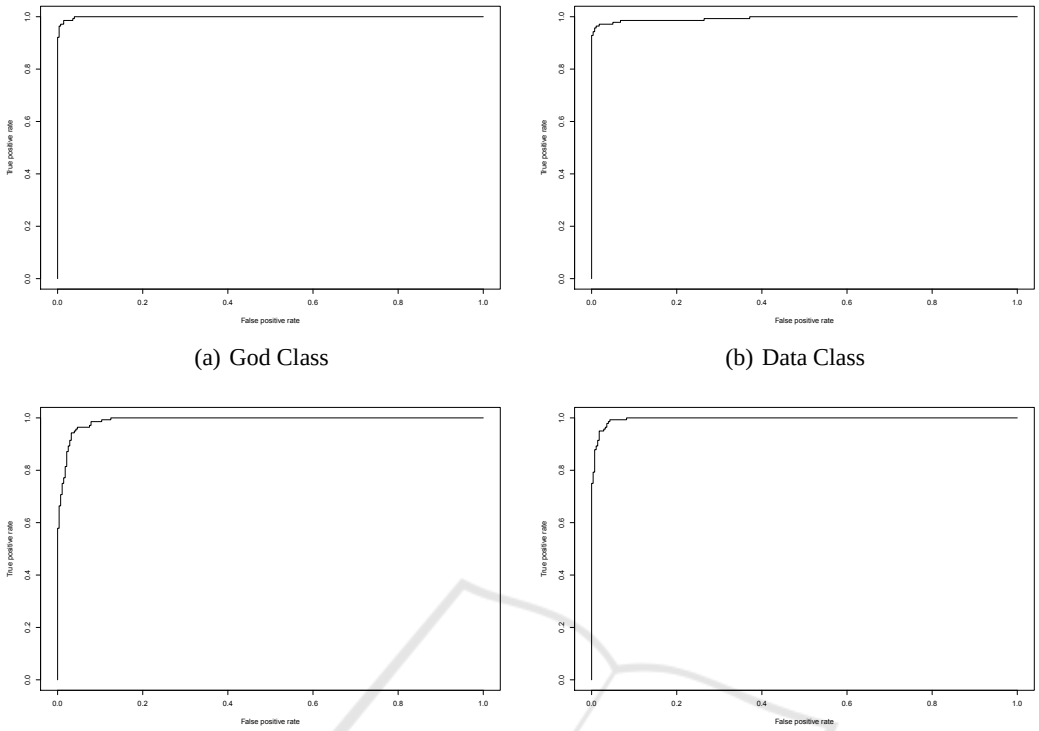
Where TP (True Positive) indicates the number of the correctly detected code smells. TN (True Negative) is the number of non code smells that are correctly classified. While FP (False Positive) and FN (False Negative) correspond to incorrectly classified code smells and non code smells respectively.

4.3.2 Performance Evaluation

To evaluate the performance of the model, we use 10-fold cross-validation (Witten et al., 2011) where the data is randomly split into 10 subsets. Each time one subset is used for testing and the other nine subsets are used for training. Once the 10 subsets are reported, the mean of each performance measurements is considered.

Overall, the values of the precision and the recall have demonstrated high accuracy results. As shown in Table 4, the best F-measure value is of 98.93%, it is reached with the code smell God Class. Even at method level, the F-measure surpasses 96%. In summary, the results were very close which means the effectiveness of the proposed approach at both abstraction levels.

To further evaluate our approach, we plot the ROC (Receiver Operating Characteristic) curves for each code smell. The curve is obtained by means of plotting the TPR (True Positive Rate) against the FPR (False Positive Rate) for different threshold values. Figure 4 shows four ROC curves that correspond to



(a) God Class (b) Data Class

(c) Feature Envy (d) Long Method

Figure 4: The ROC curves of (a) God Class, (b) Data Class, (c) Feature Envy and (d) Long Method.

Table 4: The performance results.

Code Smell	Precision	Recall	F-measure
God Class	99.28%	98.58%	98.93%
Data Class	98.92%	98.22%	98.57%
Feature Envy	96.78%	96.09%	96.44%
Long Method	96.78%	97.83%	97.30%

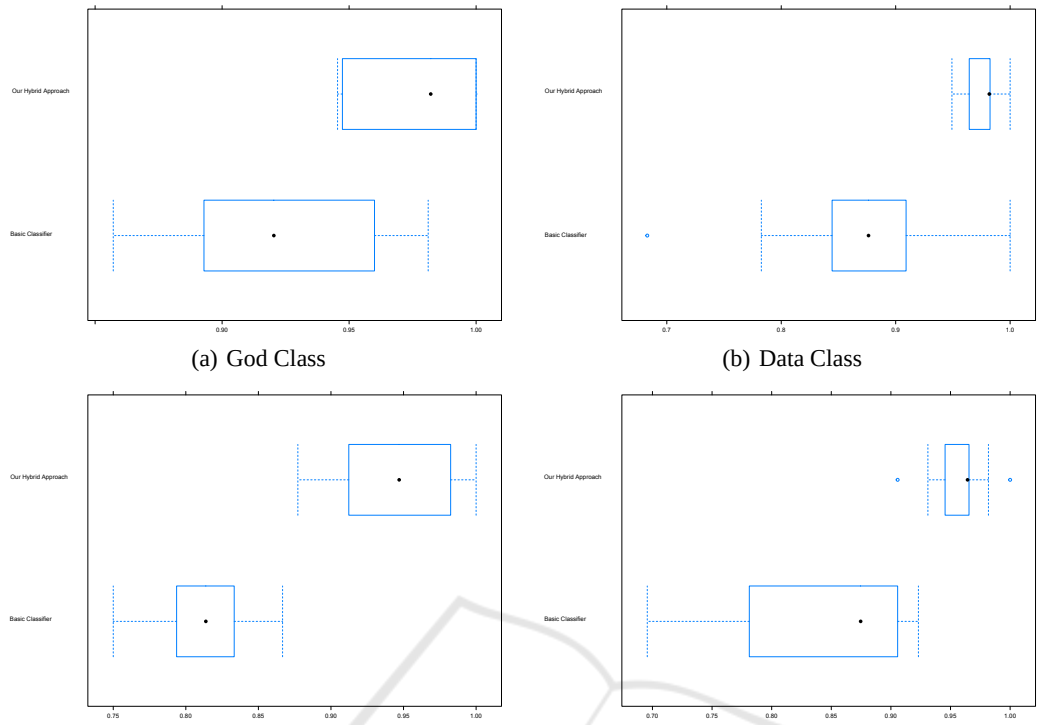
the four code smells. The area under the ROC curve is a sign of the performance of the classifier. The larger area under the ROC curve is, the better the classifier's performance will be. In parallel with the results depicted in Table 4, the ROC of God Class outperforms the ROC of the other code smells.

Generally, the values are very encouraging and indicate the efficiency of dimensionality reduction by means of the auto-encoder in improving the detection performance of the classifier.

4.3.3 Comparison Between our Hybrid Approach and the Basic Classifier

To evaluate the effectiveness of our approach, we conduct a comparison between our hybrid detection method and a detection based solely on the basic classifier. The latter detection is the followed strategy applied in previous works where the original features are directly fed into the basic classifier that is the ANN in our case. For more consistent evaluation, we use the boxplots to compare side by side the results of both approaches by means of the median values and even the distribution of values between the lower and upper quartiles.

The F-measure is the selected metric for the comparison as it provides a balance between the precision and recall metrics. As depicted in Figure 5, our results are better than those of the basic classifier. It is significantly observed from the distribution of values across the quartiles that the detection with the auto-encoder outperforms the basic classifier which is trained without dimensionality reduction. Another difference resides in the tuning of the ANN. When applying a conventional detection, the tuning of the ANN parameters becomes more complex and time-consuming because of the high number of features (that is 62 at class level and 83 at method level). Whereas the tuning of the ANN parameters in our approach is faster and less complex with the extracted features.



(a) God Class (b) Data Class

(c) Feature Envy (d) Long Method

Figure 5: Comparison of F-measure between our hybrid approach and the basic classifier of (a) God Class, (b) Data Class, (c) Feature Envy and (d) Long Method.

5 THREATS TO VALIDITY

There is a number of potential threats to validity that could influence the results of our study.

- *Internal validity* refers to whether there is a causal relationship between the experiment and the obtained results. In our study, this threat can be related to the adopted datasets (Arcelli Fontana et al., 2016). As we have mentioned above, the datasets are partially built manually. The manual part is not performed by experienced developers. Potentially, this may affect, to some extent, the experimental results. However, since there is an automatic part in the construction of the datasets that is performed by previous detection tools and rules, this can alleviate the actual threat.
- *Conclusion validity* concerns the relationship between the treatment and the results of the experiment. It relates to the analysis of the experimental results, i.e., the implementation of the treatment and the measurement performance. This threat is treated by performing a comparison to another scenario of detection that is similar to the strategy of the prior works, where the detection is based solely on the basic classifier with the original features.
- *External validity* refers to the ability to generalize the findings obtained from the experiment. Actually, we use generated datasets from only open source systems. Thus, we cannot generalize our findings to industrial projects. Additional study is required to resolve this issue.

6 CONCLUSION

In this paper, we propose a hybrid approach that applies both unsupervised and supervised algorithms to detect four code smells. The first phase performs a dimensionality reduction by using a deep auto-encoder that extracts the most relevant features. Once the feature space is reduced with a small reconstruction error, the ANN classifier learns the new generated data and outputs the final results. Our approach is validated on four adopted datasets that are generated from a large number of open source systems. The experimental results confirm that the dimensionality reduction prior to classification plays an important role in

improving the detection accuracy. Additionally, we observed that the hybrid approach is able to outperform the basic classifier algorithm because it is based on the most relevant features that are extracted with the deep auto-encoder.

Our future direction focuses on exploring other types of features in order to underpin the results of our detection approach. The features that we intend to add, are fine-grained and cover the detection of other types of code smells. In addition, we plan to apply other deep learning algorithms and compare between them. For this reason, we will expand the data because deep learning outperforms better results with richer dataset.

REFERENCES

April, A. and Abran, A. (2012). *Software maintenance management: evaluation and continuous improvement*, volume 67. John Wiley & Sons.

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

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828.

Bengio, Y. et al. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127.

Brown, W. H., Malveau, R. C., McCormick, H. W., and Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons.

Erlikh, L. (2000). Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3):17–23.

Ferme, V. (2013). JCodeOdor: A software quality advisor through design flaws detection. *Master's thesis, University of Milano-Bicocca, Milano, Italy*.

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

Fu, S. and Shen, B. (2015). Code Bad Smell Detection through Evolutionary Data Mining. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–9.

Han, J., Pei, J., and Kamber, M. (2011). *Data mining: concepts and techniques*. Elsevier.

Hassaine, S., Khomh, F., Gueheneuc, Y. G., and Hamel, S. (2010). IDS: An Immune-Inspired Approach for the Detection of Software Design Smells. In *Seventh International Conference on the Quality of Information and Communications Technology*, pages 343–348.

Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507.

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

Khomh, F., Vaucher, S., Guhneuc, Y. G., and Sahraoui, H. (2009). A Bayesian Approach for the Detection of Code and Design Smells. In *Ninth International Conference on Quality Software*, pages 305–314.

Khomh, F., Vaucher, S., Guhneuc, Y.-G., and Sahraoui, H. (2011). Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559 – 572. The Ninth International Conference on Quality Software.

Kreimer, J. (2005). Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science*, 141(4):117 – 136. Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Maiga, A., Ali, N., Bhattacharya, N., Saban, A., Guhneuc, Y. G., and Aimeur, E. (2012a). SMURF: A SVM-based Incremental Anti-pattern Detection Approach. In *19th Working Conference on Reverse Engineering*, pages 466–475.

Maiga, A., Ali, N., Bhattacharya, N., Saban, A., Guhneuc, Y. G., Antoniol, G., and Ameur, E. (2012b). Support vector machines for anti-pattern detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 278–281.

Mäntylä, M. V. and Lassenius, C. (2006). Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431.

Marinescu, C., Marinescu, R., Florin Mihancea, P., Ratiu, D., and Wettel, R. (2005). iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. In *Proceedings of International Conference on Software Maintenance*, pages 77–80.

Marinescu, R. (2005). Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance*, pages 701–704.

Mens, T. and Tourwe, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Moha, N., Gueheneuc, Y. G., Duchien, L., and Meur, A. F. L. (2010a). DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Moha, N., Guéhéneuc, Y.-G., Meur, A.-F. L., Duchien, L., and Tiberghien, A. (2010b). From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, 22(3):345–361.

Murphy-Hill, E. and Black, A. P. (2010). An Interactive Ambient Visualization for Code Smells. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 5–14, New York, NY, USA. ACM.

Nongpong, K. (2012). *Integrating" code smells" detection*

with refactoring tool support. PhD thesis, The University of Wisconsin-Milwaukee.

Oliveto, R., Khomh, F., Antoniol, G., and Gueheneuc, Y. G. (2010). Numerical Signatures of Antipatterns: An Approach Based on B-Splines. In *14th European Conference on Software Maintenance and Reengineering*, pages 248–251.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., and Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *28th IEEE/ACM International Conference on Automated Software Engineering*, pages 268–278.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Poshyvanyk, D., and Lucia, A. D. (2015). Mining Version Histories for Detecting Code Smells. *IEEE Transactions on Software Engineering*, 41(5):462–489.

R Core Team (2014). R: A Language and Environment for Statistical Computing.

Sahin, D., Kessentini, M., Bechikh, S., and Deb, K. (2014). Code-Smell Detection As a Bilevel Problem. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(1):6:1–6:44.

Soh, Z., Yamashita, A., Khomh, F., and Guhneuc, Y. G. (2016). Do Code Smells Impact the Effort of Different Maintenance Programming Activities? In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 1, pages 393–402.

Stoianov, A. and Şora, I. (2010). Detecting patterns and antipatterns in software using Prolog rules. In *International Joint Conference on Computational Cybernetics and Technical Informatics*, pages 253–258.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Asia Pacific Software Engineering Conference*, pages 336–345.

Wieman, R. (2011). Anti-Pattern Scanner: An Approach to Detect Anti-Patterns and Design Violations.

Witten, I. H., Frank, E., and Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition.