The Journal of Systems and Software 224 (2025) 112375

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# EnseSmells : Deep ensemble and programming language models for automated code smells detection☆

Anh Ho [a], Anh M.T. Bui [b,*], Phuong T. Nguyen [c], Amleto Di Salle [d], Bach Le [a]

[a] The University of Melbourne, Australia
[b] Hanoi University of Science and Technology, Viet Nam
[c] Università degli studi dell'Aquila, 67100 L'Aquila, Italy
[d] Gran Sasso Science Institute, Italy

ARTICLE INFO

*Keywords:*
Code smells
Software metrics
Pre-trained Models

ABSTRACT

A smell in software source code denotes an indication of suboptimal design and implementation decisions, potentially hindering the code understanding and, in turn, raising the likelihood of being prone to changes and faults. Identifying these code issues at an early stage in the software development process can mitigate these problems and enhance the overall quality of the software. Current research primarily focuses on the utilization of deep learning-based models to investigate the contextual information concealed within source code instructions to detect code smells, with limited attention given to the importance of structural and design-related features. This paper proposes a novel approach to code smell detection, constructing a deep learning architecture that places importance on the fusion of structural features and statistical semantics derived from pre-trained models for programming languages. We further provide a thorough analysis of how different source code embedding models affect the detection performance with respect to different code smell types. Using four widely-used code smells from well-designed datasets, our empirical study shows that incorporating design-related features significantly improves detection accuracy, outperforming state-of-the-art methods on the MLCQ dataset with improvements ranging from 5.98% to 28.26%, depending on the type of code smell.

## 1. Introduction

*Code smell* (Fowler, 1999) is the term used to discernible traits or recurring patterns in software source code, indicating probable deficiencies or regions warranting enhancement. Code smells differ from conventional software bugs or errors in that they do not manifest as malfunctions; instead, they function as indications of latent design or implementation quandaries with the potential to precipitate issues in subsequent phases of development and maintenance. Identifying and addressing code smells is an important part of code review and maintenance, as it can help minimize technical debt and make code easier to understand, modify, and extend.

Various studies have been undertaken to investigate the smell metaphor in software engineering. These studies includes comprehensive elucidations of different kinds of code smells (Zhang et al., 2011; Singh and Kaur, 2018), methodologies for detecting such smells (Zhang et al., 2022; Sharma et al., 2021) as well as quantifiable metrics that may serve as indicators for discerning bad smells (Marinescu, 2005; Sharma et al., 2016). Software metrics (Lanza and Marinescu, 2006) have been widely studied to identify design defects in the source code (Marinescu, 2004; Munro, 2005; Van Rompaey et al., 2007). Approaches to code smell detection, whether based on metrics or heuristics, often depended on manually setting thresholds for various measurements. While these methods obtain promising outcomes, crafting optimal rules or heuristics manually proves to be highly challenging (Liu et al., 2016). Alternatively, relying solely on software metrics could result in a potential classification bias, as code snippets exhibiting distinct behaviors may share identical metrics' value while exhibiting entirely different code smells.

In order to bypass the need for manually designed heuristics and rules based on thresholds, machine learning techniques have been adopted to establish a sophisticated relationship between code metrics and smell labels. A significant amount of research has focused on the application of traditional classification algorithms, such as SVM (Maiga et al., 2012a), Naive Bayes (Khomh et al., 2011), regression (Fontana

☆ Editor: Alexander Chatzigeorgiou.

* Corresponding author.
*E-mail addresses:* anh.ho1@student.unimelb.edu.au (A. Ho), anhbtm@soict.hust.edu.vn (A.M.T. Bui), phuong.nguyen@univaq.it (P.T. Nguyen), amleto.disalle@gssi.it (A. Di Salle), bach.le@unimelb.edu.au (B. Le).

https://doi.org/10.1016/j.jss.2025.112375
Received 26 January 2024; Received in revised form 5 February 2025; Accepted 6 February 2025
Available online 15 February 2025
0164-1212/© 2025 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

and Zanoni, 2017), for the detection of code smells. Despite their promising potential, these approaches have often overlooked the examination of code representation features. Deep learning models have subsequently been adopted for the extraction of textual representation from source code. Numerous studies have been dedicated to the exploration of source code mining using deep neural networks (DNNs) (Hadj-Kacem and Bouassida, 2018; Liu et al., 2021; Ho et al., 2022), convolutional neural networks (CNNs) (Liu et al., 2021; Das et al., 2019), long short-term memory networks (LSTMs) (Ho et al., 2023), recurrent neural networks (RNNs) (Sharma et al., 2021), to name but few. Prior studies have primarily regarded source code as textual data and applied natural language processing techniques for text mining. However, it is evident that there exists syntactical distinctions between source code and natural language. Approaching source code as text for mining purposes and overlooking the semantic intricacies within its underlying structures (e.g., nesting control), poses the potential of failing to preserve the intended meaning. Moreover, code smells are frequently linked to distinct symptoms that manifest in control and data flows.

In this paper, we propose a holistic approach to code smell detection, focusing on assessing the effectiveness of incorporating both statistical semantic structures and design-related features to capture the relationship between various types of smells and source code. In particular, we conceptualize EnseSmells for code smell detection on top of a two-tier deep learning architecture. The first tier focuses on acquiring statistical semantic representations of code by integrating DeepSmells (Ho et al., 2023), including CNNs and LSTMs, as an adapter layer to extract code-smell-specific features from code embeddings generated by pre-trained models such as Code2Vec, CodeBERT, and CuBERT. Furthermore, as code smell symptoms are often linked to specific design metrics, the second tier employs deep neural networks (DNNs) with a single adaptive layer to emphasize critical metrics. This tier is designed to learn from selected features, preserving the relationship between code smell symptoms and these metrics. The outputs from both tiers are subsequently concatenated and fed into an Imbalanced Deep Neural Networks for a more effective imbalanced classification.

In summary, this paper extends our previous work (Ho et al., 2023) with the following enhancements:

- We enhance the approach presented in the conference version by introducing a structural module designed to automatically learn the relationships between different code smells and software metrics.
- We extend the scope of experiments with a three-fold objective:
    1. Investigating the impact of adjusting various code embedding techniques on prediction performance, considering four types of code smells (i.e., Long Method, Feature Envy, Data Class, God Class). This paper broadens the scope of code representation by exploring multiple code embedding approaches such as Code2Vec, CodeBERT, and CuBERT.
    2. Examining whether the combination of code representation vectors and hand-crafted metrics can effectively contribute to the detection of code smells.
    3. Demonstrating the effectiveness of EnseSmells compared to state-of-the-art studies.
- We publish the packages developed alongside the metadata processed in this paper to promote future research.[1]

Section 2 introduces the background related to the overview of code smells, the survey of software metrics for code smell detection, as well as pre-trained programming language models. It also provides an overview of related work in our research. Section 3 presents the proposed EnseSmells model for code smell detection. Experimental settings, including the benchmark datasets, the employed evaluation metrics, and our evaluation plan, are specified in Section 4. Section 5 discusses the results and address potential threats to the validity of our proposed approach in Section 6. Finally, Section 7 concludes the paper with insights into future directions.

[1] https://github.com/brojackvn/JSS-EnseSmells

## 2. Background and related work

This section offers a brief overview of the key elements in code smell detection. Initially, we explore different code smells that have garnered attention in recent times. We also provide an overview of software metrics, considering their relevance in the study of code smells. Following this, our focus shifts to deep learning architectures, emphasizing their role in code representation and their importance in the identification of code smells.

### 2.1. Categorization of code smells

Fowler introduced the concept of *code smells* to describe potential issues within software source code that require refactoring (Fowler, 1999). Software developers have identified a wide range of code smells which can be classified into categories such as *implementation*, *design* and *architecture* depending on their granularity level and the overall impact on the source code (Fowler, 1999; Suryanarayana et al., 2014). Implementation smells emerge at a detailed level including *long method, long parameter list, complex conditional*, among others (Fowler, 1999). Design smells refer to violations of object-oriented design principles, such as abstraction and coupling, and are exemplified by issues like *god class, data class*, and *multifaceted abstraction*, among others. Suryanarayana et al. (2014). Architectural smells, including instances like *god component and scattered functionality*, manifest at a high level of granularity, spanning across multiple components and affecting the entire system (Garcia et al., 2009).

In this study, we focus on four types of code smells including *feature envy, long method, data class* and *god class* as they have been widely used (Azeem et al., 2019; Sharma and Spinellis, 2018). Moreover, we utilized the MLCQ dataset containing these four smells as the benchmark (Madeyski and Lewowski, 2020). A detailed explanation of these code smells is provided below.

**Feature Envy.** Beck and Fowler introduced the concept *feature envy* to describe a scenario where a method accesses the data of another object more than its own data (Fowler, 1999). This kind of code smell indicates misplaced methods, typically characterized by frequent interactions with the attributes and methods of other classes.

**Long Method.** A method that takes on too many responsibilities is regarded as a sign of the *long method*. This can be characterized by source code complexity measurements such as lines of code (LOCs), Halstead's metrics (Curtis et al., 1979), McCabe Cyclomatic (McCabe, 1976).

**God Class.** A *god class*, also referred to as a *Blob*, is characterized as a class that dominates a system's functionality by taking on too many responsibilities. This is reflected in its extensive number of attributes, methods and interactions with data classes. Similar to *long method*, this code smell is also distinguished by a high number of lines of code and a complex vocabulary.

**Data Class.** The code smell *data class* describes a class in a software system that mainly functions as a repository for data, offering limited functionality. These classes serve primarily as data holders, while the logic related to the data is often scattered across the code-base. This dispersion can result in maintenance difficulties and increase the error-proneness (Fowler, 1999).

### 2.2. Software metric-based approaches for code smell detection

A significant amount of research has utilized software metrics to detect code smells. These studies are commonly known as *metric-based approaches* and/or *rule/heuristic-based approaches* (Marinescu, 2004, 2005; Macia et al., 2010; Vidal et al., 2015; Lanza and Marinescu, 2006). Marinescu et al. introduced a detection strategy based on design metrics including *Weighted Method Count* (WMC), *Tight Class Cohesion* (TCC) and *Access To Foreign Data* (ATFD) to identify the *god class* code smell (Marinescu, 2004). This approach was further expanded to incorporate various formulas based on source code complexity and design metrics, enabling the detection of ten different code smells (Marinescu, 2005). Lanza and Marinescu combined metric-based formulas with thresholds to detect 11 anti-patterns (Lanza and Marinescu, 2006). They proposed heuristics by logically combined metric-threshold pairs using logical operators to define detection rules. These heuristics have been adopted inside the *InCode* (Marinescu et al., 2010) package as an Eclipse-plugin. Sales et al. have proposed a dependency based approach, called *JMove* to identify *Feature Envy* (Sales et al., 2013). This approach involves defining metrics to quantify the similarity between the dependencies created by the considering method and those of all methods in dependent classes. Likewise, numerous tools and methods have been proposed to detect different kinds of code smells including CCFinder (Kamiya et al., 2002), SpIRIT (Vidal et al., 2015), DECOR (Moha et al., 2009), among others. While these studies have yielded encouraging outcomes in detecting common code smells in source code, further extensive research is required to reach a level of maturity.

An important challenge in rule- or heuristic-based approaches is the determination of appropriate metric thresholds. Defining these thresholds manually within smell detection algorithms represents a considerable difficulty for software engineers (Liu et al., 2016). Machine learning-based methods have drawn considerable interest from researchers as a solution to this challenge (Maiga et al., 2012b; Arcelli Fontana et al., 2016; Azadi et al., 2018). These methods involve training algorithms to capture and learn the complex relationships between source code features and code smell categories. Maiga et al. proposed SVMDetect based on the Support Vector Machine for code smell detection (Maiga et al., 2012b). They employed an input vector encompassing 60 structural metrics for each class to detect four well known code smells including God Class, Functional Decomposition, Spagheti Code and Swiss Army Knife. Fontana et al. investigated 16 different machine learning algorithms along with their boosting variants to detect four specific code smells: data class, god class, feature envy and long method (Arcelli Fontana et al., 2016). The experiments were carried out using independent metrics at various levels of granularity including method, class, package and project, showcasing the potential of machine learning-based approach comparing to rule-based ones. However, the ambiguity in detecting code smells from a given code snippet arises because various code smells may be associated with the same metrics (Sharma et al., 2021). This necessitates distinct characteristics to effectively differentiate one type of code smell from others.

### 2.3. Deep learning-based approaches for code smell detection

A growing body of research has shifted towards applying deep learning models to capture deeper features from source code and to learn the complex relationships between code snippets and code smell classes (Liu et al., 2021; Sharma et al., 2021; Ho et al., 2023). Building on this work, we introduced DeepSmells as an effective tool for code smell detection, leveraging diverse deep learning methods to identify patterns and semantic features in code snippets, outperforming previous approaches (Ho et al., 2023). Another research direction has focused on utilizing code embeddings, marking significant advancements in applying NLP techniques to software engineering (Von der Mosel et al., 2022). Recently, there has been a shift towards Transformer-based models (e.g., BERT (Devlin et al., 2019), RoBERTa (Liu et al., 2019), CodeBERT (Feng et al., 2020), among others). These models leverage large datasets and Transformer architectures with attention mechanisms to capture various distinctive features of code, a process commonly referred to as capturing semantics. Prior studies on code smell detection have successfully applied these models to automate tasks requiring an understanding of program semantics (Nguyen et al., 2024). Kovačevi'c et al. explored the effectiveness of code embedding models, such as *code2vec*, *code2seq*, and CuBERT, in comparison to traditional code metrics for detecting God Class and Long Method code smells (Kovačević et al., 2022).

Motivated by these advancements, we aim to leverage pre-trained code and language models for code smell detection, focusing on extracting additional features. We refer to these as *statistical semantic* features to distinguish them from literal semantics (e.g., operational semantics in programming languages) and to highlight their derivation from statistical patterns learned by these models. The concept of "statistical semantics" was previously introduced in the book Statistical Semantics (Sikström and Garcia, 2020), where it refers to understanding and modeling the meaning of words and concepts based on their statistical co-occurrence in large text corpora. This approach assumes that meaning emerges from distributional patterns in natural language rather than from predefined dictionaries or human-labeled definitions. Extending this concept to the domain of source code analysis, we apply the term "statistical semantics" to describe the extraction of meaning from source code using statistical patterns. Pre-trained models such as code2vec, CodeBERT, and CuBERT are trained on a large corpora of source code, leveraging these patterns to derive semantic representations of code snippets. Similar to natural language models, these code models capture both syntactic and semantic patterns, though their effectiveness varies. Recent research (Ma et al., 2024) reveals that different models exhibit distinct capabilities in encoding syntactic and semantic properties of code. Thus, we adopt "statistical semantics" to describe this feature, as it reflects the statistical nature of meaning extraction in source code.

To harness the benefits of both code metrics and statistical semantic features, we integrate these two types of features in our proposed approach, detailed in Section 3.

## 3. The proposed approach

Building upon our prior research, namely DeepSmells (Ho et al., 2023), where we concentrated on extracting unique features and patterns from source code, our current focus is on seamlessly integrating the strengths of code metrics-based and deep learning approaches. EnseSmells combines the automated generation of both *statistical semantic* and *structural* features from source code. The aim is to improve code smell detection accuracy and address the specific challenges encountered in previous methods.

As illustrated in Fig. 1, our approach comprises two primary components, including *Extractor* and *Classifier*. The *Extractor* component is designed to maximize the capture of relevant features from code snippets. Specifically, we focus on two types of features: *(i)* statistical semantic features, extending our previous work (Ho et al., 2023); *(ii)* structural features, automatically learned through a broad view of code metrics. The extracted features are then concatenated and fed into the *Classifier* component, where they are processed by an imbalanced DNN for classification. Once the *Classifier* is constructed, with weights and biases determined, it calculates a probability for each code snippet, indicating whether it exhibits a code smell or is clean.

### 3.1. Extractor

The *Extractor* includes two modules to capture distinct features of code snippets: code structure and statistical semantics.

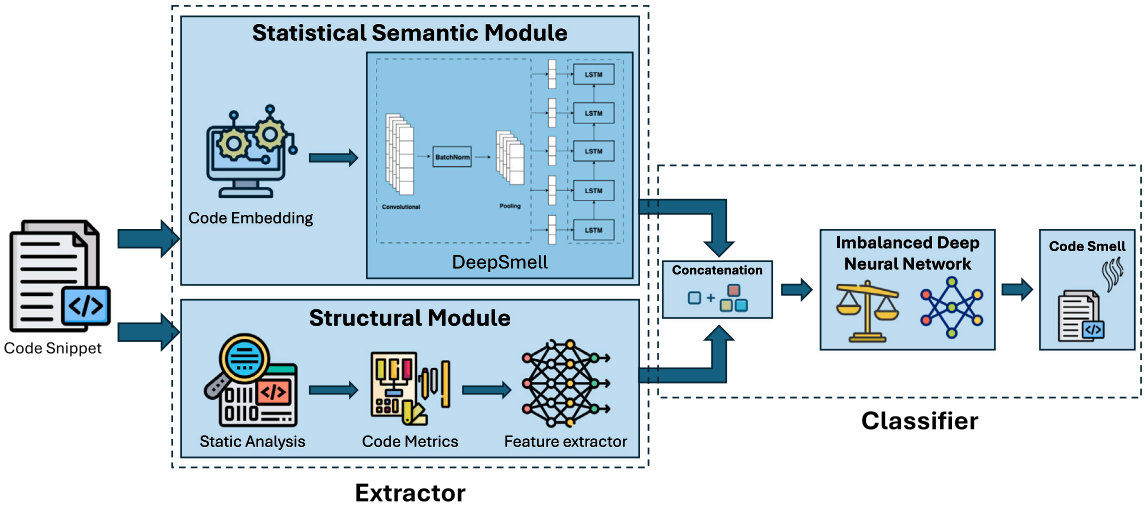


**Fig. 1.** The overall architecture of EnseSmells .

### 3.1.1. Statistical semantic module

This module is designed to capture statistical semantic features by leveraging code embedding models, as illustrated in Fig. 1. It operates in two stages: the encoding phase and the model-building phase. During the encoding phase, we have experimented with three pre-trained code and language models including CodeBERT, CuBERT and *code2vec*. Additionally, to evaluate the effectiveness of these models compared to a natural language embedding model, we have also employed the token indexing technique to represent a source code as a vector of token indices. Unlike simple index vectors that provide minimal contextual information about source code, pre-trained code embedding models are specifically designed to represent programming languages and are trained on extensive and complex datasets. These models extract statistical and semantic features from code snippets, enabling a more comprehensive and nuanced understanding of the underlying code semantics.

- ***Token-Indexing*** involves mapping tokens to integers to convert token sequences into integer vectors. To do so, we conducted several steps, including: *(i)* embedding code tokens with a tokenizer[2]; *(ii)* computing statistical information on the sample lengths, excluding those that deviate by more than one standard deviation from the mean; *(iii)* padding sequences with zeros to align with the longest input. This method was previously utilized in our prior work, DeepSmells (Ho et al., 2023).
- ***CodeBERT*** formats input as two segments joined by special tokens: $[CLS], w_1, w_2, \ldots, w_n, [SEP], c_1, c_2, \ldots, c_m, [EOS]$. The first segment generally contains natural language text, such as function comments, while the second segment consists of code tokens. The $[CLS]$ token at the beginning represents the entire sequence, aiding in classification tasks (Feng et al., 2020). CodeBERT produces two types of outputs: *(i)* contextual vector representations for each token in both the text and code segments, and *(ii)* an aggregated vector for the entire sequence represented by the $[CLS]$ token.
- ***CuBERT*** uses a custom Java tokenizer developed by Kanade et al. (2020) to preprocess code snippets before feeding them into the CuBERT model. CuBERT processes individual lines of code as input, transforming each line into a 1024-dimensional embedding derived from the special $[CLS]$ token (Devlin et al., 2019). To encode each method in the input source code, we compute embeddings for each line of code within the method body and then aggregate them by summing the resulting vectors. Similarly, for class-level embeddings, we apply this process across all methods in the class and calculate the sum of their embeddings. This method of summing embeddings has been shown to deliver consistently strong performance in recent studies (Kovačević et al., 2022).
- ***code2vec*** generates fixed-length embeddings from code snippets, using method bodies as inputs and producing corresponding tags as outputs (Alon et al., 2019). To generate the code embedding, we apply *code2vec* to represent each method as a 384-dimensional vector through the following steps: *(i)* extracting AST paths from each method using a Java path extractor, and *(ii)* feeding these paths into a pre-trained *code2vec* model, omitting the `softmax` layer to obtain the code vector. To compute the embedding for a class, we treat it as a collection of methods. The representation vector of a class is computed as the average of all its method embeddings. This approach was also be employed in previous studies (Compton et al., 2020).

To address code smell detection, we utilize our previous model, DeepSmells, which has demonstrated effectiveness in this task. In the model-building phase, the code embedding is processed through DeepSmells. The model initially processes the embedding through convolutional blocks, which automatically extract distinctive features. While stacking multiple convolutional blocks can capture increasingly complex and abstract features, including high-level token relationships, deeper networks often encounter a degradation problem where accuracy plateaus and then declines as the network depth grows (He et al., 2016). To mitigate this issue, we use two convolutional blocks, as outlined below.

- The first convolution block consists of a 1D-CNN (`torch.nn.Conv1d`) with 16 filters with a kernel size $k$ which specifies the length of the 1D convolution window. We experiment different values of $k$ (i.e., $k = 3, 4, 5, 6, 7$) to evaluate the effectiveness of the convolution layer. This layer employs a `ReLU` activation function, and the remaining parameters are kept as defaults, following by a 1D-Batch Normalization (`torch.nn.BatchNorm1d`) to accelerate the network training and to reduce internal covariate shift (Ioffe and Szegedy, 2015). The obtained feature map is then passed to a `MaxPooling` layer (`torch.nn.MaxPool1d`) by a factor of size three to reduce the spatial dimension.
- The second convolution block is similar to the first one, excepting that we use 32 filters.

[2] Available at: https://github.com/dspinellis/tokenizer.

**Table 1**
The class-level source code metrics.

| | | | |
|---|---|---|---|
| AnonymousClassesQty | AssignmentsQty | TotalFieldsQty | LogStatementsQty |
| InnerClassesQty | ReturnQty | CisibleFieldsQty | Modifiers |
| AbstractMethodsQty | LoopQty | VariablesQty | TCC |
| PrivateMethodsQty | UniqueWordsQty | TryCatchQty | LCC |
| ProtectedMethodsQty | PrivateFieldsQty | LambdasQty | WMC |
| PublicMethodsQty | ProtectedFieldsQty | ParenthesizedExpsQty | LOC |
| DefaultMethodsQty | PublicFieldsQty | NumbersQty | LCOM |
| StaticMethodsQty | DefaultFieldsQty | ComparisonsQty | RFC |
| FinalMethodsQty | StaticFieldsQty | MaxNestedBlocksQty | CBO |
| SynchronizedMethodsQty | FinalFieldsQty | MathOperationsQty | DIT |
| TotalMethodsQty | SynchronizedFieldsQty | StringLiteralsQty | NOSI |

**Table 2**
The method-level source code metrics.

| | | | |
|---|---|---|---|
| LOC | VariablesQty | TryCatchQty | AnonymousClassesQty |
| RFC | ParametersQty | ParenthesizedExpsQty | InnerClassesQty |
| CBO | MethodsInvokedQty | StringLiteralsQty | LambdasQty |
| WMC | MethodsInvokedLocalQty | NumbersQty | UniqueWordsQty |
| LogStatementsQty | MethodsInvokedIndirectLocalQty | AssignmentsQty | Modifiers |
| Constructor | LoopQty | MathOperationsQty | |
| ReturnsQty | ComparisonsQty | MaxNestedBlocksQty | |

Once the output from the convolutional neural networks has been obtained, it is fed into an LSTM network to preserve the meaning and context of the data. The `torch.nn.LSTM` function is used to configure the LSTM network, with the input size of each LSTM unit being based on the initial size of the initial embedding source code vector. The number of LSTM units in the network is also set to 32 to match the 32 filters in the second convolution block. Additionally, we adopt a Bi-LSTM architecture to capture long-term dependencies in sequential data, processing it in both forward and backward directions. This architecture consists of two separate LSTM layers: one processes the input sequence in the forward direction, while the other processes it in reverse. The hidden state at each time step is the concatenation of the forward and backward hidden states, enabling the model to incorporate information from both past and future contexts. We maintain the same configuration as before but set the `bidirectional` hyperparameter to `True`. The final hidden state from this network serves as the *statistical semantic features* of this module.

#### 3.1.2. Structural module

Previous studies have demonstrated the effectiveness of source code metrics in detecting code smells using rule- or heuristic-based approaches (Marinescu et al., 2010; Sales et al., 2013; Vidal et al., 2015). In this research, we also take into consideration source code metrics as structural features. However, instead of relying on threshold-based rules, we propose a deep learning model to capture non-linear relationships between code metric vectors and different categories of code smells (Fig. 1).

Code snippets are first analyzed to compute software metrics. Arcelli et al. synthesized and classified these metrics into different aspects, including size, complexity, cohesion, coupling, encapsulation, and inheritance (Arcelli Fontana et al., 2016). Tables 1 and 2 provide a summary of commonly used metrics in prior research, including 44 class-level metrics and 26 method-level metrics.[3] These metrics cover multiple dimensions of code structure, complexity, and object-oriented design, which have been demonstrated in numerous studies to have a strong correlation with different types of code smells (Marinescu, 2005; Lanza and Marinescu, 2006; Marinescu et al., 2010; Maiga et al., 2012b). We utilized the CK tool[4] to calculate the code metrics for each code snippet, based on whether it is at the class or method level. Before feeding the code metric vector into a deep learning model, some pre-processing steps have been conducted as follows.

- Categorical value metrics are encoded using label encoding.
- Metrics with constant values are removed as they are ineffective for making distinctions.
- In cases where a metric value could not be calculated for a specific code sample, we used k-Nearest Neighbors from the `scikit-learn` library[5] to impute the missing value. However, if the number of missing data samples for a given metric exceeds 5% of the total samples, we exclude this metric (Schafer, 1997; Bennett, 2001).
- Finally, all metric features are normalized using `StandardScaler` from the `scikit-learn` library.

These pre-processing steps ensure the data is properly formatted for the final stage, where we apply a deep neural network with a single adaptive layer. This layer is both efficient and convenient, as it dynamically adjusts the number of hidden nodes, unlike traditional neural networks, where parameters are tuned through iterative processes. The adaptive layer assigns higher weights to the importance of each metric and maps them into a lower-dimensional latent space (Xu et al., 2019). This approach effectively captures the key *structural features* necessary for code smell detection.

### 3.2. Classifier

The *Classifier* component combines both structural and statistical semantic features from the source code using an ensemble approach (Fig. 1).

In our previous work (Ho et al., 2023), we relied solely on the token-indexing technique to encode code snippets, paired with a simple deep learning model for classifying smelly and non-smelly source code. In this study, we aim to employ a more complex model architecture that learns a richer representation of code snippets, incorporating both statistical semantic and structural features for more effective code smell detection. Two embedding vectors resulting from the *Extractor* component are concatenated using the `torch.cat` operator of `PyTorch`. The ensemble vector is then fed as input into the Deep Imbalanced Neural network for classification. This network comprises two hidden layers using `ReLU` (Nair and Hinton, 2010) as the activation function. The output layer, designed for the binary classification task, consists of a single node with a `Sigmoid` activation function. The optimal number of hidden nodes is determined empirically, allowing the model to adapt to different datasets. Due to a significant class imbalance,

[3] Only the metric abbreviations are shown here; the full names and detailed descriptions are provided in Appendix.

[4] Available at: https://github.com/mauricioaniche/ck.

[5] Available at: https://scikit-learn.org/.

with the number of smelly code snippets being much lower than non-smelly ones, the model may develop a bias toward the majority class, despite the minority class being more important. To address this issue, we introduce a sensitivity weight into the binary cross-entropy loss function.

$$\mathcal{L}(x_i) = \beta \hat{y}_i log(y_i) + (1 - \hat{y}_i) log(1 - y_i) \quad (1)$$

where $\beta$ is the sensitive weight, $\hat{y}_i$ is the actual label of input $x_i$ and $y_i$ is the model's prediction for input $x_i$. This weight adjustment enables us to fine-tune the importance of the minority class in the classification process. To determine the optimal weight, we conduct a series of experiments comparing the performance of the binary cross-entropy with $\beta$ setting to 1 against weighted binary cross-entropy with varying weight values for $\beta$, such as 2, 4, 8, 12, 32, and 84. The results of these comparisons help us select the best hyper-parameter for our proposed method.

During the training and testing phase, we trained our model using a mini-batch size of 128 and set the learning rate to 0.025. Training was performed over 85 epochs using SGD to optimize the weighted loss function.

## 4. Experimental settings

We ran the evaluation with Google Colab Pro[6] using a virtual machine equipped with an Intel Xeon CPU with 2 vCPUs (virtual CPUs) and 13 GB of RAM. The virtual machine was equipped with a NVIDIA Tesla T4 GPU with 16 GB of VRAM to accelerate the deep learning computations. To evaluate EnseSmells , we re-ran all the baselines using the same benchmark datasets introduced in Section 4.2. To ensure a robust evaluation, we initially split the dataset into 80%:20% for training and testing, respectively. Moreover, to mitigate any potential bias and variance related to the test set resulting from the dataset split, stratified 5-fold cross-validation was applied, resembling traditional k-fold cross-validation but preserving the class distribution within each split.

### 4.1. Research questions

**RQ$_1$**: *How do software metrics impact the performance enhancement of* EnseSmells *?* The objective of this research question is to investigate the impact of adding structural modules to DeepSmells, forming EnseSmells . To achieve this, we compare EnseSmells and DeepSmells under the same configurations for the statistical semantic module.

**RQ$_2$**: *How do various embedding techniques impact each category of code smell?* The objective of this research question is to investigate code embedding techniques, each characterized by unique architectures that capture distinctive features associated with individual smells. To achieve this, we compare the performance of the aforementioned code embeddings in each architecture (EnseSmells and DeepSmells) under each category of code smell.

**RQ$_3$**: *How effective is* EnseSmells *compared with classical ML classifiers when utilizing only structural features?* The objective of this research question is to evaluate the effectiveness of EnseSmells , which combines various features within a complex architecture, compared to traditional machine learning classifiers that rely solely on software metrics proven effective for performing on tabular data (i.e., our structural features). To achieve this, we perform a grid search on classic classifiers with configurations outlined in Table 4 to determine the optimal choice and compare EnseSmells under its optimal configuration.

**RQ$_4$**: *How does* EnseSmells *perform compared to the baselines?* The objective of this research question is to demonstrate the performance of EnseSmells compared to baseline models. To achieve this, we conduct experiments with existing studies, including our work (Ho et al., 2023), the state-of-the-art, ML_CuBERT model (Kovačević et al., 2022) and three baseline models (Sharma et al., 2021). Three baselines are variants of an auto-encoder model designed to compress source code and learn salient information reflected in the reconstructed output, which are: *(i) AE-Dense:* An auto-encoder model that employs dense layers for both the encoder and decoder. *(ii) AE-CNN:* An auto-encoder model that employs two CNN networks, one for the encoder and another for the decoder. *(iii) AE-LSTM:* Similar to AE-CNN, but with CNNs replaced by LSTM networks. In all cases, the encoder and decoder layers are followed by a fully connected dense layer for classification.

[6] Available at: https://colab.research.google.com/.

**Table 3**
Statistics of the datasets.

| Smell | Smell alias | # Negative | # Positive |
|---|---|---|---|
| Long Method | LM | 1,993 | 243 |
| Feature Envy | FE | 2,126 | 64 |
| Data Class | DC | 1,838 | 282 |
| God Class | GC | 1,857 | 228 |

### 4.2. Benchmark dataset

The MLCQ dataset, originally made available by an existing study (Madeyski and Lewowski, 2020), was used in our experiments. The dataset comprises 14,739 reviews related to approximately 5000 Java code snippets gathered from 26 software developers. The reviews are categorized into four types of code smells, i.e., God Class and Data Class at the class level, and Feature Envy and Long Method at the method level.

The initial dataset labeling follows a structured approach: each reviewer assigns exactly one severity level for each code smell in a given code snippet, choosing from four levels: *none, minor, major*, and *critical*. We defined the task as a binary classification for each code snippet, labeling it as either *smelly* or *non-smelly*. However, since each code snippet has multiple reviews, we determine the final label using *majority vote* to reconcile the varying labels. Code samples are labeled as *smelly* if they receive more smelly reviews (i.e., minor, major, or critical severity) than non-smelly reviews (i.e., none severity). Conversely, they are classified as *non-smelly* if they have a greater number of non-smelly reviews. To maintain a clear classification, code snippets with an equal count of smelly and non-smelly reviews are subsequently removed from the dataset, as they do not provide a definitive classification and could introduce ambiguity into the binary labeling process.

The resulting dataset is summarized in Table 3. As shown, this classification problem exhibits a significant class imbalance, with the positive class (smelly instances) representing the minority class. The average imbalance rate across all datasets is 10.46%.

### 4.3. Evaluation metrics

Performance metrics commonly used in classification problems include Precision (P), Recall (R), F1-Score (F1), and Accuracy. However, these metrics have specific limitations. For instance, Accuracy often performs poorly on imbalanced datasets as it disproportionately favors the majority class. Additionally, Precision, Recall, and F1-Score focus on three quadrants of the confusion matrix (true positives, false positives, and false negatives) while excluding true negatives, which can lead to incomplete interpretations. To enable meaningful comparisons, we report these metrics while also including Matthews Correlation Coefficient (MCC), which provides a more balanced evaluation by considering all quadrants of the confusion matrix. Consistent with prior work (Sharma et al., 2021; Madeyski and Lewowski, 2023; Kovačević et al., 2022), these metrics are used to assess model performance.

**Precision, Recall, and F1-Score.** Confusion matrix is an effective means to evaluate any classifier with four possible outcomes including *true positive* (TP), *true negative* (TN), *false positive* (FP) and *false negative*

**Table 4**
The parameter settings of the used machine learning classifiers.

| Classifier | Hyperparameter settings (*scikit-learn* library) |
|---|---|
| Naive Bayes (NB) | `sklearn.naive_bayes.GaussianNB`: $var\_smoothing = [1e^{-9}, 1e^{-8}, 1e^{-7}, 1e^{-6}, 1e^{-5}]$ |
| Nearest Neighbor (NN) | `sklearn.neighbors.KNeighborsClassifier`: $n\_neighbors = [1, 3, 5, 7, 9]$ |
| Random Forest (RF) | `sklearn.ensemble.RandomForestClassifier`: $n\_estimators = [10, 50, 100, 200]$ $max\_depth = [None, 10, 20, 30]$ |
| Logistic Regression (LR) | `sklearn.linear_model.LogisticRegression`: $C\_values = [0.001, 0.01, 0.1, 1, 10, 100]$ |
| Classification and Regression Tree (CART) | `sklearn.tree.DecisionTreeClassifier`: $max\_depth\_values = [None, 10, 20, 30]$ $min\_samples\_split\_values = [2, 5, 10]$ $min\_samples\_leaf\_values = [1, 2, 4]$ $max\_features\_values = [auto, sqrt, log2]$ |
| Support Vector Machine (SVM) | `sklearn.svm.SVC`: $kernel\_function = GaussianRBF$ $gamma\_values = [0.001, 0.01, 0.1, 1, 10, 100]$ $C\_values = [0.001, 0.01, 0.1, 1, 10, 100]$ |
| Back Propagation Neural Networks (BP) | `sklearn.neural_network.MLPClassifier`: $hidden\_layer\_size = [16, 32, 64, (32, 16)]$ $learning\_rate\_init = [0.001, 0.01, 0.1]$ $max\_iter = [100, 200, 300]$ $tol\_err = [1e^{-4}, 1e^{-3}, 1e^{-2}]$ |

(FN). *Precision* measures how many of the positive predictions made by the model are actually correct: $P = \frac{TP}{TP+FP}$; *Recall* counts the number of positive cases in the dataset that model can identify: $R = \frac{TP}{TP+FN}$; *F1-Score* represents the harmonic mean of *precision* and *recall* of the prediction model: $F1 = \frac{2*P*R}{P+R}$.

**MCC.** The metric is used with imbalanced classification, measuring the correlation between the predicted and actual class, which lies in the range $[-1, 1]$, where 1 represents a perfect prediction, and $-1$ shows a perfect negative correlation; An MCC equal to 0 corresponds to a random prediction.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (2)$$

Notably, MCC and F1-Score both reflect overall performance (Sharma et al., 2021; Madeyski and Lewowski, 2023): MCC offers a balanced assessment by considering all quadrants of the confusion matrix, while F1-Score is widely used to capture the trade-off between precision and recall, offering insights into the classifier's ability to identify positive cases.

## 5. Experimental results and discussion

This section reports and analyzes the experimental results by answering the research questions introduced in Section 4.1.

### 5.1. $RQ_1$: How do software metrics impact the performance enhancement of EnseSmells ?

Fig. 2 compares the performance of EnseSmells and DeepSmells under identical structures and embedding techniques for the semantic module. Notably, EnseSmells incorporates an additional module, the structural model. Analyzing each embedding technique, starting with the *token indexing* method, we observe that EnseSmells outperforms DeepSmells across all four types of smells. Particularly, in the case of DC smell, EnseSmells exhibits significantly higher performance compared to DeepSmells. This trend is similarly observed for *CodeBERT*, with a substantial increase in performance for EnseSmells in GC and DC smells. As for *code2vec*, the same pattern persists, but the considerable outperformance is clearly evident across all smells.

Notably, *CuBERT* presents an exception in LM and GC smell, where EnseSmells slightly trails DeepSmells by less than 1% in both F1 and MCC metrics. However, in FE smell, EnseSmells outperforms DeepSmells by nearly 10% in both F1-Score and MCC. The most significant disparity is observed in DC smell, where EnseSmells showcases a substantial performance improvement of approximately 40% in both F1-Score and MCC compared to DeepSmells.

**Answer to $RQ_1$.** Incorporating structural modules enables EnseSmells to obtain a better prediction performance, highlighting the crucial role of software metrics. The performance improvement reaches up to 40%, depending on the pre-trained model and the type of code smell.

### 5.2. $RQ_2$: How do various embedding techniques impact each category of code smell?

In Fig. 3, we compare the performance of embedding techniques within the same architecture for both EnseSmells and DeepSmells. This study aims to assess the performance of each embedding technique across various smells, examining the performance patterns for each method in relation to each type of smell.

Starting with method-level smells, the use of *code2vec* in both EnseSmells and DeepSmells exhibits the lowest performance compared to other techniques. In contrast, *token indexing* emerges as a highly recommended choice due to its consistently superior performance. Notably, in this context, the consideration of *CuBERT* may be considered, given its potential impact, as it demonstrates only a slightly lower performance than *token indexing*.

Shifting focus to class-level smells, in the case of GC smell, both EnseSmells and DeepSmells employing the *token indexing* embedding technique show the best results compared to other techniques. Transitioning to DC smell, the use of *code2vec* delivers the highest performance, especially in DeepSmells, where it exhibits significant outperformance.

**Answer to $RQ_2$.** Each pre-trained programming language model possesses a distinct architecture, suitable for addressing specific code smells. Specifically, *token-indexing* is effective for *Long Method* and *God Class*, *CuBERT* is appropriate for *Feature Envy*, and *code2vec* is well-suited for *Data Class*, showcasing the suitability of each model for different types of code smells.

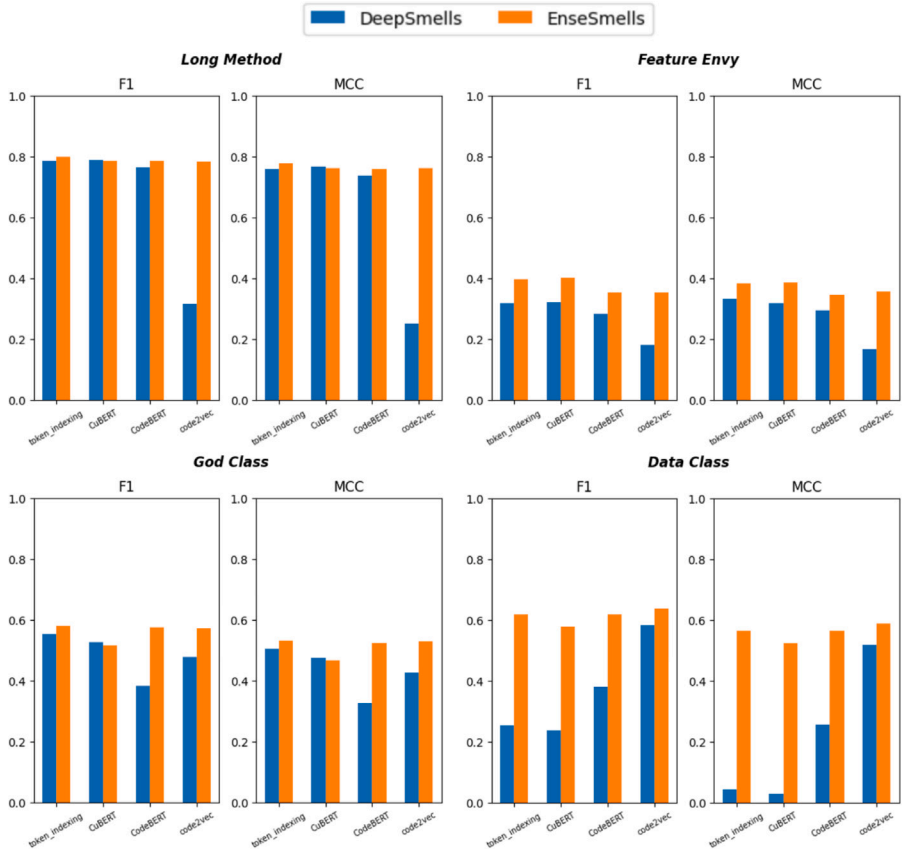


**Fig. 2.** Performance of EnseSmells and DeepSmells across different embedding techniques.

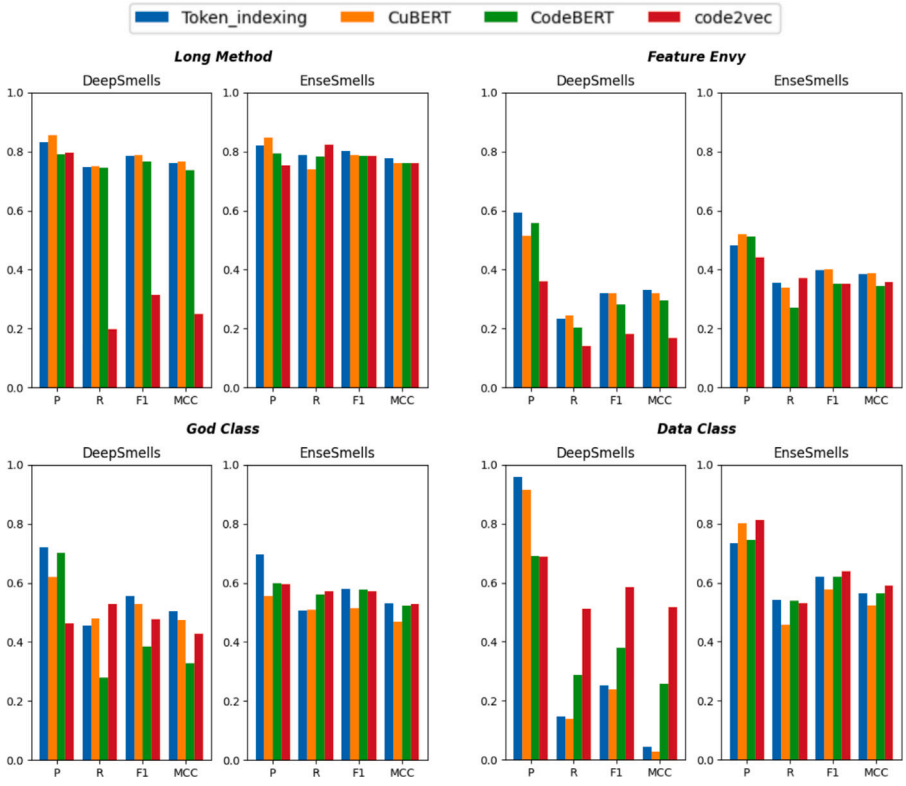


**Fig. 3.** Performance of EnseSmells and DeepSmells in various embedding techniques.

**Table 5**
Comparing the performance of seven classifiers on software metrics with EnseSmells model.

| Smell | Metric | Machine learning classifier | | | | | | | EnseSmells |
|---|---|---|---|---|---|---|---|---|---|
| | | NB | NN | RF | LR | CART | SVM | BP | |
| LM | P | 0.5698 | 0.8048 | 0.8137 | 0.8168 | 0.7595 | 0.7560 | **0.8477** | 0.8215 |
| | R | **0.8196** | 0.5310 | 0.6821 | 0.6535 | 0.6429 | 0.6901 | 0.6857 | 0.7873 |
| | F1 | 0.6715 | 0.6360 | 0.7416 | 0.7248 | 0.6943 | 0.7199 | 0.7566 | **0.8016** |
| | MCC | 0.6339 | 0.6173 | 0.7150 | 0.6994 | 0.6625 | 0.6874 | 0.7347 | **0.7780** |
| FE | P | 0.1162 | 0.1265 | 0.3000 | 0.2000 | 0.4611 | 0.2246 | 0.2126 | **0.5215** |
| | R | **0.7440** | 0.1220 | 0.0462 | 0.0736 | 0.2099 | 0.1220 | 0.2451 | 0.3393 |
| | F1 | 0.1993 | 0.1226 | 0.0800 | 0.1076 | 0.2784 | 0.1545 | 0.2181 | **0.4025** |
| | MCC | 0.2453 | 0.0987 | 0.1084 | 0.1068 | 0.2911 | 0.1441 | 0.1978 | **0.3868** |
| GC | P | 0.1620 | 0.5415 | **0.7442** | 0.6231 | 0.5296 | 0.6109 | 0.5836 | 0.6220 |
| | R | **0.8308** | 0.3943 | 0.3662 | 0.2744 | 0.4505 | 0.4011 | 0.5494 | 0.5867 |
| | F1 | 0.2709 | 0.4552 | 0.4895 | 0.3760 | 0.4845 | 0.4809 | 0.5656 | **0.5851** |
| | MCC | 0.1280 | 0.3944 | 0.4761 | 0.3570 | 0.4166 | 0.4339 | 0.5028 | **0.5429** |
| DC | P | 0.1644 | 0.5415 | 0.6849 | 0.5980 | 0.5525 | 0.6343 | 0.5839 | **0.8123** |
| | R | **0.8205** | 0.3943 | 0.3979 | 0.2816 | 0.4263 | 0.3838 | 0.5393 | 0.5314 |
| | F1 | 0.2734 | 0.4552 | 0.5019 | 0.3803 | 0.4808 | 0.4772 | 0.5573 | **0.6393** |
| | MCC | 0.1291 | 0.3944 | 0.4703 | 0.3541 | 0.4184 | 0.4366 | 0.4971 | **0.5907** |

### 5.3. $RQ_3$: *How effective is* EnseSmells *compared with classical ml classifiers when utilizing only structural features?*

Table 5 illustrates the performance of EnseSmells in comparison to seven classical machine learning models, which include *Naive Nayes (NB), Nearest Neighbor (NN), Random Forest (RF), Logistic Regression (LR), Classification and Regression Tree (CART), Back Propagation neural networks (BP),* and *Support Vector Machine (SVM)*. These machine learning models exclusively utilize software metrics. The results indicate that EnseSmells outperforms all other models, demonstrating the highest performance across all evaluated models.

Specifically, for the LM code smell, the F1-Score and MCC of EnseSmells are 80.16% and 77.80% higher than those of BP, with improvements of 4.50% and 4.33%, respectively. Notably, the precision value of BP is 2.62% higher than EnseSmells but comes at the cost of a significant reduction in recall by 10.16%. Additionally, while the recall value of NB achieves the highest at 81.96%, it is higher than EnseSmells ; however, the precision, F1-Score, and MCC values are significantly lower by 25.17%, 13.01%, and 14.41%, respectively.

Regarding the FE code smell, when comparing EnseSmells to CART, which is considered the best machine learning classifier for this type of code smell, EnseSmells outperforms in all four evaluation metrics, showing improvements of 6.04%, 12.94%, 12.41%, and 9.57% in precision, recall, F1-Score, and MCC, respectively. The scenario is significantly different when comparing NB and EnseSmells . While NB achieves the highest precision value among these models at 74.40%, its recall value is only 11.62%, highlighting its weakness in handling the high imbalanced dataset.

Turning to GC code smell, while BP exhibits the best overall performance among these ML classifiers, when compared to EnseSmells , our model demonstrates outperformance in all four metrics. Furthermore, NB and RF are two models with the highest values in recall and precision, but they also exhibit significant reductions in precision and recall, respectively. Consequently, this leads to a significant decrease in both F1-Score and MCC, ranging from 6.68% to 41.49%.

As for the remaining code smell, DC, when comparing EnseSmells to BP, which demonstrates good performance across almost all types of code smells including this one, our model reveals significant differences in precision, F1-Score, and MCC, corresponding to 22.84%, 8.2%, and 9.36%. Additionally, when comparing EnseSmells to NB, which boasts the highest recall value at 82.05%, the precision value of only 16.44% highlights its weakness in handling highly imbalanced datasets leading to a remarkable decrease in both F1-Score and MCC, with a deviation of 36.59% and 46.16% lower than EnseSmells , respectively.

**Table 6**
Performance of different embedding techniques in EnseSmells architecture under optimal configurations.

| Smell | Model | Evaluation metric | | | |
|---|---|---|---|---|---|
| | | P | R | F1 | MCC |
| LM | $\text{EnseSmells}_{token-indexing}$ | 0.8215 | 0.7873 | **0.8016** | **0.7780** |
| | $\text{EnseSmells}_{CuBERT}$ | **0.8482** | 0.7391 | 0.7877 | 0.7616 |
| | $\text{EnseSmells}_{CodeBERT}$ | 0.7938 | 0.7824 | 0.7857 | 0.7611 |
| | $\text{EnseSmells}_{code2vec}$ | 0.7526 | **0.8242** | 0.7841 | 0.7618 |
| FE | $\text{EnseSmells}_{token-indexing}$ | 0.4821 | 0.3554 | 0.3982 | 0.3849 |
| | $\text{EnseSmells}_{CuBERT}$ | **0.5215** | **0.3393** | **0.4025** | **0.3868** |
| | $\text{EnseSmells}_{CodeBERT}$ | 0.5128 | 0.2718 | 0.3527 | 0.3449 |
| | $\text{EnseSmells}_{code2vec}$ | 0.4423 | 0.3727 | 0.3538 | 0.3568 |
| GC | $\text{EnseSmells}_{token-indexing}$ | **0.6962** | 0.5077 | **0.5801** | **0.5318** |
| | $\text{EnseSmells}_{CuBERT}$ | 0.5565 | 0.5105 | 0.5153 | 0.4675 |
| | $\text{EnseSmells}_{CodeBERT}$ | 0.5993 | 0.5607 | 0.5764 | 0.5241 |
| | $\text{EnseSmells}_{code2vec}$ | 0.5947 | **0.5724** | 0.5729 | 0.5288 |
| DC | $\text{EnseSmells}_{token-indexing}$ | 0.7340 | **0.5427** | 0.6203 | 0.5638 |
| | $\text{EnseSmells}_{CuBERT}$ | 0.8027 | 0.4567 | 0.5772 | 0.5238 |
| | $\text{EnseSmells}_{CodeBERT}$ | 0.7447 | 0.5390 | 0.6199 | 0.5648 |
| | $\text{EnseSmells}_{code2vec}$ | **0.8123** | 0.5314 | **0.6393** | **0.5907** |

**Answer to $RQ_3$.** EnseSmells outperforms all traditional ML classifiers that build prediction models using *structural* features. In terms of F1-Score, EnseSmells achieves superior performance ranging from 1.95% to 12.41% compared to the best ML classifier for each code smell. For MCC, the improvement ranges from 4.01% to 14.15%.

### 5.4. $RQ_4$: *How does* EnseSmells *perform compared to the baselines?*

Tables 6 and 7 present the optimal performance of the EnseSmells and DeepSmells models, respectively, evaluated using different code embedding techniques, including *token-indexing, CuBERT, CodeBERT*, and *code2vec*, each under their respective optimal network configurations.

Table 8 demonstrates the performance of DeepSmells and EnseSmells under their best configurations, showing that EnseSmells outperforms DeepSmells. For the LM code smell, EnseSmells achieves improvements of 1.28% in F1-Score and 1.07% in MCC. Although DeepSmells exhibits a 3.45% higher precision, this comes at the expense of a 3.83% lower recall. Similarly, for the GC code smell, EnseSmells surpasses DeepSmells by 3.09% in F1-Score and 3.8% in MCC. While DeepSmells achieves a higher precision by 9.73%, its recall is 13.09% lower compared to EnseSmells .

For the remaining code smells, EnseSmells shows superior performance across all evaluation metrics. Notably, EnseSmells outperforms

**Table 7**
Performance of different embedding techniques in DeepSmells architecture under optimal configurations.

| Smell | Model | Metric | | | |
|---|---|---|---|---|---|
| | | P | R | F1 | MCC |
| LM | $\text{DeepSmells}_{token-indexing}$ (Ho et al., 2023) | 0.8327 | 0.7470 | 0.7865 | 0.7607 |
| | $\text{DeepSmells}_{CuBERT}$ | **0.8560** | **0.7490** | **0.7888** | **0.7673** |
| | $\text{DeepSmells}_{CodeBERT}$ | 0.7903 | 0.7448 | 0.7661 | 0.7373 |
| | $\text{DeepSmells}_{code2vec}$ | 0.7974 | 0.1982 | 0.3152 | 0.2509 |
| FE | $\text{DeepSmells}_{token-indexing}$ (Ho et al., 2023) | **0.5936** | 0.2322 | 0.3198 | **0.3318** |
| | $\text{DeepSmells}_{CuBERT}$ | 0.5143 | **0.2439** | **0.3203** | 0.3201 |
| | $\text{DeepSmells}_{CodeBERT}$ | 0.5590 | 0.2048 | 0.2833 | 0.2947 |
| | $\text{DeepSmells}_{code2vec}$ | 0.3603 | 0.1422 | 0.1815 | 0.1683 |
| GC | $\text{DeepSmells}_{token-indexing}$ (Ho et al., 2023) | **0.7193** | 0.4559 | **0.5542** | **0.5049** |
| | $\text{DeepSmells}_{CuBERT}$ | 0.6191 | **0.4809** | 0.5277 | 0.4747 |
| | $\text{DeepSmells}_{CodeBERT}$ | 0.7007 | 0.2798 | 0.3845 | 0.3270 |
| | $\text{DeepSmells}_{code2vec}$ | 0.4647 | 0.5293 | 0.4780 | 0.4272 |
| DC | $\text{DeepSmells}_{token-indexing}$ (Ho et al., 2023) | **0.9579** | 0.1476 | 0.2531 | 0.0439 |
| | $\text{DeepSmells}_{CuBERT}$ | 0.9158 | 0.1389 | 0.2382 | 0.0281 |
| | $\text{DeepSmells}_{CodeBERT}$ | 0.6916 | 0.2868 | 0.3802 | 0.2573 |
| | $\text{DeepSmells}_{code2vec}$ | 0.6883 | **0.5132** | **0.5839** | **0.5185** |

**Table 8**
Comparison of DeepSmells and EnseSmells models with optimal configurations.

| Smell | Model | Metric | | | |
|---|---|---|---|---|---|
| | | P | R | F1 | MCC |
| LM | DeepSmells | **0.8560** | 0.7490 | 0.7888 | 0.7673 |
| | EnseSmells | 0.8215 | **0.7873** | **0.8016** | **0.7780** |
| FE | DeepSmells | 0.5936 | 0.2322 | 0.3198 | 0.3318 |
| | EnseSmells | **0.5215** | **0.3393** | **0.4025** | **0.3868** |
| GC | DeepSmells | **0.7193** | 0.4559 | 0.5542 | 0.5049 |
| | EnseSmells | 0.6220 | **0.5867** | **0.5851** | **0.5429** |
| DC | DeepSmells | 0.6883 | 0.5132 | 0.5839 | 0.5185 |
| | EnseSmells | **0.8123** | **0.5314** | **0.6393** | **0.5907** |

DeepSmells by 8.27% in F1-Score and 5.5% in MCC for the FE code smell, and by 7.22% in F1-Score and 5.5% in MCC for the DC code smell.

Table 9 presents a comparison between EnseSmells and the state-of-the-art approach, *ML_CuBERT* (Kovačević et al., 2022), and existing approaches that include various auto-encoder variants (Sharma et al., 2021). The table demonstrates that our proposed model outperforms the other models by all evaluation metrics. Especially for LM, EnseSmells exhibits the lowest recall value at 78.73%, which is lower than the others by 2.69% to 6.07%. However, it boasts the highest precision value, surpassing the others by 7.83% to 15.35%. As a result, the overall performance demonstrates the highest F1-Score and MCC, with significant differences compared to the other models. Regarding the FE code smell, following a similar pattern LM, EnseSmells showcases an overall outperformance, with substantial differences compared to ML_CuBERT and the variants of autoencoder. The F1-Score varies from 5.25% to 19.04%, while the MCC differs from 3.59% to 11.85%.

In relation to the GC code smell, compared to AE-CNN, which shows the best performance among the baseline models with values of 53.34% (precision), 66.22% (recall), 57.75% (F1-Score), and 53.133% (MCC), EnseSmells exhibits a slightly higher in F1-Score and MCC by 0.76% and 1.18%, respectively. Notably, the precision value of EnseSmells is 8.86% higher than AE-CNN, while the recall value is only 7.55% lower.

The remaining code smell is the DC. The autoencoder variants exhibit a weakness in dealing with highly imbalanced datasets, showcasing very high recall values of up to 91.84%, but the precision values hover around 15%, resulting in the lowest overall performance. Moreover, EnseSmells outperforms ML_CuBERT in all four evaluation metrics by 31.7% (precision), 24.61% (recall), 27.78% (F1-Score), and 28.26% (MCC).

**Answer to $RQ_4$.** EnseSmells demonstrates superior prediction performance for all four types of code smells compared to the state-of-the-art baselines. It gains a better accuracy compared to that of DeepSmells, with improvements ranging from approximately 5% to 10%. The MCC (overall performance) surpasses ML_CuBERT by approximately 5.98% to 28.26% and the best Autoencoder variants by 1.18% to 48.39% across the code smell categories.

## 6. Threat to validity

We anticipate that there are the following threats to the validity of our findings.

- **Internal validity.** This concerns the extent to which our evaluation reflects real-world scenarios. In our work, we used existing datasets (Madeyski and Lewowski, 2020) curated and classified by human experts. Additionally, we assigned labels to each code snippet based on majority-based approach. However, these labels may not be universally accepted, which could impact the overall dataset quality and, consequently, the accuracy of our predictions. Furthermore, the dataset also exhibits class imbalance, mirroring real-world distributions but potentially biasing the model towards more common classes. To address this, we applied class-weight adjustments during training and used metrics designed for imbalanced data, ensuring a fair evaluation across all classes.
- **External validity.** This concerns the generalizability of the findings beyond the scope of this study. We attempted to mitigate threats by evaluating with different experimental configurations to simulate real-world scenarios. The findings of our work might apply only to the considered datasets. For other datasets, additional empirical evidence is required before reaching a final conclusion.
- **Construct validity.** This dimension relates to how we set up our experiments to compare EnseSmells with the baseline models. To ensure a fair comparison, we used the original implementations provided by the authors of the baseline models, maintaining their internal structures. Furthermore, our method involved measuring various software metrics using well-accepted open-source tools in our field. It is crucial to note that our study focused on the extensive MLCQ dataset, comprising a substantial number of projects (792). In such a vast dataset, there is a possibility of errors in metric calculations due to parsing issues. During our error analysis, our domain expert identified a few instances where unique word counts were miscalculated. Nevertheless, we have confidence that the use of established tools widely recognized in the research community helps minimize the impact of these potential errors.

**Table 9**
Comparison with baseline models.

| Smell | Model | Metric | | | |
|---|---|---|---|---|---|
| | | P | R | F1 | MCC |
| LM | ML_CuBERT (Kovačević et al., 2022) | 0.6933 | 0.8142 | 0.7481 | 0.7182 |
| | AE-Dense (Sharma et al., 2021) | 0.7432 | 0.8274 | 0.7804 | 0.7548 |
| | AE-CNN (Sharma et al., 2021) | 0.7363 | 0.8315 | 0.7728 | 0.7500 |
| | AE-LSTM (Sharma et al., 2021) | 0.6680 | **0.8480** | 0.7471 | 0.7187 |
| | EnseSmells | **0.8215** | 0.7873 | **0.8016** | **0.7780** |
| FE | ML_CuBERT (Kovačević et al., 2022) | 0.1223 | **0.8033** | 0.2121 | 0.2683 |
| | AE-Dense (Sharma et al., 2021) | 0.2325 | 0.6295 | 0.3356 | 0.3509 |
| | AE-CNN (Sharma et al., 2021) | 0.2704 | 0.5333 | 0.3500 | 0.3478 |
| | AE-LSTM (Sharma et al., 2021) | 0.1364 | 0.7987 | 0.2329 | 0.2888 |
| | EnseSmells | **0.5215** | 0.3393 | **0.4025** | **0.3868** |
| GC | ML_CuBERT (Kovačević et al., 2022) | 0.4580 | 0.5763 | 0.5092 | 0.4492 |
| | AE-Dense (Sharma et al., 2021) | 0.5495 | 0.6444 | 0.5751 | 0.5306 |
| | AE-CNN (Sharma et al., 2021) | 0.5334 | **0.6622** | 0.5775 | 0.5311 |
| | AE-LSTM (Sharma et al., 2021) | 0.5227 | 0.6490 | 0.5706 | 0.5200 |
| | EnseSmells | **0.6220** | 0.5867 | **0.5851** | **0.5429** |
| DC | ML_CuBERT (Kovačević et al., 2022) | 0.4953 | 0.2853 | 0.3615 | 0.3081 |
| | AE-Dense (Sharma et al., 2021) | 0.1559 | 0.8576 | 0.2613 | 0.1068 |
| | AE-CNN (Sharma et al., 2021) | 0.1489 | **0.9184** | 0.2561 | 0.0977 |
| | AE-LSTM (Sharma et al., 2021) | 0.1532 | 0.8328 | 0.2567 | 0.0922 |
| | EnseSmells | **0.8123** | 0.5314 | **0.6393** | **0.5907** |

**Table A.10**
The abbreviation and software metric description for method level.

| Abbreviation | Software metric description |
|---|---|
| LOC | Lines of code (excluding empty lines and comments). |
| CBO | Coupling between objects: measures the number of classes or methods a given method depends on, considering parameters, return types, local variables, and method calls, excluding dependencies on Java standard libraries (e.g., java.lang.String). |
| WMC | Weighted method complexity: counts the number of branch instructions (e.g., decision points like if, for, while) within methods, based on McCabe's complexity. |
| RFC | Response for a class: counts the unique method invocations within a method. |
| modifiers | The modifiers of methods, including public, private, abstract, protected, and native. |
| constructor | Indicates whether the method is a constructor. |
| logStatementsQty | The number of log statements. |
| returnsQty | The number of return instructions. |
| variablesQty | The number of variables declared. |
| parametersQty | The number of parameters. |
| methodsInvokedQty | The total number of methods directly invoked, including local and indirect local invocations. |
| methodsInvokedLocalQty | The number of methods invoked locally within the method. |
| methodsInvokedIndirectLocalQty | The number of indirectly invoked methods within the method. |
| loopQty | The number of loops (e.g., for, while, do-while) within methods. |
| comparisonsQty | The number of comparison operations (e.g., ==, !=, <, >) within methods. |
| tryCatchQty | The number of try/catch blocks within methods. |
| parenthesizedExpsQty | The number of expressions inside parentheses within methods. |
| stringLiteralsQty | The number of string literals used in methods. |
| numbersQty | The number of numerical literals (e.g., int, long, double, float) used in methods. |
| assignmentsQty | The number of assignment operations within methods. |
| mathOperationsQty | The number of mathematical operations (e.g., +, -, *, /) within methods. |
| maxNestedBlocksQty | The maximum depth of nested blocks (e.g., loops, conditionals) within methods. |
| anonymousClassesQty | The number of anonymous classes used within methods. |
| innerClassesQty | The number of inner classes defined within methods. |
| lambdasQty | The number of lambda expressions used in methods. |
| uniqueWordsQty | The number of unique words (e.g., identifiers, keywords) used in methods. |

## 7. Conclusion

This paper introduces EnseSmells , an innovative approach for detecting four types of code smells — two at the method level (*Long Method* and *Feature Envy*) and two at the class level (*God Class* and *Data Class*). Our methodology leverages a unique ensemble of two feature types: *semantic features* derived from novel pre-trained programming language models and *structural features* obtained through object-oriented metrics extracted by a static analysis tool. Building upon the foundation of our sophisticated model in a previous work, DeepSmells, we conduct extensive experiments to address five research questions and assess the effectiveness of EnseSmells . The results demonstrate that EnseSmells outperforms existing methods, achieving state-of-the-art performance on the MLCQ dataset with improvements ranging from 5.98% to 28.26%, depending on the type of code smell. Our future work involves expanding the application of this approach to additional code smells and identifying areas for improvement to further enhance its capabilities.

**Table A.11**
The abbreviation and software metric description for class level.

| Abbreviation | Software metric description |
|---|---|
| CBO | Coupling between objects measures the number of dependencies a class has, considering all types used within the class (fields, method return types, variables, etc.), excluding dependencies on Java standard libraries (e.g., java.lang.String). |
| DIT | Depth inheritance tree counts the number of parent classes a class has. The minimum DIT is 1 (all classes inherit `java.lang.Object`). If a class depends on an external dependency (e.g., a JAR file), the depth is counted as 2. |
| WMC | Weight method class measures the complexity of a class by counting the number of branch instructions (e.g., decision points like if, for, while) within its methods, based on McCabe's complexity. |
| TCC | Tight class cohesion measures the cohesion of a class (within a range of 0 to 1) based on direct connections between visible methods, where methods or their invocation trees access the same class variable. |
| LCC | Loose class cohesion extends TCC by including indirect connections between visible classes, ensuring LCC $\geq$ TCC. |
| LCOM | Lack of cohesion of methods measures the cohesion within a class. Higher cohesion indicates a well-structured class, while lower cohesion suggests the class may handle multiple responsibilities. |
| LOC | Lines of code (excluding empty lines and comments). |
| NOSI | The number of static invocations counts the number of invocations to static methods. |
| RFC | Response for a class counts unique method invocations in a class. |
| abstractMethodsQty | The number of abstract methods. |
| anonymousClassesQty | The number of anonymous classes. |
| assignmentsQty | The number of assignments. |
| comparisonsQty | The number of comparisons. |
| defaultFieldsQty | The number of default fields. |
| defaultMethodsQty | The number of default methods. |
| finalFieldsQty | The number of final fields. |
| finalMethodsQty | The number of final methods. |
| innerClassesQty | The number of inner classes. |
| lambdasQty | The number of lambda expressions. |
| logStatementsQty | The number of log statements. |
| loopQty | The number of loops (i.e., for, while, do while). |
| mathOperationsQty | The number of math operations. |
| maxNestedBlocksQty | The number of maximum nested blocks. |
| modifiers | The modifiers of classes include public, abstract, private, protected, or native. |
| privateFieldsQty | The number of private fields. |
| privateMethodsQty | The number of private methods. |
| protectedFieldsQty | The number of protected fields. |
| protectedMethodsQty | The number of protected methods. |
| publicFieldsQty | The number of public fields. |
| publicMethodsQty | The number of public methods. |
| returnQty | The number of return instructions. |
| staticFieldsQty | The number of static fields. |
| staticMethodsQty | The number of static methods. |
| stringLiteralsQty | The number of string literals. |
| synchronizedFieldsQty | The number of synchronized fields. |
| synchronizedMethodsQty | The number of synchronized methods. |
| totalFieldsQty | The total number of fields. |
| totalMethodsQty | The total number of methods. |
| tryCatchQty | The number of try/catch blocks. |
| visibleFieldsQty | The number of visible fields. |
| numbersQty | The number of numerical literals (e.g., int, long, double, float). |
| parenthesizedExpsQty | The number of expressions inside parenthesis. |
| uniqueWordsQty | The number of unique words. |
| variablesQty | The number of variables. |

## CRediT authorship contribution statement

**Anh Ho:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Data curation, Conceptualization. **Anh M.T. Bui:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Formal analysis, Conceptualization. **Phuong T. Nguyen:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation. **Amleto Di Salle:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology. **Bach Le:** Writing – review & editing, Validation, Supervision, Methodology.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

Our research has been funded by Hanoi University of Science and Technology (HUST), Vietnam under project number T2023-PC-002. This work has been supported by the COmmunity-Based Organized Littering (COBOL) national research project funded by the MUR under the PRIN 2022 PNRR program (nr. P20224K9EK), by the "Progetto PE 0000020 CHANGES, PNRR Missione 4 Componente 2 Investimento 1.3" funded by EU - NextGenerationEU, by the European HORIZON-KDT-JU research project MATISSE "Model-based engineering of Digital Twins for early verification and validation of Industrial Systems", HORIZON-KDT-JU-2023-2-RIA, Proposal number: 101140216-2, KDT232RIA_00017, and by the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 - VITALITY – CUP: D13C21000430001. We acknowledge the Italian "PRIN 2022" project TRex-SE: *"Trustworthy Recommenders for Software Engineers",* grant n. 2022LKJWHC. We thank the anonymous reviewers

for their useful comments and suggestions that helped us improve our manuscript.

## Appendix. The information of software metrics at method and class level

See Tables A.10 and A.11.

## Data availability

The authors do not have permission to share data.

## References


Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. code2vec: Learning distributed representations of code. In: Proceedings of the ACM on Programming Languages, Vol. 3. POPL, pp. 1–29. http://dx.doi.org/10.1145/3290353.

Arcelli Fontana, F., Mäntylä, M.V., Zanoni, M., Marino, A., 2016. Comparing and experimenting machine learning techniques for code smell detection. Empir. Softw. Eng. 21, 1143–1191. http://dx.doi.org/10.1007/S10664-015-9378-4.

Azadi, U., Fontana, F.A., Zanoni, M., 2018. Machine learning based code smell detection through wekanose. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings. ICSE 2018, Gothenburg, Sweden, May 27 – June 03 2018, ACM, pp. 288–289. http://dx.doi.org/10.1145/3183440.3194974.

Azeem, M.I., Palomba, F., Shi, L., Wang, Q., 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Inf. Softw. Technol. 108, 115–138. http://dx.doi.org/10.1016/J.INFSOF.2018.12.009.

Bennett, D.A., 2001. How can i deal with missing data in my study? Aust. N. Z. J. Public Health 25 (5), 464–469. http://dx.doi.org/10.1111/j.1467-842X.2001.tb00294.x, URL https://www.sciencedirect.com/science/article/pii/S1326020023036488.

Compton, R., Frank, E., Patros, P., Koay, A.M.Y., 2020. Embedding Java classes with code2vec: Improvements from variable obfuscation. In: MSR '20: 17th International Conference on Mining Software Repositories. Seoul, Republic of Korea, 29–30 June, 2020, ACM, pp. 243–253. http://dx.doi.org/10.1145/3379597.3387445.

Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., Love, T., 1979. Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. IEEE Trans. Softw. Eng. 5 (2), 96–104. http://dx.doi.org/10.1109/TSE.1979.234165.

Das, A.K., Yadav, S., Dhal, S., 2019. Detecting code smells using deep learning. In: TENCON 2019-2019 IEEE Region 10 Conference. TENCON, IEEE, pp. 2081–2086. http://dx.doi.org/10.1109/TENCON.2019.8929628.

Devlin, J., Chang, M., Lee, K., Toutanova, K., 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2–7, 2019, Volume 1 (Long and Short Papers). Association for Computational Linguistics, pp. 4171–4186. http://dx.doi.org/10.18653/V1/N19-1423.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. Codebert: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics. EMNLP 2020, Online Event, 16-20 November 2020, In: Findings of ACL, vol. EMNLP 2020, Association for Computational Linguistics, pp. 1536–1547. http://dx.doi.org/10.18653/V1/2020.FINDINGS-EMNLP.139.

Fontana, F.A., Zanoni, M., 2017. Code smell severity classification using machine learning techniques. Knowl.-Based Syst. 128, 43–58. http://dx.doi.org/10.1016/J.KNOSYS.2017.04.014.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co. Inc., USA, ISBN: 0201485672, URL http://martinfowler.com/books/refactoring.html.

Garcia, J., Popescu, D., Edwards, G., Medvidovic, N., 2009. Identifying architectural bad smells. In: 13th European Conference on Software Maintenance and Reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems. Kaiserslautern, Germany, 24–27 March 2009, IEEE Computer Society, pp. 255–258. http://dx.doi.org/10.1109/CSMR.2009.59.

Hadj-Kacem, M., Bouassida, N., 2018. A hybrid approach to detect code smells using deep learning. In: Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering. ENASE 2018, Funchal, Madeira, Portugal, March 23–24, 2018, SciTePress, pp. 137–146. http://dx.doi.org/10.5220/0006709801370146.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2016, Las Vegas, NV, USA, June 27–30, 2016, IEEE Computer Society, pp. 770–778. http://dx.doi.org/10.1109/CVPR.2016.90.

Ho, A., Bui, A.M.T., Nguyen, P.T., Di Salle, A., 2023. Fusion of deep convolutional and LSTM recurrent neural networks for automated detection of code smells. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering. EASE 2023, Oulu, Finland, June 14–16, 2023, ACM, pp. 229–234. http://dx.doi.org/10.1145/3593434.3593476.

Ho, A., Hai, N.N., Anh, B.T.M., 2022. Combining deep learning and kernel PCA for software defect prediction. In: The 11th International Symposium on Information and Communication Technology. SoICT 2022, Hanoi, Vietnam, December 1–3, 2022, ACM, pp. 360–367. http://dx.doi.org/10.1145/3568562.3568587.

Ioffe, S., Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Bach, F.R., Blei, D.M. (Eds.), Proceedings of the 32nd International Conference on Machine Learning. ICML 2015, Lille, France, 6–11 July 2015, In: JMLR Workshop and Conference Proceedings, vol. 37, JMLR.org, pp. 448–456, URL http://proceedings.mlr.press/v37/ioffe15.html.

Kamiya, T., Kusumoto, S., Inoue, K., 2002. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28 (7), 654–670. http://dx.doi.org/10.1109/TSE.2002.1019480.

Kanade, A., Maniatis, P., Balakrishnan, G., Shi, K., 2020. Learning and evaluating contextual embedding of source code. In: Proceedings of the 37th International Conference on Machine Learning. ICML 2020 13-18 July 2020, Virtual Event, In: Proceedings of Machine Learning Research, vol. 119, PMLR, pp. 5110–5121, URL http://proceedings.mlr.press/v119/kanade20a.html.

Khomh, F., Vaucher, S., Guéhéneuc, Y., Sahraoui, H.A., 2011. BDTEX: a gqm-based bayesian approach for the detection of antipatterns. J. Syst. Softw. 84 (4), 559–572. http://dx.doi.org/10.1016/J.JSS.2010.11.921.

Kovačević, A., Slivka, J., Vidaković, D., Grujić, K.-G., Luburić, N., Prokić, S., Sladić, G., 2022. Automatic detection of long method and god class code smells through neural source code embeddings. Expert Syst. Appl. 204, 117607. http://dx.doi.org/10.1016/J.ESWA.2022.117607.

Lanza, M., Marinescu, R., 2006. Object-Oriented Metrics in Practice - using Software Metrics To Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, ISBN: 978-3-540-24429-5, http://dx.doi.org/10.1007/3-540-39538-5.

Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y., Zhang, L., 2021. Deep learning based code smell detection. IEEE Trans. Softw. Eng. 47 (9), 1811–1837. http://dx.doi.org/10.1109/TSE.2019.2936376.

Liu, H., Liu, Q., Niu, Z., Liu, Y., 2016. Dynamic and automatic feedback-based threshold adaptation for code smell detection. IEEE Trans. Softw. Eng. 42 (6), 544–558. http://dx.doi.org/10.1109/TSE.2015.2503740.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., 2019. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.

Ma, W., Liu, S., Zhao, M., Xie, X., Wang, W., Hu, Q., Zhang, J., Liu, Y., 2024. Unveiling code pre-trained models: Investigating syntax and semantics capacities. ACM Trans. Softw. Eng. Methodol. (ISSN: 1049-331X) 33 (7), http://dx.doi.org/10.1145/3664606.

Macia, I., Garcia, A., von Staa, A., 2010. Defining and applying detection strategies for aspect-oriented code smells. In: 2010 Brazilian Symposium on Software Engineering. IEEE, pp. 60–69. http://dx.doi.org/10.1109/SBES.2010.14.

Madeyski, L., Lewowski, T., 2020. MLCQ: industry-relevant code smell data set. In: EASE '20: Evaluation and Assessment in Software Engineering. Trondheim, Norway, April 15-17, 2020, ACM, pp. 342–347. http://dx.doi.org/10.1145/3383219.3383264.

Madeyski, L., Lewowski, T., 2023. Detecting code smells using industry-relevant data. Inf. Softw. Technol. 155, 107112. http://dx.doi.org/10.1016/J.INFSOF.2022.107112.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Guéhéneuc, Y., Aïmeur, E., 2012a. SMURF: A svm-based incremental anti-pattern detection approach. In: 19th Working Conference on Reverse Engineering. WCRE 2012, Kingston, ON, Canada, October 15–18, 2012, IEEE Computer Society, pp. 466–475. http://dx.doi.org/10.1109/WCRE.2012.56.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Guéhéneuc, Y., Antoniol, G., Aïmeur, E., 2012b. Support vector machines for anti-pattern detection. In: IEEE/ACM International Conference on Automated Software Engineering. ASE'12, Essen, Germany, September 3–7, 2012, ACM, pp. 278–281. http://dx.doi.org/10.1145/2351676.2351723.

Marinescu, R., 2004. Detection strategies: Metrics-based rules for detecting design flaws. In: 20th International Conference on Software Maintenance. (ICSM 2004), 11–17 September 2004, Chicago, IL, USA, IEEE Computer Society, pp. 350–359. http://dx.doi.org/10.1109/ICSM.2004.1357820.

Marinescu, R., 2005. Measurement and quality in object-oriented design. In: 21st IEEE International Conference on Software Maintenance. ICSM 2005, 25-30 September 2005, Budapest, Hungary, IEEE Computer Society, pp. 701–704. http://dx.doi.org/10.1109/ICSM.2005.63.

Marinescu, R., Ganea, G., Verebi, I., 2010. Incode: Continuous quality assessment and improvement. In: 14th European Conference on Software Maintenance and Reengineering. CSMR 2010 15–18 March 2010, Madrid, Spain, IEEE Computer Society, pp. 274–275. http://dx.doi.org/10.1109/CSMR.2010.44.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. (4), 308–320. http://dx.doi.org/10.1109/TSE.1976.233837.

Moha, N., Guéhéneuc, Y.-G., Duchien, L., Le Meur, A.-F., 2009. Decor: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. 36 (1), 20–36. http://dx.doi.org/10.1109/TSE.2009.50.

Munro, M.J., 2005. Product metrics for automatic identification of bad smell design problems in java source-code. In: 11th IEEE International Symposium on Software Metrics. METRICS 2005, 19–22 September 2005, Como Italy, IEEE Computer Society, p. 15. http://dx.doi.org/10.1109/METRICS.2005.38.

Nair, V., Hinton, G.E., 2010. Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning. ICML-10, June 21-24, 2010, Haifa, Israel, Omni Press, pp. 807–814, URL https://icml.cc/Conferences/2010/papers/432.pdf.

Nguyen, H., Treude, C., Thongtanunam, P., 2024. Encoding version history context for better code representation. In: Proceedings of the 21st International Conference on Mining Software Repositories. pp. 631–636.

Sales, V., Terra, R., Miranda, L.F., Valente, M.T., 2013. Recommending move method refactorings using dependency sets. In: 20th Working Conference on Reverse Engineering. WCRE 2013, Koblenz, Germany, October 14–17, 2013, IEEE Computer Society, pp. 232–241. http://dx.doi.org/10.1109/WCRE.2013.6671298.

Schafer, J.L., 1997. Analysis of Incomplete Multivariate Data. Chapman & Hall/CRC, http://dx.doi.org/10.1201/9780367803025.

Sharma, T., Efstathiou, V., Louridas, P., Spinellis, D., 2021. Code smell detection by deep direct-learning and transfer-learning. J. Syst. Softw. 176, 110936. http://dx.doi.org/10.1016/j.jss.2021.110936, URL https://www.sciencedirect.com/science/article/pii/S0164121221000339.

Sharma, T., Mishra, P., Tiwari, R., 2016. Designite: a software design quality assessment tool. In: Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities, BRIDGE@ICSE 2016, Austin, Texas, USA, May 17 2016. ACM, pp. 1–4. http://dx.doi.org/10.1145/2896935.2896938.

Sharma, T., Spinellis, D., 2018. A survey on software smells. J. Syst. Softw. 138, 158–173. http://dx.doi.org/10.1016/J.JSS.2017.12.034.

Sikström, S., Garcia, D., 2020. Statistical semantics. In: Methods and Applications. Springer, Cham, http://dx.doi.org/10.1145/3664606.

Singh, S., Kaur, S., 2018. A systematic literature review: Refactoring for disclosing code smells in object oriented software. Ain Shams Eng. J. 9 (4), 2129–2151. http://dx.doi.org/10.1016/j.asej.2017.03.002, URL https://www.sciencedirect.com/science/article/pii/S2090447917300412.

Suryanarayana, G., Samarthyam, G., Sharma, T., 2014. Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann, http://dx.doi.org/10.1016/j.asej.2017.03.002, URL https://www.sciencedirect.com/science/article/pii/S2090447917300412.

Van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M., 2007. On the detection of test smells: A metrics-based approach for general fixture and eager test. IEEE Trans. Softw. Eng. 33 (12), 800–817. http://dx.doi.org/10.1109/TSE.2007.70745.

Vidal, S.A., Vázquez, H.C., Pace, J.A.D., Marcos, C.A., Garcia, A.F., Oizumi, W.N., 2015. Jspirit: a flexible tool for the analysis of code smells. In: 34th International Conference of the Chilean Computer Science Society. SCCC 2015, Santiago, Chile, November 9–13, 2015, IEEE, pp. 1–6. http://dx.doi.org/10.1109/SCCC.2015.7416572.

Von der Mosel, J., Trautsch, A., Herbold, S., 2022. On the validity of pre-trained transformers for natural language processing in the software engineering domain. IEEE Trans. Softw. Eng. 49 (4), 1487–1507. http://dx.doi.org/10.1109/TSE.2022.3178469.

Xu, Z., Liu, J., Luo, X., Yang, Z., Zhang, Y., Yuan, P., Tang, Y., Zhang, T., 2019. Software defect prediction based on kernel pca and weighted extreme learning machine. Inf. Softw. Technol. 106, 182–200. http://dx.doi.org/10.1016/j.infsof.2018.10.004.

Zhang, Y., Ge, C., Hong, S., Tian, R., Dong, C., Liu, J., 2022. Delesmell: Code smell detection based on deep learning and latent semantic analysis. Knowl.-Based Syst. 255, 109737. http://dx.doi.org/10.1016/J.KNOSYS.2022.109737.

Zhang, M., Hall, T., Baddoo, N., 2011. Code bad smells: a review of current knowledge. J. Softw. Maint. Evol.: Res. Pr. 23 (3), 179–202. http://dx.doi.org/10.1002/SMR.521.