Detecting Code Smells using Deep Learning

Ananta Kumar Das[1], Shikhar Yadav[2], Subhasish Dhal[3]
[1]*Department of Software Engineering*
International Institute of Information Technology Bangalore, India
[2,3]*Department of Computer Science and Engineering*
Indian Institute of Information Technology Guwahati, India

***Abstract*—A smell in software refers to a symptom introduced in software artifacts such as architecture, design or code. A code smell can potentially cause deeper and serious problems, while dealing with mainly non-functional requirements such as testability, maintainability, extensibility and scalability. The detection of code smell is an essential step in the refactoring process, which facilitates non functional requirements in a software. The existing approaches for detecting code smells use detection rules or standards using a combination of different object-oriented metrics. Although a variety of code smell detection tools have been developed, they still have limitations and constraints in their capabilities. The most well-known object-oriented metrics are considered to identify the presence of smells in software. This paper proposes a deep learning based approach to detect two code smells (Brain Class and Brain Method). The proposed system uses thirty open source Java projects, which are shared by many users in GitHub repositories. The dataset of these Java projects is partitioned into mutually exclusive training and test sets. Our experiments have demonstrated high accuracy results for both the code smells.**

***Index Terms*—Code smells, Deep Learning, Object-oriented metrics, Code smell detection tools.**

I. Introduction

A characteristic in software design or implementation that may indicate a deeper problem such as violation of fundamental principles in architecture, design or code. Code smells are neither bug nor technically incorrect events and do not prevent the program from its expected functioning. However, they may later become weaknesses, which may slow down the development speed or increase the risk of bugs or failures in the future. Code smells can be an indicator of factors that contribute to technical debt [15].

Software quality attributes are important factor for successful software products. Code smells are bad programming practices. They may produce software of poor qualities and hamper reusability, and maintainability. Software systems that have such symptoms are prone to develop bugs over time, increasing risks in the maintenance activities and therefore contributing to a troublesome and costly maintenance process. Such situations often translate into higher fault rates and more bugs filed by users and developers. For the over volume of software with high complexity, the maintenance may become arduous, time-consuming and hence more costly. Usually, more than 65% cost of the total project budget is used in maintenance phase, mainly because of the continuous modification and enhancements [8]. Very often, these changes, done by developers could impact software quality attributes due to violating software design and coding principles. In such cases, possible deeper problems may appear in software artifacts, mainly in design and code. These problems are known in the literature as code smells or by other designations like anti-patterns, bad smells, design flaws, design defects, etc [9].

Refactoring is a technique, which reconstructs the internal structure of a system without altering the external behavior [7]. It is a popular maintenance activity, devoted to cost effective enhancement of software qualities. The refactoring process consists of three consecutive steps: (i) detection of code smells, (ii) identification of the refactoring tasks needed and (iii) reevaluation of the software quality attributes [4]. This process is a must to prevent the problems caused by the code smells [7].

Detection of code smell at early stage can make the refactoring economic in terms of cost and time. Also, a manual approach is tedious and takes more time to identify code smells when the size of software is more. Many approaches [4]–[6] have been proposed to detect the variety of code smells. Detailed literature review and analysis on code smell detection have been made in [6], [10]. Most of the existing code smell detection approaches are based on manual predictions relying on software metrics [6], [10], [11]. However, it is challenging to manually construct the optimal heuristics. Outcome on such approaches [6] also suggested that different metrics and different heuristics for the same code smells could very well be available and result in low agreement between different detectors [12]. To avoid manually designed heuristics, statistical machine learning techniques, like SVM, Naive Bayes, and LDA, are employed to build the complex mapping between code metrics (as well as lexical similarity) and predictions [12], [13]. However, empirical study in [12] suggested that such statistical machine learning based smell detection approaches have critical limitations that deserve further research.

Well known machine learning methods have been applied to detect source code smells; however, the neural network based approaches are new in software engineering. Deep learning, a subset of machine learning that allows computational models composed of multiple hidden processing layers to learn representations of data with multiple levels of abstraction. [16], [17]. With significant advances in deep learning techniques, they have been successfully used in different domains, e.g.,

978-1-7281-1895-6/19/$31.00

text processing, natural language processing, speech recognition, image, audio and video [14] etc. However, code smell detection based on deep learning techniques have not been largely explored. That is the reason why we employ deep learning techniques in this paper to build a neural network based classifier that classifies methods in subject applications into 'smelly' and 'non-smelly'. In this paper, we propose a deep learning based approach for detecting two code smells, namely, Brain Class and Brain Method [18]. In best of our knowledge, for the first time, these code smells have been studied using deep learning approach in this paper. The rest of the paper is organized as follows. Section II briefly presents the existing concepts and tools, which have been used in this work. In Section III the related works have been introduced. Section IV presents the proposed approach. Section V illustrates the experimental setup we have used for conducting the study and in Section VI, we have discussed the result that have been obtained through our experiment and finally this paper has been concluded in Section VII.

II. Preliminaries

In this section, we revisited the existing concepts and tools that helped us to conduct our study for detecting code smells. We have selected the code smells, namely, Brain Class and Brain Method and the concepts of these code smells have been presented in the first part. In the second part of this section, we have briefly presented the CNN type neural network as a deep learning tool.

A. Brain Class

Brain Classes tend to be complex and centralize the functionalities of the system. A class could be considered as a Brain Class if it has at least a few methods affected by Brain Method.Also, if the class has high LOC and functional complexity (WMC) then the class is considered to be a Brain Class even if it has only one Brain Method [18].

Another code smell, named God class [18] appears to be same as Brain class, sometimes. This is partially true, because both refer to complex classes as they centralize the intelligence of the system. Yet the two problems are different in nature. The presence of a God Class is not only having a complex method, but its behaviour on encapsulation breaking, as it directly accesses many attributes from other classes. In contrary, a Brain Class tries to complement the God Class strategy by catching those excessively complex classes that are not detected as God Classes either because they do not abusively access data of "satellite" classes, or because they are a little more cohesive [18].

Equation (1) shows the detection strategy used to identify a Brain Class (BC) [18]. The thresholds are based on Marinescus work [18], [20].

$$BC(C) = \begin{cases} true, \neg GC(C) \wedge \\ \quad ((WMC(C) \geq 47) \wedge (TCC(C) < 0.5)) \wedge \\ \quad (((NBM(C) > 1) \wedge LOC(C) \geq 197)) \vee \\ \quad ((NBM(C) \equiv 1) \wedge (LOC(C) \geq 2.197) \wedge \\ \quad (WMC(C) \wedge 2.47))) \\ false, else \end{cases} \tag{1}$$

- GC(C) is true is C is God Class [18].
- (WMC(C)) is the sum of the cyclomatic complexity of all methods in C [19].
- Tight Class Cohesion (TCC(C)) is the relative number of directly connected methods in C. Two methods are directly connected if they access the same instance variables of C [21].
- NBM(C) is the number of methods identified as Brain Methods [18] in C.
- LOC(C) is the number of lines of code in C.

B. Brain Method

Brain Methods are the methods that centralize the intelligence of a class. A method should avoid size extremities (Proportion Rule). In the case of Brain Methods the problem concerns overlong methods, which are harder to understand and debug, and practically impossible to reuse. A well-written method should have an adequate complexity, which is concordance with the method's purpose (Implementation Rule).

The detection strategy using equation (2) for Brain Methods is proposed in [18], [20].

$$BM(M) = \begin{cases} true, ((LOC(M) > 65) \wedge \\ \quad (CYCLO(M)/LOC(M) \geq 0.24) \wedge \\ \quad (MAXNESTING(M) \geq 5) \wedge \\ \quad (NOAV(M) > 8)) \\ false, else \end{cases} \tag{2}$$

- BM(M) is true, if M is a Brain Method, otherwise false.
- LOC(M) is the number of lines of code of a method.
- CYCLO(C) is cyclomatic complexity of method M [19].
- MAXNESTING(M) is maximum nesting level of control structures within the method [18].
- NOAV(M) is the total number of variables accessed variable directly by method M [18].

The strategy for detecting this design flaw is based on the presumed convergence of three simple code smells described by Fowler [7].

C. Convolutional Neural Networks (CNNs)

A CNN model [3] is a series of convolutions filters for extracting some intended information. Implementing a CNN model [3] mainly requires one input layer, one output layer and one or more hidden layers. The raw input is usually

fed to the input layer. The neurons in this layer is then process the data and provide the output to the neurons in the hidden layer nearby the input layer. Thus the processing of data is performed through a sequence of hidden layers. The last layer is the output layer, which is responsible to provide the prediction result. This layer takes input from the neurons in the last hidden layer and processes these inputs to generate the final output. In a CNN model supported by Keras library [27], there will be single or multiple convolution layers, which performs convolution operations based on the specified number of filters and activation function. The neurons in these layers adjust the weights after every epoch and provides the output to the next layer. The Convolution layers can be followed by a number of optional layers such as pooling layer, dropout layer, flatten layer, etc. Pooling layer is usually used to avoid over-fitting. There are two kind of pooling layers, where in one kind, namely, Max pooling, the maximum weight among a set of weights under the kernel is considered. On the other hand, in another type, known as average pooling, the average value of all the weights under the kernel is considered. Dropout performs another type of regularization by ignoring some randomly selected nodes during training in order to prevent over-fitting. This also helps to increase the efficiency in training process. A flatten can also be used to transform the data into the appropriate format before feeding them to the first dense layer. There can be multiple dense layers. There can be single or multiple dense layers, which are densely connected classifier network. These layers use an appropriate activation function for classification. An activation function is a mathematical gate within a neuron. There are several activation functions, namely, tanh, sigmoid, ReLU (Rectified Linear Unit), Leaky ReLU, Parametric ReLU, Softmax, Swish, etc.

We have used a python library for deep learning, Keras [27] to implement CNN model for detecting the code smells: Brain Class and Brain Method. We have selected this model because we are to find simple patterns that can be used to form complex patterns in higher layers.

III. Related Work

Several approaches have been adapted in the literature to detect code smells. A large number of works have been done to detect smells in source code. Traditionally, metric-based [22] and rule/heuristic-based [23] code smell detection techniques are commonly used. In recent years, machine-learning based code smell detection techniques [24] have emerged as a potent alternative as they not only have the potential to bring human judgment in the smell detection but also provide the grounds for transferring results from one problem to another. Researchers have used Bayesian belief networks [24], [25], support vector machines [24], and binary logistic regression [25] to identify smells.

This paper focuses on deep learning based approaches because it is quite young among the existing approaches. A hybrid detection approach with unsupervised and supervised learning algorithms is proposed in [1] which works on four code smells God Class, Data Class, Feature Envy and Long Method. A code smell detection system is presented in [2] which uses neural network model that delivers the relationship between code smell and object-oriented metrics by taking a corpus of Java projects as the experimental data set. Code smells like God Class, Large Class, Feature Envy, Parallel Inheritance Hierarchies, Data Class and Lazy Class were detected using this model. The authors in [3] proposed a deep learning based approach to identifying feature envy along with an automatic approach to generate labeled training data for detection of feature envy code smell. In the literature, none of the existing deep learning based approaches detect brain method and brain class code smells. The proposed approach is the first one to detect these two code smalls with CNN based deep learning techniques.

IV. Proposed Approach

In this study, we have proposed a deep learning-based approach to detect code smell.

Deep learning architectures are being used extensively for addressing a multitude of detection, classification, and prediction problems. Several deep learning architectures such as Convolution Neural Networks(CNN), Recurrent Neural Networks(RNN) are available. Since we have to find simple patterns, which will be used to form complex patterns in higher layer, we decided to explore Convolution Neural Networks-1D (CNN) model.

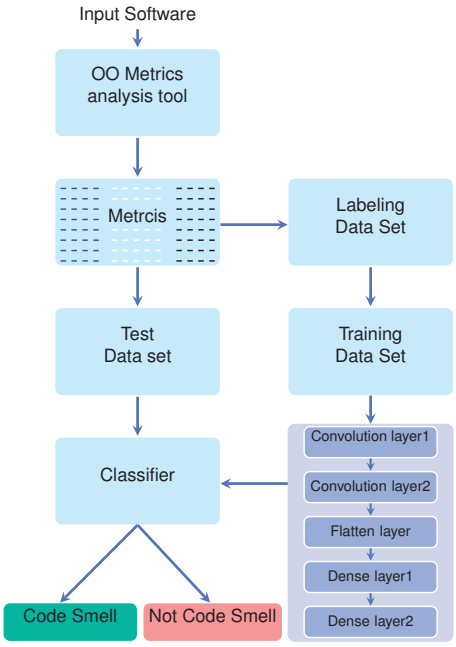

Fig. 1. Proposed Approach

TABLE I
TRAINING METRIC OF BRAIN CLASS

Class Name	God Class	TCC	WMC	LOC	NBM	Brain Class
ClassPathProcessorST	0	0.4	45	358	2	1
emmaTask	0	0.05	10	489	0	0
DataFactory	0	0.07	99	776	2	1
ANTMain	0	1	1	22	0	0
InstrProcessorST	0	0.44	117	997	4	1

TABLE II
TRAINING METRIC OF A BRAIN METHOD

Method Name	LOC	CYCLO	MAXNESTING	NOAV	Brain Method
init	9	1	0	5	0
setClassname	7	2	1	3	0
nextToken	114	16	3	16	1
getDependencies	71	11	5	20	1
computeSUID	228	26	6	69	1

TABLE III
TEST METRIC OF A BRAIN CALSS

Class Name	God Class	TCC	WMC	LOC	NBM
emmajavaTask	1	0.16	78	551	1
EMMARuntimeException	0	0	6	77	0
emmarun	0	1	1	21	0
Command	0	0.09	48	296	0

TABLE IV
TEST METRIC OF A BRAIN METHOD

Method Name	LOC	CYCLO	MAXNESTING	NOAV
setJar	7	2	1	3
setClasspath	6	2	1	2
createClasspath	6	2	1	2
parse	283	36	7	38

Fig. 1 presents the overview of the proposed approach. Based on a large corpus of software applications, it generates a huge number of training samples which are labeled to indicate if it is a code smell of kind Brain class and Brain method. Such training samples are employed to train a neural network based classifier (CNN-1D). At prediction phase, unlabeled test data have been given to the system as input . Details approach are presented in the following sections.

A. Input to the System

Open source Java software are given as raw input to the proposed system. Software's code are then fed into metrics analysis tool to collect the required metrics used to identify Brain class and Brain method.

B. Metric Generation

This work uses an object oriented metrics analysis tool IPlasma [26] which has library of more than 80 state-of-the-art and novel design metrics that can be applied at different levels of abstraction ranging from system-level metrics used to obtain an overview of the system to primitive metrics which describe the details within a single method. We use IPlasma [26] tool to generate required metrices GC, WMC, TCC, NBM, LOC, CYCLO, MAXNESTING and NOAV which are required to detect whether a method m is a brain method and a class c is a brain class in the input Java source code. The metrics are collected in a data structure suitable for CNN 1-D model.

C. Generation of Training Data

To train the neural network based classifier, it needs labeled training data. The rules in equations (1) and (2) are applied on each set of related metrics to detect brain class and brain methods, respectively. One of the biggest challenge for deep learning based smell detectors is to collect a large number of labelled samples to train the classifiers. Methods and classes are then labelled with 1 or 0 based on the result of rules depicted equations (1) and (2) in Section II. Value 1 is tagged to all brain methods and value 0 is tagged for remaining methods. Training data set takes the form as depicted in Table IV.

D. Generation of Test Data

A new set of open source Java software different than the one used to generate training data set are used as input for generating same metrics. The test data set is formed same way as training data set that expect tagging a label for indicating a brain method or brain class. Table III shows an example of test data set.

V. EXPERIMENTAL SETUP

The proposed deep learning CNN model has been trained and tested with a corpus of the ten Java projects, which contain large number of Java classes and source code lines. These data has been used in previous studies and do not include a specific application domain, but a varieties of domains. In this study, the data set 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.

A. Experiment setup to detect Brain Class Using Keras library

We have used Keras library [27] to implement the CNN model for detecting the code smell, namely, Brain Class. The construction of this deep learning model using keras library depicted in Fig. 3 and it is illustrated in the following sequence of phases:

- **Preparation of Data**: The raw data as listed in Table V i.e., the software codes have been preprocessed with the help of Iplasma tool that outputs the required metric values and then the labelling of data has been performed using the formula in equations 1. We have designed an automatic tool for labelling the data.
- **Define Model**: We have used sequential model depicted in Fig. 2 to implement the deep neural network. The

TABLE V
INPUT SOFTWARE

No.	Software Name	NOC	NOM	LOC
1	argouml	1971	17810	179372
2	aspectj	2043	30116	379131
3	axion	236	2910	20615
4	batik	1686	16305	170801
5	c_jdbc	727	5975	85296
6	cayenne	2758	19091	190096
7	cobertura	135	3506	51420
8	collections	469	6324	53878
9	colt	297	3961	41872
10	columba	1181	6869	76327
11	compiere	53	37014	375247
12	displaytag	305	1724	17309
13	drawswf	218	2330	21308
14	drjava	535	7850	80813
15	fitjava	61	456	2916
16	fitlibraryforfitnesse	793	4801	25660
17	freecol	717	8750	107904
18	freecs	139	1404	20720
19	freemind	451	5786	48610
20	galleon	293	3990	99496
21	ganttproject	537	5128	42637
22	hadoop	1950	21352	250602
23	itext	447	5974	68162
24	james	407	3041	31093
25	javacc	158	1170	17403
26	maven	807	6101	53455
27	pmd	748	5859	51148
28	velocity	369	2994	34194
29	webmail	98	722	5051
30	xmojo	157	1382	17639

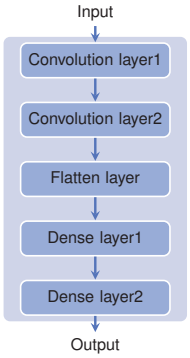

Fig. 2. Architecture of employed CNN Model

following are the sequence of layers we have defined in our model.

- In the first layer of this model, we have defined an 1D CNN layer with 256 filters, and the kernel size we have taken as 1. We have activated the neurons in this layer using the activation function tanh.
- In the second layer, we have defined another 1D CNN layer having 128 filters with kernel size 1. For this layer also we have use tanh as the activation function.
- A flatten layer after the second convolution layer has been added as the third layer to connect the convolution layer with dense layers.
- In the fourth layer, we have defined a dense layer having 128 filters with activation function ReLU.
- Finally, in the last layer we have defined another dense layer having only one filter with activation function "Sigmoid". This layer is acting as the output layer. Here we have used sigmoid as activation function and only one filter because our objective is to obtain a binary prediction.

- **Compile the Model**: After defining the model, the compilation of our model is required. In order to compile the defined model, we have used *adam* for optimization of the defined model. We have used this optimizer to obtain a good adjustment in learning rate during the learning phase of our model. Along with it, we have used *binary-crossentropy* as the loss functions.
- **Training the Model**: The fit() function is used to train the model. The parameters that we have used in this function are the training data, labels of the training data, number of epochs and the batch size. We have used 100 epochs to train our model, while the batch size is 1.
- **Evaluate the Model**: This is the last phase of our experiment. In this phase, we have used the function evaluate(), where we have used the test data and obtains the accuracy of the model.

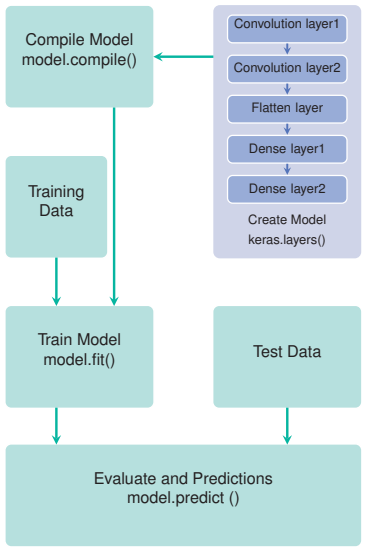

Fig. 3. Keras workflow

Keras provides a very simple workflow for training and evaluating the CNN models. Fig. 3 illustrates the model we have defined. We have defined the model and train the model using the training data. Some of these are depicted in Table I and Table II.

B. Experiment setup to detect Brain Method Using Keras library

Similar to Brain Class, we have used Keras library [27] to implement the CNN model for detecting Brain Method code

smell. The construction is almost similar to the construction of the model for Brain Class code smell.

VI. Experimental Result

The result of our experiment to detect Brain Class and Brain Method code smells have been presented in this section. Table VI and VII illustrates the results for Brain Class and Brain Method, respectively. For brain class, we have used maximum 30 software samples, among which one set of samples have been used for training and another set of samples have been used to test our model. Thus, we have experimented in five phases as shown in Table VI. In first three phases, we have tried 100 epochs and in last two phases, we have tried 300 epochs. In all the phases, we have obtained more than 97% accuracy in predicting Brain Class code smell. For brain method, we have used maximum 28 software samples. In similar fashion, we have selected one set of samples for training and another set of samples for testing our model. As Table VII shows, we have obtained close to 95% accuracy in predicting the Brain Method code smell through both 15 and 30 epochs.

TABLE VI
Brain Class Result

No. of Training Samples	No. of Testing Samples	Epochs	Prediction Accuracy
5	5	100	99.67384910
5	5	100	99.70592953
10	10	100	99.29423087
15	15	300	97.61000909
20	10	300	97.63674277

TABLE VII
Brain Method Result

No. of Training Samples	No. of Testing Samples	Epochs	Prediction Accuracy
4	4	30	95.08431435
8	8	30	94.44499682
4	4	15	95.08829144
12	12	15	94.44996818
14	14	15	94.41019726

Therefore, the results shows that our proposed deep learning approach can be able to classify the software smells almost accurately.

VII. Conclusion

To the best of our knowledge, we are the first to exploit deep learning for detecting Brain Class and Brain Method code smells. We evaluated using the original source code of open-source applications without any revision. Evaluation results suggest that the proposed approach achieves a high accuracy in predicting the Brain Class and Brain Method 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.

References

[1] M. Hadj-Kacem and N. Bouassida, A Hybrid Approach To Detect Code Smells using Deep Learning. International Conference on Evaluation of Novel Approaches to Software Engineering (2018), p137-146.
[2] K. Dong Kwan, Finding Bad Code Smells with Neural Network Models, International Journal of Electrical and Computer Engineering.
[3] L. Hui, X. Zhifeng, Z. Yanzhen, Deep Learning Based Feature Envy Detection, Conference on Automated Software Engineering (2018).
[4] T. Mens, and T. Tourwe, A survey of software refactoring, IEEE Transactions on Software Engineering(2004), p126–139.
[5] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, A textualbased technique for Smell Detection. International Conference on Program Comprehension (2016), p1–10.
[6] Z. Min, H. Tracy and B. Nathan, Code Bad Smells: a review of current knowledge. Journal of Software Maintenance and Evolution: Research and Practice (2011), p179–202.
[7] F. Martin, B. Kent, B. John, O. William and R. Don, Refactoring: Improving the Design of Existing Code. AddisonWesley Professional(1999).
[8] A. April and A. Abran A, Software maintenance management: evaluation and continuous improvement(2012).
[9] W.H. Brown, R.C. Malveau, H.W. McCormick and T.J. Mowbray, AntiPatterns: refactoring software, architectures, and projects in crisis (1998).
[10] S. Tushar and S. Diomidis, A survey on software smells. Journal of Systems and Software(2018), p158 – 173.
[11] H. Liu, Q. Liu, Z. Niu, and Y. Liu. 2016. Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection, IEEE Transactions on Software Engineering (2016), p544–558.
[12] V.M. Mika and L. Casper, Subjective evaluation of software evolvability using code smells: An empirical study. Empirical Software Engineering (2006), p395–431.
[13] A.F. Francesca, V.M. Mika, Z. Marco and M. Alessandro, Comparing and experimenting machine learning techniques for code smell detection, Empirical Software Engineering (2016), p1143–1191.
[14] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik and A. De Lucia, Detecting code smells using machine learning techniques: Are we there yet?. International Conference on Software Analysis, Evolution and Reengineering (2018). p612–621.
[15] K. Philippe, L.N. Robert and O. Ipek, Technical Debt: From Metaphor to Theory and Practice (2012), p18–21.
[16] L. Yann, B. Yoshua and H. Geoffrey, Deep learning. nature (2015), 436.
[17] G. Ian, B. Yoshua, C. Aaron and B. Yoshua Bengio, Deep learning. Vol. 1.MIT press Cambridge(2016).
[18] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice(2006) Springer.
[19] S. R. Chidamber and C.F. Kemerer, A Metrics Suite for Object Oriented Design (1994), p476-493.
[20] C. Daniela and I.K.S. Dag, Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems (2010).
[21] J. M. Bieman and B.K. Kang, Cohesion and Reuse in an Object Oriented System. Proc. Int'l Symp. Software Reusability (1995), p259-262.
[22] R. Marinescu, Measurement and quality in object-oriented design. Conference on Software Maintenance (2005), p701–704.
[23] M. Naouel Moha, G. Yann-Ga¨el, D. Laurence and L.M. Anne-Fran, DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Trans. Software Eng. (2010), p20–36.
[24] M. Abdou, A. Nasir, B. Neelesh, S. Aminata, G. Yann-Ga¨el, A. Giuliano and A. Esma, Support vector machines for anti-pattern detection. International Conference on Automated Software Engineering (2012), p278–281.
[25] B. S´ergio, B.A. Fernando and M. Miguel, Reducing subjectivity in code smells detection: Experimenting with the Long Method, International Conference on the Quality of Information and Communications Technology (2010), p337–342.
[26] C. Marinescu, R. Marinescu, P. Florin Mihancea, D. Ratiu, and R. Wettel, iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design, International Conference on Software Maintenance (2005), p77–80.
[27] Chollet, Francis and others, Keras (2015), keras.io.