Deep semantic-Based Feature Envy Identification

Xueliang Guo
School of Computer Science, Beijing Institute Of Technology
Beijing, China
guoxueliang@bit.edu.cn

Chongyang Shi
School of Computer Science, Beijing Institute Of Technology
Beijing, China

He Jiang
School of Computer Science, Beijing Institute Of Technology
Beijing, China

ABSTRACT

Code smells regularly cause potential software quality problems in software development. Thus, code smell detection has attracted the attention of many researchers. A number of approaches have been suggested in order to improve the accuracy of code smell detection. Most of these approaches rely solely on structural information (code metrics) extracted from source code and heuristic rules designed by people. In this paper, We propose a method-representation based model to represent the methods in textual code, which can effectively reflect the semantic relationships embedded in textual code. We also propose a deep learning based approach that combines method-representation and a CNN model to detect feature envy. The proposed approach can automatically extract semantic and features from textual code and code metrics, and can also automatically build complex mapping between these features and predictions. Evaluation results on open-source projects demonstrate that our proposed approach achieves better performance than the state-of-the-art in detecting feature envy.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

Code Smell, Deep Learning, Software Refactoring, Feature Envy, Deep Semantic

ACM Reference format:
Xueliang Guo, Chongyang Shi, and He Jiang. 2019. Deep semantic-Based Feature Envy Identification. In *Proceedings of Internetware '19: Proceedings of the 11th Asia-Pacific Symposium on Internetware, Fukuoka, Japan, October 28–29, 2019 (Internetware '19),* 6 pages.
https://doi.org/10.1145/3361242.3361257

1 INTRODUCTION

Code smell detection has become an established approach to discovering problems in source code (or design) to be corrected through software refactoring, with the aim of improving software quality[6, 15]. However, most projects have large amounts of code and complex structures, and different developers have different coding styles; therefore, it is very hard to accurately detect code smells by way of manual inspection. To reduce the cost of detection, many automatic or semi-automatic approaches have been proposed to identify code smells[1, 19, 27].

Most approaches to code smell detection rely solely on structural information (code metrics) extracted from the source code and heuristic rules. However, code metrics only contain the numerical features embedded in source code, and the understanding of the definition of code smells is very subjective[26]. Therefore, it is very difficult to manually build rules, which could also lead to the variance in performance evaluation[13]. To deal with the confusion surrounding this lack of consensus and to facilitate the identification of code smells, traditional machine learning based approaches to code smell detection have been proposed, such as SVM, J-48, and Naive Bayes[11]. These approaches can provide additional objectivity to avoid this problem by extracting features from code metrics automatically[7]. However, some studies demonstrate that these traditional machine learning based approaches do not achieve good performance when they detect two or more kinds of code smells[5].

Recently, deep learning techniques have also been applied to code smell detection. Liu et al.[12] propose a deep learning based approach to detect feature envy. It exploits some textual information in its feature envy detection method, but it does not consider the semantics relationships embedded in different levels of code.Unlike traditional machine learning techniques, deep learning technology provides multiple levels of abstraction of the data, ranging from low to high levels, and achieves better performance on deep feature extraction[2].

Accordingly, in this paper, we propose method-representation, which is based on attention mechanism and LSTM to represent the textual information of methods in the source code. Word2Vec has been proven to be efficient for learning high-quality distributed vector representations that capture precise syntactic and semantic word relationships[16, 17]. Similar to Word2Vec, the method-representation approach we propose achieves good performance on the vector representations of methods, which could facilitate the extraction of semantic features from the textual information, and precisely reflect the contextual relationships between the code parts.

We also propose a novel approach that combines method-representation and a CNN model to detect feature envy more accurately. The deep semantic based approach we propose could automatically extract semantic and features from textual code and code metrics. Consequently, we build a CNN and method-representation based classifier

Internetware '19, October 28–29, 2019, Fukuoka, Japan
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7701-0/19/10…$15.00
https://doi.org/10.1145/3361242.3361257

that can classify the given methods into two types, smelly or no-smelly.

We evaluate our proposed approach on datasets from 74 well-known open-source projects[22]. Firstly, we preprocess the datasets and split them into two parts, namely the textual part and metrics part. Secondly, we evaluate the proposed approach on the datasets to detect feature envy. The experimental results suggest that our proposed approach achieves better performance than the state-of-the-art approaches. The main contributions of this paper are as follows:

- Proposing method-representation, which is based on attention mechanism and LSTM networks, to represent the textual information of methods in source code. It can extract the semantic features from the textual information, and reflect the contextual relationships among the code parts. More specifically, through the attention mechanism, method-representation extracts the specific features that are significant to code smell detection. The semantic features can effectively supplement the limited structural information in code metrics, and improve the accuracy of code smell detection.
- Proposing a deep semantic based approach that combines method-representation and a CNN model to detect feature envy. It can extract semantic and features from textual code and code metrics, and construct complex mapping between such features and predictions.
- Evaluating our proposed approach on datasets from 74 open-source projects. The results suggest that our approach achieves significantly better performance than state-of-the-art approaches.

2 RELATED WORK

Researchers have presented various tools and approaches for detecting code smells. Most of these approaches are based on rules that rely on manually designed heuristics to map code metrics[9]. Marinescu et al.[14] propose a metric-based approach to detecting code smells, in which they use detection strategies to formulate metric-based rules that capture deviations from good design. Moha et al.[18] propose a rule-based approach named *DÉCOR* requiring that the rules are specified in the form of domain specific language; this specification process must be undertaken by domain experts, engineers or quality experts.

Recently, many traditional machine learning based detection approaches have been proposed[7]. Khomh et al.[5] present BDTEX (Bayesian Detection Expert), a Goal Question Metric approach to building Bayesian Belief Networks from the definitions of anti-patterns; and validate BDTEX with Blob, Functional Decomposition, and Spaghetti Code anti-patterns on two open-source programs. Wang et al.[8] propose an approach that assists in understanding the harmfulness of intended cloning operations using Bayesian Networks and a set of features such as history, code, and destination features. Yang et al.[10] study the judgment of individual users by applying machine learning algorithms on code clones. Recently, deep learning techniques also have been used in code smell detection. Liu et al.[12] propose a deep learning based approach to detect feature envy, which achieved good performance on open-source applications. They also propose an approach to automatically generating the datasets infected with code smells for code smell detection.

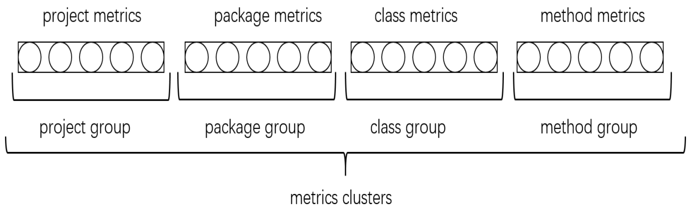

Figure 1: The structure of metric cluster.

3 METHODOLOGY

In this section, we propose a deep semantics based approach that combines method-representation and a CNN model to detect feature envy. The input of the proposed classifier is presented in the first section; in the following sections, we will present the details of the proposed model.

3.1 Input

To enable better extraction of the features, we use both structural information (code metrics) and textural information as the input of the proposed approach. For the structural information, in addition to the metrics of method, we also use the metrics of project, package and class. We split the four levels of metrics into four different metric groups, and form those metric groups into one metric cluster. Fig. 1 shows the structure of metric clusters. For each given metric cluster, there are many features that it includes, the most prominent of which is the containment relation.

In addition to the metrics, we also exploit textual information in the source code, including the name of the project, the name of the package, the name of the class and the name of the method. Ideally, the semantic features embedded in the name of the method may be associated with code smells. In general, the names of code parts are based on the functions of these code parts. For example, if a method name includes a lot of meaning, the method may have long method code smells. As a conclusion, the input of the proposed approach is a group including two parts; the textual part and the metrics part.

$$input = < textual\ input,\ metrics\ cluster > \tag{1}$$

$$textual\ input = < name_{pr},\ name_{pa},\ name_{c},\ name_{m} > \tag{2}$$

$$metric\ cluster = < group_{pr},\ group_{pa},\ group_{c},\ group_{m} > \tag{3}$$

where $name_i$ is the identifier of the code entity i. pr expresses the project, pa represents the package, c expresses the class, and m is the method. In the same way, $group_{pr}$, $group_{pa}$, $group_c$, $group_m$ respectively express the groups of project, package, class, and method.

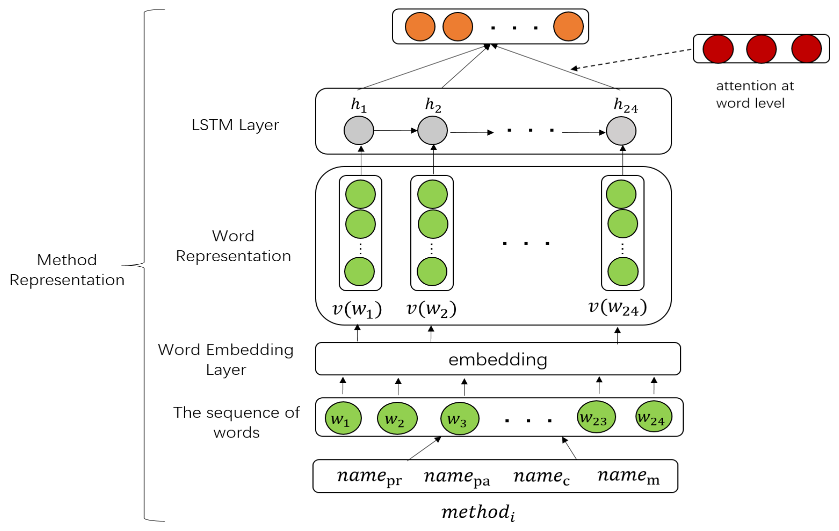

Figure 2: Method Representation.

3.2 Method-Representation Based Semantic Extraction

To obtain the semantic features from the source code, we propose an attention and LSTM deep neural network based approach to represent methods, which we refer to as method-representation.

For each identifier of code parts composed of natural languages, we partition it into a sequence of words according to word segmentation algorithms, capital letters and underscores.To save computer memory and facilitate the design of deep neural networks, we set six as the length of the word sequence for each code part identifier. The representation of the identifier of code parts is as follows:

$$name_i = < w_1,\ w_2,\ \cdots,\ w_k > \tag{4}$$

$name_i$ is the identifier of code part i, and it can be partitioned to a list of words $< w_1,\ w_2,\ \cdots,\ w_k >$.

According to Formula 2, we compose all words in the identifiers of four levels into a long word sequence. In this way, we can view the sequence as a long sentence that contains the textual information of project, package, class and method. Using this sentence, we can extract ample features from these relations, such as the inclusion relationships and the coordinating relationships.

The structure of the method-representation is presented in Fig. 2. It consists of three main components: the word embedding layer, LSTM layer, and the attention layer. The word embedding layer can convert each word into a numerical vector, which is accomplished by the well-known Word2vec[4, 20, 21]. According to Formulas 2 and 4, the definition of method input and the word representation of method input is as follows:

$$\begin{aligned} m\_input =& concat(name_{pr},\ name_{pa},\ name_c,\ name_m) \\ =& concat(w_1,\ w_2,\ \cdots,\ w_n),\ n = 24 \qquad (5) \\ V(m\_input) =& concat(V(w_1), V(w_2), \cdots, V(w_n)),\ n = 24 \qquad (6) \end{aligned}$$

$concat(\cdot)$ is the function used to concatenate all parts together. $V(w_i)$ converts word w_i into a numerical vector with Word2vec, and $V(m\_input)$ represents the word representation of the method input.

The LSTM layer is used to extract the contextual relations from the sentence of method input, and to obtain semantic features from these relations. It uses three gates to control the state flow in the LSTM unit. At each time step t, given the method vector $V(w_t)$, the current cell state c_t and hidden state h_t can be updated with the previous cell state c_{t-1} and hidden state h_{t-1}, as follows:

$$\begin{bmatrix} i_t \\ f_t \\ o_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \end{bmatrix} (\mathbf{W}[h_{t-1}; V(w_t) + \mathbf{b}]), \tag{7}$$

$$\hat{\mathbf{c}_t} = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}; V(w_t)] + \mathbf{b}_c), \tag{8}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1}^{\hat{}} + \mathbf{i}_t \odot \hat{c_t}, \tag{9}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{10}$$

where i_t, f_t and o_t are gate activations and in [0,1], σ is *logistic sigmoid* function and $\odot$ means element-wise multiplication. Generally, the input gate i_t controls the amount of information with which each unit should be updated, the forget gate f_t controls how much information should be forgotten by the previous memory cell, and the output gate o_t controls how much information the internal memory state should be exposed to. The hidden state h_t denotes the output information of the LSTM unit's internal memory cell.

However, not all features embedded in hidden states equally reflect the relationships between the code parts, and thereby benefit code smell detection. Thus, directly using all hidden states $h_{1:n}$ will not accurately represent the methods in the source code[25]. In order to avoid this issue, we use the self-attention mechanism[23, 24] to extract the specific context features that are significant to code smell detection. Finally, the method-representation is composed by the representations of these features. Formally, the method-representation $\mathbf{m}_i$ is a weighted sum of hidden states, as follows:

$$\mathbf{m}_i = \sum_{t=1}^{n} \mathbf{a}_t \mathbf{h}_t,\ n = 24 \tag{11}$$

where $\mathbf{h}_t$ is the hidden state of the t-th word in the i-th method, $\mathbf{a}_t$ is the attention weight of $\mathbf{h}_t$ and measures the importance of the t-th hidden states for code smell detection. More specifically, the attention weight $\mathbf{a}_t$ for each hidden state can be defined as:

$$e(\mathbf{h}_t) = \mathbf{W}_1 \tanh(\mathbf{W}_2 \mathbf{h}_t) \tag{12}$$

$$\mathbf{a}_t = \frac{\exp(e(\mathbf{h}_t))}{\sum_{k=1}^{K} \exp(e(\mathbf{h}_k))} \tag{13}$$

where $\mathbf{W}_1$ and $\mathbf{W}_2$ are weight matrices, and $e(\cdot)$ is a score function that scores the importance of hidden states for code smell detection.

3.3 CNN and Method-Representation Based Classifier

The structure of the CNN and method-representation based classifier is presented in Fig. 3. The input of this structure has two parts, namely the numerical input and the textual input. According to Formula 2, the textual input is a long sequence composed of the identifiers of project, package, class and method. According to Formula 3, the numerical input is a metric cluster created by concatenating the metrics of four levels (method, class, package, project).

In the first part ① , we feed the textual input into the method-representation, so that it can convert the textual information into vectors representing the description of the method. The process of this conversion is introduced in section B.

In the second part ②, the metric cluster is fed into the Convolutional Neural Network (CNN) based model, which has three convolutional layers and does not set the pooling layer. The CNN-based model can extract the features from the structural information in the code metrics in order to fully reflect the relations between adjacent metrics. Finally, we apply a flatten layer to turn the shape of the input into a one-dimensional vector.

The third part ③ of the proposed model is the MLP based neural network. Firstly, the outputs of the CNN model and the output of the method-representation are connected at the connection layer, which simply concatenates all inputs, including the features from the textual input and the metrics input. Behind the connection layer, we set two dense layers and one output layer to facilitate the final classification, which maps the textual input and the metrics input into a single output. Finally, the output layer only has 1 neuron, which represents the result of the identifier, i.e. smelly or no-smelly, and the activation function in this layer is a *sigmoid* function. We select *binary_crossentropy* as the loss function; *binary_crossentropy* is defined as follows:

$$L = \sum_{i=1}^{N} y^{(i)} \log y^{\hat{(i)}} + (1 - y^{(i)}) \log(1 - y^{\hat{(i)}}) \tag{14}$$

where $y^{\hat{(i)}}$ is the true type of the code parts, and $y^{(i)}$ is the prediction of our proposed model.

4 EXPERIMENTS

The purpose of these experiments is to act as a proof of concept demonstrating that the proposed approach performs well in detecting feature envy. It mainly evaluates the performance of the proposed approach against the state-of-the-art approaches in detecting feature envy.

4.1 Datasets

In this paper, we evaluate the proposed approach on two feature envy datasets from Fontana et al[7]. In the following paragraphs, we present some the information about the datasets. The datasets themselves, along with a more detailed description of these datasets are available at **http://essere.disco.unimib.it/reverse/MLCSD.html**

These datasets come from 74 open-source projects, which contain the source code for various domains and all kinds of styles. They mainly comprise the metrics of source codes and the label (smelly or no-smelly). The code metrics and the label are extracted by means of refactoring tools and have been corrected manually. In addition, we use the SMOTE[3] algorithm to make these datasets more balanced; then each dataset contains 1/5 smelly samples, 4/5 are no-smelly samples. In this paper, two feature envy datasets are extracted using Fluid Tool and iPlasma, respectively.

In the following subsections, we carry out a k-fold(k=5) cross-validation on the datasets. For each dataset, we devide it into five; the one fold is used as the testing data whereas the others are used as training data. We evaluate our proposed approach on the test datasets, and make comparisons with the help of the experimental results.

4.2 Experimental Design

We evaluate our performance of the proposed approach and the state-of-the-art approaches on two feature envy datasets. The compared approaches we select are MLP and J48[7]. MLP is a famous classification method based on deep neural network. J48 is an implementation of the C4.5 decision tree and it is a famous machine learning approach. This algorithm produces human understandable rules for the classification of new instances. The reasons why these approaches have been selected comparison are as follows. First, these approaches are always used to identify code smells[7]. Second, these approaches not only have good performance on code smell detection, but also do well in other classified tasks. The reason why we do not compare our proposed approach against with the approach proposed by Liu et al.[12] is that their approach only applies to the specific datasets they generate.

After the training process, we evaluate these detection approaches using recall and precision on test sets, which are well-known metrics in information retrieval and pattern recognition:

$$precision = \frac{TP}{TP + FP} \tag{15}$$

$$recall = \frac{TP}{TP + FN} \tag{16}$$

Where TP represents the number of true positive bad smells detected, FP represents the number of false positive bad smells detected, and FN represents the number of false negative bad smells detected. We then use F-measure, which is the harmonic mean of precision and recall, to combine precision and recall.

$$F_1 - Measure = 2 * \frac{precision * recall}{precision + recall} \tag{17}$$

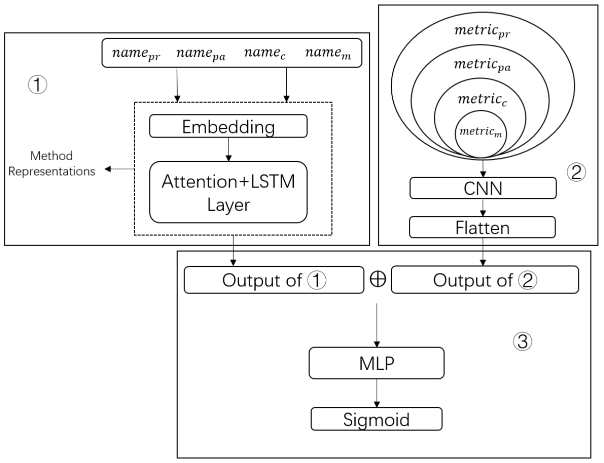

Figure 3: The CNN and Method-Representation based model.

Table 1: Evaluation Results on Feature envy Datasets

Feature Envy Datasets	Proposed Approach			MLP			J-48		
	Precision	*Recall*	F_1	*Precision*	*Recall*	F_1	*Precision*	*Recall*	F_1
Iplasma	97.15%	99.56%	98.34%	99.24%	79.54%	88.31%	97.15%	96.97%	97.05%
Fluid Tool	96.98%	98.69%	97.83%	98.20%	75.96%	85.67%	97.21%	96.59%	96.90%
Average	**97.07%**	**99.13%**	**98.09%**	**98.77%**	**77.75%**	**86.99%**	**97.18%**	**96.78%**	**96.98%**

5 RESULTS AND DISCUSSION

To prove our proposed approach have better performance than the state of the art, we compare the proposed approach against MLP and J48, by detecting feature envy. The evaluation results of feature envy detection are presented in Tables 1. The first column presents the name of datasets. Columns 2-4 present precision, recall, and F-measure of the proposed approach, respectively. The performance of MLP and J-48 is presented in columns 5-7 and columns 8-10, respectively. The last row of the table presents the average performance of the selected approaches.

From Tables 2, we can observe that our approach achieves both high precision and high recall; thus, its F-Measure is also high. First, the proposed approach can detect more feature envy than the traditional machine learning based approaches. From the Tables we can see that the average recall of the proposed approach is 99.13%. Comparing with MLP and J-48, It improves recall dramatically by 21.56% (=99.13%-77.75%) and 2.36% (=99.13%-96.78%), respectively. However, the average precision of the proposed approach is 97.07% which is slightly lower than the average precision of MLP(98.77%) and J-48 (97.18%).

Second, the proposed approach significantly outperforms the state-of-the-art approaches where the F-measure is concerned. For these two feature envy datasets, the proposed approach improves average F-measure by 11.1% (=98.09%-86.99%) and 1.13% (=98.09%-96.98%).

From these results, we can conclude that our proposed approach capture the more context information between code methods with using the semantic information existing in the textual input, and it achieves better performance relative to the machine learning based approaches when it comes to detecting feature envy. In addition, the proposed approach also outperforms than the simple deep neural network based approaches such as MLP based approaches.

6 CONCLUSION

In this paper, we propose method-representation which is based on attention and LSTM, to represent the methods contained in code. This approach can extract the semantic features from the textual information, as well as reflect the contextual relationships among the code parts. We also propose a deep semantic based classifier that combines method-representation and a CNN model to detect code smells of feature envy. This enables the automatic selection of useful numerical and textual features, along with the mapping of these features to enable automatic classification.

We evaluate our approach through experiments on two feature envy datasets. In the first part, we compare our proposed approach with the state-of-the-art approaches based on machine learning technology. The results show that our proposed approach achieves better performance than the state-of-the-art approaches in detecting code smells of feature envy.

In the future, we will validate our approach on different types of software systems. We also plan to use our approach to represent the code parts of other levels and detect the code smells of other levels. Furthermore, we will also explore deep learning based automated refactoring to remove code smells automatically.

ACKNOWLEDGEMENT

This work was supported by the National Key Research and Development Program of China[No. 2018YFB1003903].

REFERENCES

[1] Jehad Al Dallal. 2015. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and software Technology* 58

(2015), 231–249.

[2] Yoshua Bengio et al. 2009. Learning deep architectures for AI. *Foundations and trends® in Machine Learning* 2, 1 (2009), 1–127.

[3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[4] Huimin Chen, Maosong Sun, Cunchao Tu, Yankai Lin, and Zhiyuan Liu. 2016. Neural sentiment classification with user and product attention. In *Proceedings of the 2016 conference on empirical methods in natural language processing*. 1650–1659.

[5] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting code smells using machine learning techniques: are we there yet?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 612–621.

[6] Palomba Fabio, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *40th International Conference on Software Engineering, ICSE 2018*. ACM, 482–482.

[7] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (2016), 1143–1191.

[8] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[9] Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and Ali Ouni. 2014. A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Transactions on Software Engineering* 40, 9 (2014), 841–861.

[10] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2009. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*. IEEE, 305–314.

[11] Jochen Kreimer. 2005. Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science* 141, 4 (2005), 117–136.

[12] Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 385–396.

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

[14] Radu Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*. IEEE, 350–359.

[15] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.

[16] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[18] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2010), 20–36.

[19] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. 2016. A textual-based technique for smell detection. In *2016 IEEE 24th international conference on program comprehension (ICPC)*. IEEE, 1–10.

[20] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. 2015. End-to-end memory networks. In *Advances in neural information processing systems*. 2440–2448.

[21] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Grammar as a foreign language. In *Advances in neural information processing systems*. 2773–2781.

[22] Di Wu, Nabin Sharma, and Michael Blumenstein. 2017. Recent advances in video-based human action recognition using deep learning: a review. In *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2865–2872.

[23] Zhen Wu, Xin-Yu Dai, Cunyan Yin, Shujian Huang, and Jiajun Chen. 2018. Improving review representations with user attention and product attention for sentiment classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[24] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*. 2048–2057.

[25] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 1480–1489.

[26] Stanley Benjamin Zdonik and David Maier. 1990. *Readings in object-oriented database systems*. Morgan Kaufmann.

[27] Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice* 23, 3 (2011), 179–202.