Expert Systems With Applications 204 (2022) 117607

Contents lists available at ScienceDirect

Expert Systems With Applications

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

Automatic detection of Long Method and God Class code smells through neural source code embeddings

Aleksandar Kovačević, Jelena Slivka [*], Dragan Vidaković, Katarina-Glorija Grujić, Nikola Luburić, Simona Prokić, Goran Sladić

Faculty of Technical Sciences, University of Novi Sad, Serbia

ARTICLE INFO

Keywords:
Code smell detection
Neural source code embeddings
Code metrics
Machine learning
Software engineering

ABSTRACT

Code smells are structures in code that often harm its quality. Manually detecting code smells is challenging, so researchers proposed many automatic detectors. Traditional code smell detectors employ metric-based heuristics, but researchers have recently adopted a Machine-Learning (ML) based approach. This paper compares the performance of multiple ML-based code smell detection models against multiple metric-based heuristics for detection of God Class and Long Method code smells. We assess the effectiveness of different source code representations for ML: we evaluate the effectiveness of traditionally used code metrics against code embeddings (code2vec, code2seq, and CuBERT). This study is the first to evaluate the effectiveness of pre-trained neural source code embeddings for code smell detection to the best of our knowledge. This approach helped us leverage the power of transfer learning – our study is the first to explore whether the knowledge mined from code understanding models can be transferred to code smell detection. A secondary contribution of our research is the systematic evaluation of the effectiveness of code smell detection approaches on the same large-scale, manually labeled MLCQ dataset. Almost every study that proposes a detection approach tests this approach on the dataset unique for the study. Consequently, we cannot directly compare the reported performances to derive the best-performing approach.

1. Introduction

Code smells are structures in code that signal the need for refactoring to avoid degrading the software's quality attributes (Fowler, 2018). These code structures negatively impact the software products' maintainability, evolvability, and testability (Sharma & Spinellis, 2018). Classes affected by code smells are change- and fault-prone (Khomh, Di Penta, Guéhéneuc, & Antoniol, 2012). Code smells negatively affect source code comprehension, decreasing developers' productivity (Sharma & Spinellis, 2018). Thus, code smells increase the overall cost of software development (Sharma & Spinellis, 2018), and software engineering experts agree that their removal leads to sustainable software development (Fowler, 2018; Sharma & Spinellis, 2018; Martin, 2009).

Unfortunately, manually detecting code smells in practice is not easy. Many code smell definitions are vague, and refactoring decisions rely heavily on human intuition and experience (Fowler, 2018). Developers frequently need to detect smells in unfamiliar code, which is challenging (Hozano, Garcia, Fonseca, & Costa, 2018). A solution to this problem is developing the tools to detect code smells in source code automatically, and many such tools have been proposed (Azeem, Palomba, Shi, & Wang, 2019).

Most code smell detection tools use metric-based heuristics defined by the domain experts to determine whether a code snippet suffers from a particular smell (Azeem et al., 2019). Metric-based heuristic approaches calculate a set of source code metrics for a code snippet and compare the obtained metric values to predefined thresholds. However, as Azeem et al. (2019) pointed out, these approaches suffer from several limitations that prevent their adoption in practice. The most critical issues are the low agreement between the existing detectors and strong dependence on user-defined parameters, for which no principled way of defining has been devised. Machine Learning (ML) learn-by-example approach promises to solve these issues (Azeem et al., 2019).

A principal challenge in using ML algorithms to detect source code issues is creating a vector representation of the analyzed source code

[*] Corresponding author at: Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovića 6, 21 000 Novi Sad, Serbia.
E-mail addresses: kocha78@uns.ac.rs (A. Kovačević), slivkaje@uns.ac.rs (J. Slivka), vdragan@uns.ac.rs (D. Vidaković), katarina.glorija@uns.ac.rs (K.-G. Grujić), nikola.luburic@uns.ac.rs (N. Luburić), simona.prokic@uns.ac.rs (S. Prokić), sladicg@uns.ac.rs (G. Sladić).

https://doi.org/10.1016/j.eswa.2022.117607
Received 28 July 2021; Received in revised form 9 May 2022; Accepted 15 May 2022
Available online 19 May 2022

snippet suitable for applying ML algorithms.

We recently witnessed a breakthrough in the Natural Language Processing (NLP) field with the introduction of Word Embeddings as a viable representation of linguistic items (Mikolov, Chen, Corrado, & Dean, 2013; Devlin, Chang, Lee, & Toutanova, 2018). These vector representations of text capture both statistical and semantic information and are effective in various NLP tasks (Bakarov, 2018). As suggested by the “naturalness hypothesis” (Allamanis, Barr, Devanbu, & Sutton, 2018), source code has similar statistical properties as natural language. Substantial empirical proof for this hypothesis exists, as models initially developed for natural language proved highly effective in many source code analysis applications (Allamanis et al., 2018). Recently, inspired by the success of pre-trained contextual embeddings for natural languages, Kanade et al. proposed CuBERT (Kanade, Maniatis, Balakrishnan, & Shi, 2020), a code understanding model inspired by the BERT model (Devlin et al., 2018) that holds state-of-the-art status in many natural-language understanding benchmarks. In their paper, Kanade et al. show that CuBERT displays state-of-the-art performance on multiple code-understanding tasks while requiring shorter training and fewer labeled examples than its alternatives. This paper will investigate the effectiveness of CuBERT on the problem of automatic code smell detection.

On the other hand, there are significant differences between the source code and natural language. In contrast to the natural language, source code is written as the formal language that presents minimal ambiguity, extensive re-use of identical “sentences,” reduced robustness to minor changes, and contains long semantic units (Lozoya, Baumann, Sabetta, & Bezzi, 2021). For these reasons, representations that exploit code’s structural nature have been proposed, such as code2vec (Alon, Zilberstein, Levy, & Yahav, 2019) and code2Seq (Alon, Brody, Levy, & Yahav, 2018). These embeddings leverage the syntactic structure of programming languages by representing the code snippet as a set of compositional paths over its abstract syntax tree (AST). Code2vec and code2seq are regarded as the representative *state-of-the-art* among the contextual embeddings for source code (Rabin et al., 2021; Kang, Bissyandé, & Lo, 2019).

Rabin, Mukherjee, Gnawali, and Alipour (2020) observed that few manually engineered features could perform very close to the higher dimensional code2vec embeddings. Thus, it is necessary to include handcrafted features as baselines. As source code metrics are frequently used handcrafted features for code smell detection (Azeem et al., 2019), we regard ML models trained using these features as baselines.

Our study aims to evaluate the effectiveness of pre-trained neural source code embeddings on the task of code smell detection. To this aim, we experiment on the MLCQ dataset (Madeyski & Lewowski, 2020), in which each code sample was manually examined for the presence and severity of code smells. In our experiments, we compare the performance of ML classifiers trained using the following source code vector representations:

- three neural source code embeddings: (1) code2vec (2) code2seq and (3) CuBERT
- a vector of source code metrics
- a vector of source code metrics and votes of heuristic-based detectors.

We include multiple heuristic-based detectors as our baselines. Fig. 1 presents an overview of the steps in our experiment.

Our target code smells are God Class and Long Method. We chose these code smells as they are highly prevalent (Palomba et al., 2018a) (i. e., developers frequently create this code smell while programming) and have a high negative impact on developer performance (Cairo, Carneiro, de Resende, & Brito, 2019), software quality attributes (Palomba et al., 2018a; Lacerda, Petrillo, Pimenta, & Guéhéneuc, 2020), and software reliability (Palomba et al., 2018a; Piotrowski & Madeyski, 2020). They frequently co-occur with other code smells (Palomba et al., 2018b; de Paulo Sobrinho, De Lucia, & de Almeida Maia, 2018), which was proven to be especially problematic (Palomba et al., 2018a; Yamashita & Moonen, 2013; de Paulo Sobrinho et al., 2018).

We consider the binary classification problem – we classify the code samples as smelly (positive class) or non-smelly (negative class). To fairly compare the performance of the algorithms, we compare all algorithms using the same experimental setting. In our experiments, the ML classifier trained using CuBERT source code embeddings achieved the best performance for both God Class (F-measure of 0.53) and Long Method detection (F-measure of 0.75). God Class detection proved to be a much more challenging task than Long Method detection. The reason might lie in the inconsistency of class-level annotations in the used dataset. This inconsistency may be alleviated by applying well-defined guidelines for annotating a code smell, modeled after the NLP field guidelines. Furthermore, as God Class is a class level smell and Long Method is a method level smell, the God Class detector must deal with a more significant input variability than the Long Method detector.

This study is the first to apply pre-trained neural source code embeddings for code smell detection. This approach is beneficial for two main reasons. Firstly, the metric extraction process is time-consuming, brittle, and not scalable. Remarkably, we can obtain better performance by automatically inferring features through Deep Learning (DL). Secondly, by employing neural source code embeddings, we leverage the power of transfer learning. As manually labeling code smells is time-consuming, tedious, and requires domain-specific expertise, code smell datasets are exceedingly small compared to datasets used in other fields such as NLP and computer vision. It is hard to train quality ML models on small datasets as ML models are prone to overfitting. A way to combat overfitting is transfer learning: exploiting the commonalities of different learning tasks to enable knowledge transfer across them. In our setting, we exploit the knowledge from learning a methods’ name from its code for which the labeled data is plentiful. We then use the features we learned to capture the codes’ semantic for code smell detection for which labels are scarce[1].

Apart from ours, we found a single study by Sharma, Efstathiou, Louridas, and Spinellis (2019) that leverages transfer learning for code smell detection. They experimented with training DL models for code smell detection in C# code and evaluating it over Java code and vice-versa. To obtain a large-scale labeled dataset, Sharma et al. automatically label code smells using heuristic tools. Our study differs from theirs in two aspects. Firstly, instead of training a DL model on the same code smell detection task in different programming languages, we transfer the knowledge captured by code understanding models. Secondly, we use a fully manually labeled code smell dataset, which has a clear advantage of having manually labeled test samples to measure our model’s usefulness in practice. By automatically labeling the datasets, Sharma et al. have shown that employing DL and transfer learning is feasible for code smell detection. However, their DL models are essentially learning the behavior of imperfect automatic code smell detectors.

Our study is the first to report the performance of multiple code smell detectors on the large-scale manually labeled MLCQ dataset. Having a high-quality benchmark dataset to compare different approaches is paramount to developing ML algorithms for any application domain. Without it, researchers are forced to perform experiments on vastly different datasets, and it is impossible to compare solution performances or measure the progress of the field. This paper emphasizes how this is currently the case for code smell detection. Several praiseworthy studies create datasets that could serve as a benchmark. However, most of these datasets suffer from difficulties with reproducibility, potentially noisy labels due to the semi-automatic labeling, or an unrealistic code smell distribution. We found that the MLCQ dataset currently represents the

[1] 1 We also combat overfitting by tuning the models’ parameters through fold-cross validation, employing ensemble models, and balancing the training set. These techniques are recognized as general ways to combat overfitting (Yang et al., 2020).

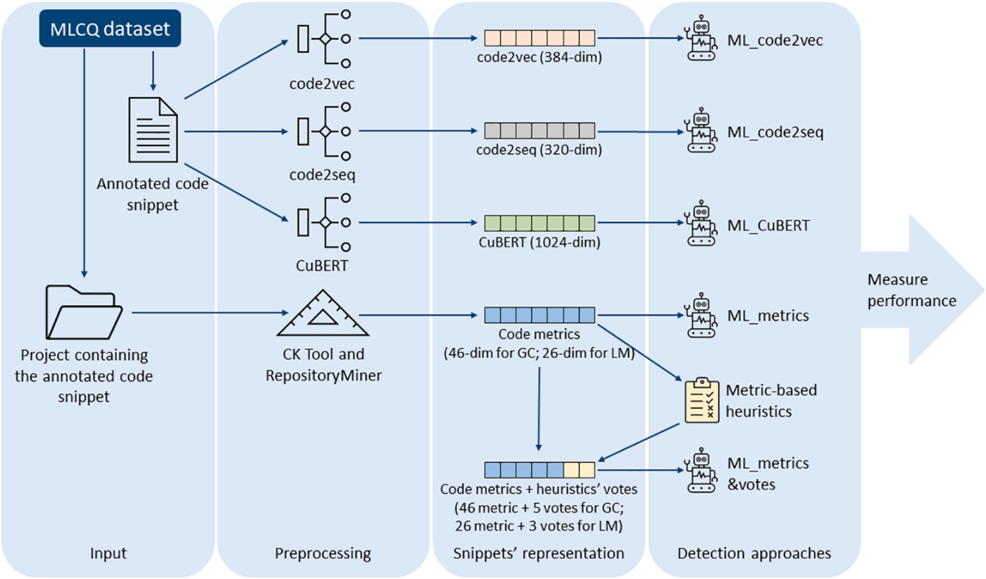

Fig. 1. Flowchart summarizing the steps in our experiment.

best benchmarking option.

With this in mind, a significant contribution of this study is that it is the first to evaluate many traditionally employed code smell detection approaches on a single benchmark dataset. Researchers are currently debating whether the ML approach for code smell detection is beneficial, i.e., whether it genuinely outperforms a heuristic-based approach (Azeem et al., 2019; Pecorelli, Di Nucci, De Roover, & De Lucia, 2020). Our study gives further proof in favor of the ML approach. To ensure the replicability of our findings, we provide a comprehensive replication package.

To summarize, the contributions of this paper are as follows:

- We evaluated the effectiveness of three state-of-the-art neural source code embeddings: code2vec, code2seq, and CuBERT on the task of God Class and Long Method detection. This study is the first to evaluate the usefulness of pre-trained neural source code embeddings for code smell detection. This study is also the first to explore whether the knowledge mined from code understanding models can be transferred to code smell detection.
- This study is the first to perform code smell detection on the MLCQ dataset. Almost every study that proposes a detection approach tests this approach on the dataset unique for the study. Consequently, we cannot directly compare the reported performances to derive the best-performing approach.
- This study is the first to compare the performances of many traditionally employed code smell detection approaches on the same large-scale, manually labeled dataset. Previous studies report the performances on substantially different datasets, which disabled their exact comparison.
- We provide a comprehensive replication package[2] researchers can use to replicate our experiments.

The rest of the paper is organized as follows. Section 2 provides the background work for our study. Section 3 describes the experiment we designed to fairly compare different code smell detection approaches. Section 4 provides the application details for the code smell detection approaches used in our experiment. Section 5 provides and discusses the result of our experiment. Section 6 lists the threats to validity that may affect our conclusions. Finally, section 7 concludes our paper.

2. Background work

This section provides the background work for our study. Section 2.1 defines our target code smells and explains the need for their detection and removal. Section 2.2 provides the background for metric-based heuristic detectors. Here, we describe the limitations of metric-based heuristics that motivate the ML-based code smell detection approach and emphasize why it is essential to use heuristics as baseline approaches in our study. Section 2.3 provides the background for ML classifiers trained using code metrics as features. We discuss the limitations of extracting such handcraft features that motivate our approach of employing automatically inferred features. Our study is the first to use neural source code embeddings for the code smell detection task – section 2.4 overviews code embeddings proposed in the literature and their application to different software engineering tasks.

2.1. Target code smells

This section motivates our choice to detect God Class and Long Method code smells. Many cataloged code smells exist (Sharma & Spinellis, 2018), and addressing all smell types would be challenging due to their varied nature. Azeem et al. (2019) recommend that researchers consider the smell harmfulness when deciding which smells to target. Palomba et al. (2018a) warn researchers should also consider the smell's prevalence, for though a specific smell may be harmful, it may not be as relevant if this type of smell rarely occurs in software projects. Here we review the available empirical studies that argue that God Class and Long Method smells have a high negative impact and that developers frequently create them while programming.

The God Class[3] code smell represents a class that implements many responsibilities and contains a large part of the system's logic (Sharma & Spinellis, 2018). It disregards an essential principle of object-oriented programming that each component in the system should have one purpose and one responsibility. God classes are typically huge classes with many attributes and methods that are not interconnected, meaning that the class has low cohesion[4]. Also, the God Class is generally

[2] [2] https://codeocean.com/capsule/5256791/tree/v1.

[3] [3] In this paper we consider God Class, Large Class, Blob Class, and Brain Class as synonyms.

[4] [4] Cohesion describes the extent to which the functions within a module are related.

associated with many other components in the system, which indicates high coupling.

The Long Method code smell occurs when a method is lengthy (Sharma & Spinellis, 2018). Typically, these methods tend to centralize the functionality of the class. They are complex, challenging to understand, and process large amounts of data from other classes. As God Classes, Long Methods represent a violation of the single responsibility principle of object-oriented programming.

God Class and Long method are both prevalent and harmful to software's quality attributes. Palomba et al. (2018a) conducted an extensive empirical investigation on code smells prevalence. They found that God Class and Long Method are among the seven most prevalent smells whose removal provides a high benefit in terms of change-proneness and thus should be removed to improve the overall maintainability of the code.

A recent tertiary systematic review (Lacerda et al., 2020) also concluded that the God Class and Long Method smells are amongst those that most affect different software quality attributes, God Class having the most considerable negative effect. Furthermore, besides negatively affecting quality attributes, instances suffering from these smells are more bug-prone than others.

Another recent systematic literature review of papers that analyze the relationship between code smells and fault-proneness (Piotrowski & Madeyski, 2020) has confirmed that code smells positively correlate with software defects. Thus, code smells are valuable predictors for fault detection models. However, not all types of smells are equally helpful for fault detection – God Class and Long Method are exceptionally effective.

Another important finding is that the interaction of code smells can increase the change-proneness (Palomba et al., 2018a) and inhibit maintainability (Yamashita & Moonen, 2013) of source code. The possible reason for this is that the co-occurrence of smells strongly hinders the developers' ability to understand the source code (Abbes, Khomh, Gueheneuc, & Antoniol, 2018). Alarmingly, Palomba et al. (2018b) found that smell co-occurrence is a widespread phenomenon. The same study found that God Class and Large class frequently co-occur with other smells.

2.2. Heuristic-based detection

This section reviews the current state of applying metrics-based heuristics for code smell detection. Most existing code smell detectors are heuristic-based (Azeem et al., 2019), and most heuristic-based approaches are metric-based (Bafandeh Mayvan, Rasoolzadegan, & Javan Jafari, 2020). Metric-based approaches detect code smells by calculating a particular set of code metrics and applying predefined thresholds for each code metric. Although metric-based smell detection approaches have reasonable performance, several limitations hinder their adoption in practice.

The existing tools suffer from a high rate of false positives (software artifacts wrongly detected as being affected by a smell) (Sharma & Spinellis, 2018). Many code smell candidates increase the needed human effort in fixing the code smells, as the developers need to analyze many code snippets to identify actual problems. The required effort may exceed the budget allocated for refactoring (Bafandeh Mayvan et al., 2020).

Metric-based approaches require the specification of thresholds for each metric. The selection of these thresholds greatly influences smell detectors' performance (Azeem et al., 2019), and there is no standardized way of choosing them.

The available approaches differ in the set of source code metrics they consider relevant for code smell detection. Additionally, even the approaches that use a similar set of metrics may differ in the predefined thresholds. Consequently, the agreement between different heuristic-based detectors is low (Azeem et al., 2019). There is no agreement on correct results and no standard criteria for comparing the detectors' performance (Bafandeh Mayvan et al., 2020).

Implementing a metric-based approach may be precarious as the code metrics definitions lack standardization. For example, the Coupling Between Objects metric (CBO) metric is a long-established and widely used metric for code smell detection in object-oriented languages (Azeem et al., 2019) that measures the extent of coupling between two classes. However, the coupling can be measured and interpreted in many ways (Briand, Wüst, Ikonomovski, & Lounis, 1999; Child, Rosner, & Counsell, 2019). Consequently, such ambiguities in metric definitions led to different metrics tools producing widely inconsistent results even for well-known metrics (Lincke, Lundberg, & Löwe, 2008; Sharma & Spinellis, 2018).

Furthermore, the metric tools' documentations often quote a standard definition of the calculated metrics, with scant detail on handling ambiguities in their implementations (Child et al., 2019). Similarly, some studies proposing a metric-based smell detection approach lack precise definitions of the metrics they use. This lack of precision further complicates threshold specification as thresholds may depend on the tool used for metric value calculation.

To overcome the limitations of heuristic-based approaches, recently, the researchers started experimenting with ML-based techniques for smell detection. Azeem et al. (2019) conducted a systematic literature review on the proposed ML techniques for code smell detection and performed a *meta*-analysis reported performances. According to their *meta*-analysis, ML techniques generally outperform heuristic-based methods. However, there is no established benchmark for comparing different techniques, and some studies found that heuristic techniques perform slightly better than ML techniques (Pecorelli et al., 2020). Thus, when developing a ML-based approach for code smell detection, it is essential to include heuristic-based approaches as a baseline to show if and when ML is called for (Allamanis et al., 2018).

2.3. Classifiers trained on code metrics

This section discusses the advantages and disadvantages of a traditional ML-based approach for code smell detection. To train an ML model for code smell detection, we first need to represent the code snippets in our dataset as fixed-length vectors of real numbers. In most ML-based code smell detection approaches, researchers represent the code snippet by extracting features hand-engineered by the domain experts, i.e., the code metrics (Azeem et al., 2019). In this paper, we consider this approach of training ML models using hand-crafted features as a baseline method against which we compare the performance of ML models trained on automatically inferred features (section 4.3).

An advantage of using hand-crafted features is that they allow us to analyze and interpret the behavior of ML models trained using them. We rely on these features when performing ML model error analysis (section 5.3). A disadvantage of this approach is that designing helpful features is challenging. It requires domain expertise, is time-consuming, and is problem-dependent: features engineered for a particular software engineering task might not be helpful in a different context (Wang, Huang, Ge, Zhang, Feng, Li, & Ng, 2020).

An alternative to hand-crafting features is to automatically infer helpful features through deep learning (section 4.3). However, this approach may lead to a slight performance improvement (Rabin et al., 2020) while sacrificing model interpretability. Thus, it is vital to include models trained on manually engineered features as baselines to estimate if the performance improvement justifies the added model complexity (Allamanis et al., 2018).

2.4. Applying source code embeddings in software engineering tasks

This section reviews code embeddings proposed in the literature and explains why we chose code2vec, code2seq, and CuBERT embeddings for our code smell detection experiment. The current practices of developing source code models can be divided into three groups: representations that capture the structural nature of code, NLP-inspired models, and code change models.

Structural embeddings model the codes' AST (Neamtiu, Foster, & Hicks, 2005; Alon et al., 2019; Alon et al., 2018), data flow (Ben-Nun, Jakobovits, & Hoefler, 2018), or control flow graph (DeFreez, Thakur, & Rubio-González, 2018). Well-known embeddings from this group that the community extensively adopted due to their public tool support (Pour, Li, Ma, & Hemmati, 2021) are code2vec (Alon et al., 2019) and code2seq (Alon et al., 2018). Researchers used these embeddings to solve various software engineering tasks such as code comment generation, code authorship identification, code clones detection (Kang et al., 2019), code summarization (Wang, Gao, & Wang, 2021), detection of source code vulnerabilities (Coimbra, Reis, Abreu, Păsăreanu, & Erdogmus, 2021), and semantic code search (Arumugam, 2020). Embeddings that capture the structural nature of code might be beneficial for God Class and Long Method detection. Code snippets that suffer from these smells have high structural complexity that hinders the codes' readability (Luburić, Prokić, Grujić, Slivka, Kovačević, Sladić, & Vidaković, 2021). To the best of our knowledge, our study is the first to use these embeddings for code smell detection.

NLP contextual embeddings capture the text semantics (Bakarov, 2018). Code embeddings inspired by these techniques hold the potential to capture source code semantics crucial for many software engineering tasks such as software defect prediction (Pan, Lu, & Xu, 2021), automated program repair (Mashhadi & Hemmati, 2021), and vulnerability detection (Wu, 2021). Karampatsis and Sutton (2020) used the ELMo framework (Peters, Neumann, Iyyer, Gardner, Clark, Lee, & Zettlemoyer, 2018) to train their embeddings and achieved state-of-the-art results for bug detection. CodeBERT (Feng, Guo, Tang, Duan, Feng, Gong, & Zhou, 2020), based on BERT (Devlin et al., 2018), achieved state-of-the-art results on natural language code search and code documentation generation tasks. Recently, Kanade et al. (2020) proposed CuBERT, a code-understanding BERT model, that outperformed other embeddings on multiple software engineering benchmark tasks. As the codes' semantics are crucial for God Class and Long Method detection (Luburić et al., 2021), we apply CuBERT for these tasks.

Embeddings that model code changes encode the difference in the syntactic structure of code before and after a change. Such embeddings are helpful for the classification of security-relevant commits (Sabetta & Bezzi, 2018; Lozoya et al., 2021), log message generation, bug fixing patch identification, and just-in-time defect prediction (Hoang, Kang, Lo, & Lawall, 2020). We did not consider this group of embeddings for code smell detection, although code change history can be helpful for God Class detection (Barbez, Khomh, & Guéhéneuc, 2019; Palomba et al., 2014). The MLCQ dataset contains samples from 792 projects and extracting the needed embeddings would be extremely time-intensive. Thus, we leave this idea for future research.

3. Experiment design

To fairly compare the performance of our algorithms, we applied all code smell detection algorithms using the same experimental setting. In this section, we explain our experiment design. Section 3.1 explains why we chose the MLCQ dataset (Madeyski & Lewowski, 2020) to train and evaluate our models. Then, in section 3.2, we describe the specifics of this dataset. Finally, section 3.3 explains our experiment design and chosen performance measure.

3.1. Choosing the dataset

A principal challenge for applying ML algorithms to the code smell detection problem is the need for a large, manually labeled dataset. This section argues that the MLCQ dataset (Madeyski & Lewowski, 2020) is the best benchmark for comparing code smell detection algorithms.

Manually labeling code smells in the source code is time-consuming (Azeem et al., 2019) and challenging (Hozano et al., 2018). To combat this issue, many researchers train their models on the datasets entirely built by applying heuristic-based tools (Azeem et al., 2019; Boutaib et al., 2021) or on synthetically generated data (Liu et al., 2019). However, we cannot reliably use such datasets to evaluate ML-based code smell detectors. The automatic annotation procedure does not guarantee that the dataset does not contain falsely labeled instances. In particular, heuristic-based tools are prone to produce a high rate of false positives (software artifacts wrongly detected as being affected by a smell) (Sharma & Spinellis, 2018).

Fontana, Mäntylä, Zanoni, and Marino (2016) manually labeled instances from the Qualitas corpus for the presence of four code smells. Their annotation procedure used five heuristic-based detectors to identify a set of code smell candidates. Then, three MSc students manually examined and labeled code smell candidates, ensuring that one-third of the instances had positive labels (representing code smells) while the remaining ones were negative. In a later study, Di Nucci, Palomba, Tamburri, Serebrenik, and De Lucia (2018) argued that the artificially enforced ratio of code smells to non-smells significantly affects the conclusions obtained on the Qualitas corpus.

Palomba, Di Nucci, Tufano, Bavota, Oliveto, Poshyvanyk, and De Lucia (2015) built the Landfill code smell dataset by manually annotating 243 instances from 20 open-source Java projects. In their annotation procedure, one author manually examined the projects to identify code smell instances, and then the second author validated these candidates to discard false positives. Unfortunately, we could not download this dataset as the hyperlinks in the paper are no longer valid (Madeyski & Lewowski, 2020).

In a later study, Palomba et al. (2018a) built a larger dataset by applying the heuristic-based tools to lower the number of manually validated code snippets. Then they manually filtered the false positives. The advantage of this dataset is its size and diversity – it encompasses 395 releases of 30 open-source systems. However, the study aimed not to train ML models for code smell detection but rather assess the diffuseness of code smells and their impact on code change- and fault-proneness. As such, this dataset has several limitations that limit its applicability for training ML models for code smell detection. Firstly, due to the employed semi-automatic annotation procedure, there is no guarantee that this dataset does not contain false-negative instances. Secondly, it is unclear whether the annotators received training and guidelines for code smell detection. Finally, as Madeyski and Lewowski (2020) pointed out, the dataset is not published in the form that can be used for reliable reproduction as it lacks information such as project URLs, commit hashtags, and full classpaths.

In this study, we use the MLCQ dataset (Madeyski & Lewowski, 2020). It is the largest publicly available dataset for which multiple annotators manually analyzed the smelliness of each code sample. As MLCQ authors stated, the advantage of this dataset over other existing code smell datasets is that it provides all information needed for reliable reproduction. In contrast to (Fontana et al., 2016), random sampling of code samples other datasets should result in a near-natural ratio of code smells and none-smells. In contrast to (Palomba et al., 2018a), each instance in the dataset was manually annotated. Finally, in contrast to all other datasets where the annotators were MSc students, the reviewers involved in the code smell assessment in the MLCQ dataset were active in the software development industry.

3.2. MLCQ dataset

This section reviews the properties of the MLCQ dataset we used in our experiments. Madeyski and Lewowski (2020) complied the MLCQ dataset through collaboration with 26 professional software developers. They gathered the code samples from 792 active industry-relevant, contemporary Java open-source projects and asked developers to review the collected code samples for bad smells. In this way, nearly 15 000 code samples were annotated for four code smell types: God Class, Long Method, Feature Envy, and Data Class. Records in the MLCQ data set contain:

- The type of smell and its severity (none, minor, major, or critical)
- Crucial information for retrieval of the code sample and the software project that contains it: repository link, revision, and the exact location of the source code's smell (a particular method for the Long Method and Feature Envy smells, or a particular class for the God Class and Data Class code smells)
- The ID of the professional developer that provided the annotation.

We note that some of the code samples were annotated by multiple developers. Some annotations were cross-checked[5], and for some, there is a non-negligible disagreement among the reviews. Annotator disagreement is more pronounced for class-level smells than method-level ones (Madeyski & Lewowski, 2020). Thus, there is some ambiguity about generating the ground-truth label for the smell. We use the majority-vote label for the samples where multiple conflicting annotations are available in our work.

Madeyski and Lewowski purposefully omitted annotator guidelines for recognizing particular code smells. They aimed to extract professional developers' contemporary understanding of code smells instead of imposing thresholds from the legacy literature. They asked the developers to answer an extensive survey on their professional experience in software development and code smells' perception. We use this information on the developer's background when analyzing our models' performance (section 5.3).

However, this annotation approach might prove problematic for training ML models for code smell prediction. Creating a domain-specific dataset with high-quality annotations is essential to train and evaluate ML models of maximum accuracy in a particular domain. For example, expert-based corpus acquisition has well-defined guidelines and best practices for creating consistent and high-quality annotations in the NLP field. Annotating a dataset requires training the annotators and is referred to as "the science of annotation" (Hovy, 2010; Ide & Pustejovsky, 2017).

It should be noted that we could not retrieve all code samples listed in the MLCQ dataset. As Madeyski and Lewowski (2020) pointed out, the project is no longer accessible when a repository is removed. We include all available code samples in our analysis: 2408 unique class samples and 2255 unique method samples.

3.3. Experimental setup

This section presents the experiment we performed on the MLCQ dataset to compare code smell detection algorithms. We used the random stratified sampling procedure to divide the available code samples into the training (80%) and test (20%) set. The stratification procedure strived to produce an approximately equal ratio of all four severity levels in the training and test set. Table 1 presents the ratio of positive and negative samples in each part.

Table 1
MLCQ dataset: the ratio of positive (code samples affected by the considered smell) and negative samples (code samples not affected by the considered smell) in train and test sets.

MLCQ Dataset	% smell instances	No. instances (pos/neg)	No. instances train (pos/neg)	No. instances test (pos/neg)
Long Method	13.0	277/2131	223/1703	54/428
God Class	12.2	251/2049	197/1642	54/407

We consider the binary classification problem – we classify the code samples as smelly ("minor," "major," or "critical" severity) or non-smelly (severity "none"). We consider the smelly class the positive class and the non-smelly class the negative class. This classification problem is highly imbalanced (Table 1), with the positive class being the minority class. Thus, as a performance measure, we calculate the F1-measure of the positive class (Eq. (1)):

$$F1 = 2 \bullet \frac{precison \bullet recall}{precision + recall}, \tag{1}$$

where:

$$precision = \frac{TP}{TP + FP}, recall = \frac{TP}{TP + FN}, \tag{2}$$

where TP denotes the number of true positives, FP denotes the number of false positives, TN denotes the number of true negatives, and FN denotes the number of false negatives.

In our paper, we train ML classifiers using the multiple source code vector representations:

- three neural source code embeddings: (1) code2vec (2) code2seq and (3) CuBERT
- a vector of source code metrics
- a vector of source code metrics and votes of heuristic-based detectors

We train the following ML classifiers on each representation: Random Forest, Bagging (using Support Vector Machines as the base algorithm), and Gradient Boosted Trees (XGBoost). We used the classifier implementations from the Scikit-learn (Pedregosa et al., 2011) and XGBoost (Chen & Guestrin, 2016) libraries. We optimized the hyper-parameters of these models by performing a stratified 5-fold-cross validation on the training set and using the Bayesian optimization strategy.

As the problem is highly imbalanced, we experimented with several sampling strategies from the *imbalanced-learn* package (Lemaître, Nogueira, & Aridas, 2017). Thus, we train each ML classifier on:

- the dataset resampled using the SMOTE (Synthetic Minority Over-sampling Technique) oversampling of the minority class,
- the dataset resampled using the Neighborhood Cleaning Rule undersampling to remove noisy samples from the dataset,
- the dataset resampled using a combined strategy of SMOTE minority oversampling and Edited Nearest Neighbor undersampling (SMOTEENN),
- the original dataset (without resampling).

When performing resampling, we made sure to resample only the training portion of the data and leave the validation data (for optimizing the model's hyper-parameters) and test data (for evaluating the model's performance) unchanged.

To ensure the reproducibility of our experiments, we fixed the random seed value in our experiments.

4. Methodology

This section provides the application details for each code smell detection approach used in our experiment: (1) heuristic-based detection (Section 4.1), (2) ML classifier trained using code metrics as features (Section 0), and (3) ML classifier trained using code embeddings as features (Section 4.3).

4.1. Heuristic-based detection

This section explains how we chose the metric-based heuristic detectors we implemented and extracted the needed code metric values. Recently, Bafandeh Mayvan et al. (2020) conducted a systematic

[5] [5] Only samples tagged with a severity higher than "none" were considered for cross-checking. Samples selected for cross-checking were samples with a single review or samples with two conflicting annotations.

literature review on bad smell specification and metric-based code smell detection approaches. To select the baseline metric-based detection approaches, we started from the list of metric-based rules for God Class and Long Method neatly summarized in their paper. We eliminated the rules for which we could not find precise metric definitions in the original paper that proposed the rule. For the remaining rules, we carefully examined the metric definitions provided in the documentation of the tools we used for metric extraction and eliminated the rules which use different definitions of the used metric or use the metrics the tools do not calculate. Table 2 shows the final set of rules we implemented as heuristic-based baselines.

Tables 3 and 4 show the class-level and method-level metric definitions and the metric tools we used to calculate the heuristic baselines. Figs. 2 and 3 show the frequency with which researchers use each metric to specify the code smell. We can see that most researchers characterize God Classes using size (NOM, NOF, CLOC) and complexity (WMC) metrics, while some also include coupling (ATFD) and cohesion (LCOM) measures. Researchers characterize Long Methods primarily by their size, while some also include their complexity.

When choosing the tools to extract metric values, we had to impose certain requirements. To cope with ambiguity in calculating the metrics, we need the tools to be open-source and well documented. As MLCQ encompasses many projects (792), we needed the tools to do batch computation and export data in a parseable format. We also looked for the tools that extract the methods through static analysis (i.e., they do not require compiled code) to avoid compilation issues. We selected the CK Tool (Aniche, 2015) and RepositoryMiner (Barbez et al., 2019) for metric extraction based on these requirements.

Table 2
Heuristic-based approaches summarized in (Bafandeh Mayvan et al., 2020) we used as baselines in our experiments.

Code smell	Denotement	Rule specification
God Class	GC_1	[ATFD] $>$ 2 & [WMC] $\geq$ 47 & [TCC] $<$ 0.33 Trifu and Marinescu (2005): God Class is complex, non-cohesive, and accesses more than two attributes of other classes.
	GC_2	[WMC] $\geq$ 47 & [TCC] $<$ 0.3 & [ATFD] $>$ 5 Macia et al. (2012): God class is complex, non-cohesive, and accesses more than five attributes of other classes.
	GC_3	[NOM] $>$ 15 | [NOF] $>$ 15 Kiefer, Bernstein, and Tappolet (2007): God Class has a lot of methods and instance variables.
	GC_4	[CLOC] $>$ 750 | [NOM] + [NOF] $>$ 20 Fard et al. (2013): God Class has many lines of code, or it has a lot of methods and properties.
	GC_5	[NOM] + [NOF] $>$ 20 Moha, Guéhéneuc, Duchien, and Le Meur (2009): God Class declares many fields and methods.
	GC_6	[LCOM] $\geq$ 0.725 & [WMC] $\geq$ 34 & [NOF] $\geq$ 8 & [NOM] $\geq$ 14 Souza, Sousa, Ferreira, and Bigonha (2017): God Class is huge, complex, and has an extremely high number of fields and methods.
	GC_7	[NOM] $>$ 20 | [NOF] $>$ 9 | [CLOC] $>$ 750 Danphitsanuphan and Suwantada (2012): God Class has many lines of code, or it has many global variables or many methods.
	GC_8	[CLOC] $>$ 100 | [WMC] $>$ 20 Liu, Ma, Shao, and Niu (2011): God class has many lines of code or is highly complex.
Long Method	LM_1	MLOC $>$ 50 Fard et al. (2013): Long Method has many lines of code.
	LM_2	MLOC $>$ 30 & VG $>$ 4 & NBD $>$ 3 Souza et al. (2017): Long Method is huge, complex and has a high number of nested blocks.
	LM_3	MLOC $>$ 50 | VG $>$ 10 Liu et al. (2011): Long Method has many lines of code or is highly complex.

Table 3
Definition of source code metrics used in heuristic-based approaches for God Class detection.

Metric	Acronym	Definition	Metric extraction tool
Number of methods	NOM	Number of methods in a class	CK Tool
Number of fields	NOF	Number of attributes of a class	
Tight class cohesion	TCC	The relative number of method pairs of a class that access in common at least one attribute of the measured class	
Lines of code	CLOC	Lines of code in a class	RepositoryMiner
Access to foreign data	ATFD	The number of attributes from unrelated classes accessed directly or by invoking accessor methods	
Weighted method count	WMC	The sum of the static complexity of all methods in a class. The McCabe's complexity (cyclomatic complexity) is used to quantify the method's complexity	
Lack of cohesion	LCOM5	Provides a measure for the lack of cohesion of a class. Lack of cohesion between the methods of the class	

Table 4
Definition of source code metrics used in heuristic-based approaches for Long Method detection.

Metric	Acronym	Definition	Metric extraction tool
Lines of code	MLOC	Lines of code in a method	CK Tool
McCabe's complexity	VG	Cyclomatic complexity of the method	
Nested blocks depth	NBD	Nested blocks depth	

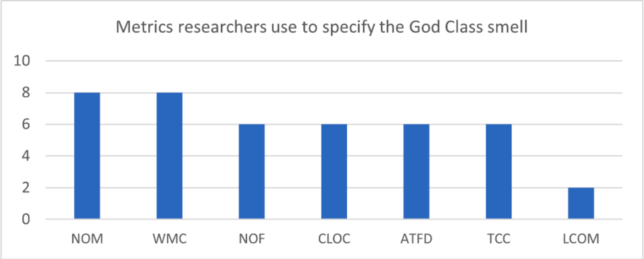

Fig. 2. Metric frequency in specifying the God Class code smell. In their systematic literature review, Bafandeh Mayvan et al. (2020) analyzed 16 studies that specified God Class detection heuristics. We created this graph based on their results.

In our experiments, we test the performance of the individual rules and their three aggregations. Let us denote the number of rules as K and the indicator function of rule k classifying the instance x as suffering from a considered smell as $\mathbb{1}_k(x)$:

$$\mathbb{1}_k(x) = \begin{cases} 1, & \text{if x is a smell according to rule } k \\ 0, & \text{if x is not a smell according to rule } k \end{cases} . \tag{3}$$

We evaluate the following three models:

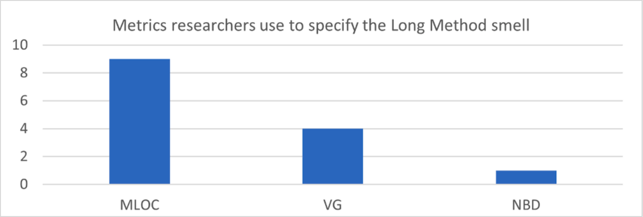

Fig. 3. Metric frequency in specifying the Long Method code smell. In their systematic literature review, Bafandeh Mayvan et al. (2020) analyzed 9 studies that specified Long Method detection heuristics. We created this graph based on their results.

- **ALL** – the model predicts that the code sample suffers from the smell if all individual rules agree it suffers from the considered smell (Eq. (4)).

$$\mathbb{I}_{ALL}(x) = \bigwedge{}^{K}_{k=1} \mathbb{I}_k(x) \tag{4}$$

- **ANY** – the model predicts that the code sample suffers from the smell if ANY of the individual rules classify it as the considered smell (Eq. (5)).

$$\mathbb{I}_{ANY}(x) = \bigvee{}^{K}_{k=1} \mathbb{I}_k(x) \tag{5}$$

- **Weighted vote** – the model calculates the probability of the code sample suffering from the smell as a weighted vote of the individual classifiers:

$$P(\widehat{y}_{vote}= 1|x) = \sum\nolimits_{k=1}^{K} w_k \bullet \mathbb{I}_k(x), \tag{6}$$

where w_k is the weight of the rule k. If $P(\widehat{y}_{vote}= 1|x) \geq 0.5$, we classify the code sample as suffering from the considered smell; otherwise, we classify it as non-smell. We derive the weights of individual rules $w_k, k \in \{1, .., K\}$ according to the F-measure the rules achieved on the training set. More precisely, we treat the F-measures of an individual rules as their scores $[s_1, s_2, \cdots, s_K]$. We then apply the *softmax* function to the rule score vector $[s_1, s_2, \cdots, s_K]$ to obtain the rule weights $[w_1, \cdots, w_K]$.

We note that our approach has several limitations. We could not implement all the rules from Bafandeh Mayvan et al. (2020) due to the explained ambiguities. Secondly, we used our implementation. Though implementing the rules is simple, we may have misinterpreted metric descriptions or threshold values from the original paper or metric-extraction tools documentation. This issue could be mitigated by using publicly available and tested tools that perform code smell detection. For example, Fernandes, Oliveira, Vale, Paiva, and Figueiredo (2016) address the same issue of comparing different code smell detection tools, and in their experiment, they use publicly available heuristic-based detectors. However, they analyze the source code of a single project. As MLCQ encompasses many projects, we were restricted to the tools able to process this dataset[6]. We also note that the issue of calculating metric values for a specific heuristic cannot be resolved entirely as programming language constructs constantly evolve. Consequently, code metric designers cannot anticipate all possibilities and precisely define the metrics for every possible case.

4.2. Classifier trained on code metrics

This section specifies how we preprocess the code snippet to represent it as a vector of source code metrics for training ML classifiers.

In our experiment, we train ML models on code metrics extracted using the CK Tool and RepositoryMiner metric extraction tools we used to facilitate the implementation of heuristic-based detectors. We include all extracted metrics, regardless of whether we use them in heuristic baselines or not. In cases where both tools calculate the same metric, we choose the metric whose implementation corresponds to the metric definition used in heuristic-based detectors. This way, we use a total of 46 class-level metrics for God Class detection and 26 method-level metrics for Long Method detection[7]. Before applying ML models, we performed the following preprocessing techniques:

- We encoded the categorical variables using label encoding.
- We used −1 to denote that the value of a particular metric cannot be calculated for the analyzed code sample.
- We normalized the metric values using z-normalization.

We refer to this approach as *ML_metrics.*

Additionally, we test another code snippet representation – a vector constructed by concatenating the values of the samples' code metrics and the vector of heuristics votes for the sample. The vector of heuristics votes has a value of 0 (code snippet is not a code smell according to the heuristic) or 1 (code snippet is a code smell according to the heuristic) for each considered heuristic. We denote this approach as *ML_metrics&votes.*

4.3. Classifier trained on neural source code embeddings

The previous section discussed how the analyzed source code snippets could be represented as a vector of hand-crafted features to apply ML models. This section presents an alternative way of representing source code as a fixed-length vector of real numbers: neural source code embeddings (Alon et al., 2019). In this approach, embeddings are learned by training deep neural networks to do a proxy task (e.g., method name prediction), for which labeled training data is plentiful. The goal is that the learned representation of the source code should preserve its semantic meaning – semantically similar source code snippets should be mapped to similar vectors (Alon et al., 2019).

This large-scale pretraining of code embeddings is beneficial for ML applications in which labeled data is scarce (Hussain, Huang, Zhou, & Wang, 2020). One such application is code smell detection, as manual labeling of code smells is time-consuming (Azeem et al., 2019) and challenging (Hozano et al., 2018). Fortunately, if we use the embeddings trained to capture code's semantics, we might require far less training data than the amount necessary to train a code smell detection model from scratch.

Researchers proposed many source code embeddings. Section 2.4 motivated our choice to use three representative embeddings for our code smell detection task: code2vec, code2seq, and CuBERT. Sections 4.3.1 to 4.3.3 explain how we use the chosen embeddings to obtain methods' representations. Finally, section 4.3.4 explains how we obtain the embedding of classes.

4.3.1. Code2vec

Alon et al. (2019) proposed code2vec, a deep neural network that learns code embeddings. Their neural network model input is the vector representation of the code snippet, and the output is a corresponding tag. For example, Alon et al. used the method's body as the input code snippet and the methods' name as the output tag.

Alon et al. obtained the models' input (source code snippets' vector representation) by representing the code snippet as an unordered set of paths in its AST:

1. A code snipped is parsed to produce its AST.

[6] 6 At the beginning of this section, we explain the requirements we had to impose when choosing the tools to extract metric values.

[7] 7 The exact metrics and their extracted values can be found in our replication package https://codeocean.com/capsule/5256791/tree/v1.

2. The paths between all pairs of AST leaves are extracted. A single path is a tuple of the two leaf nodes in AST and the path (an ordered sequence of AST nodes) between them.
3. Each path is assigned a weight that corresponds to its' importance to the models' output. An attention mechanism is used to learn the paths' importance.
4. A single vector representation of the code snippet is calculated as a weighted average of the path vectors.

We used code2vec to represent each method in our dataset as a fixed-length vector for our code smell detection task. We used the implementation provided by Alon et al. (2019):

1. We construct the methods' AST paths using the path extractor for the Java programming language.
2. We input the extracted AST paths to the Alon et al. trained open-sourced Java model. We remove the last softmax layer from the model and use the feature vector ("code vector") as the models' output. We treat this 384-dimensional vector as a representation of the method.

4.3.2. Code2seq

Code2Seq (Alon et al., 2018) is an encoder-decoder model that learns the vector representation of the source code. Same as code2vec, the models' input is the vector representation of the code snippet, and the output is a corresponding natural language sequence (e.g., the method name).

Like code2vec, the models' input is constructed from the code snippets' AST:

1. A code snippet is parsed to produce its AST.
2. The paths between all pairs of AST leaves are extracted. A single path is an ordered sequence of AST nodes.
3. A bi-directional LSTM encodes each AST path as a fixed-length sequence of nodes.
4. The decoder uses attention to select relevant paths while decoding to generate an output sequence.

We used code2seq to represent each method in our dataset as a 320-dimensional vector for our code smell detection task. We used the implementation provided by Alon et al. (2018), who provided the extractors for preprocessing source code of the input methods and the open-sourced model for Java programming language.

4.3.3. CuBERT

The transformer model and context-dependent embeddings have revolutionized the field of NLP, with the BERT model (Devlin et al., 2018) as one of the first and most prominent examples. Kanade et al. (2020) built the CuBERT model with that in mind. CuBERT is essentially the "Large" variant of the BERT model pre-trained on code snippets instead of natural language. BERT Large is a transformer (Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, & Polosukhin, 2017) with 4 layers with 16 attention heads that provide a 1024-dimensional vector embedding for each token. Originally Kanade et al. (2020) trained CuBERT on Python code but subsequently published a version pre-trained on Java code snippets that we used in our experiments.

As input to the BERT Large model is a sequence of tokens, we pre-processed the Java code snippets using a custom Java tokenizer developed by Kanade et al. Then, we input each sequence (line of Java code) into the CuBERT model. We then extracted the 1024-dimensional embedding from a special start-of-example ([CLS]) token to represent each line of code, as recommended by the original BERT authors (Devlin et al., 2018).

4.3.4. Embedding methods and classes

Long Method is a method-level smell. Pre-trained code2vec and code2seq models expect the methods' body as an input code snippet. Thus, we use the methods' body as input to obtain a method-level embedding and consider the resulting embedding vector as the methods' vector representation.

God Class is a class-level smell. Pre-trained code2vec and code2seq models cannot directly embed a class. Thus, to perform predictions at the class level, we need to define the appropriate class embedding. Compton, Frank, Patros, and Koay (2020) addressed this problem by treating classes as collections of methods. They evaluated the efficacy of class embeddings that constitute simple mathematical operations on sets of method embeddings. In their experiment, the *mean* aggregation method achieved the best performance. Thus, we adopted this approach for embedding classes using code2vec and code2seq models: to embed a class, we embed all its constituting methods and calculate the mean of the resulting code embedding vectors.

The pre-trained CuBERT model expects a single line of code as an input. Thus, to embed a method using the CuBERT model, we embed the method by embedding each line of code in a methods' body and then summing all resulting vectors. Similarly, to embed the class, we embed all its constituting methods and calculate the sum of the resulting code embedding vectors. Like Compton et al. (2020), we evaluated the efficacy of different mathematical operations on sets of method embeddings (mean, concatenation, and summation) and concluded that summation resulted in the highest performance for this task.

5. Results and discussion

This section presents the results of our experiment and discusses them in the context of the related research. In section 5.1, we evaluate the heuristic-based code smell detectors on the training set and select the best-performing heuristic. Section 5.2 compares this best-performing heuristic to ML-based approaches on the test set. Finally, in section 5.3, we perform the error analysis for our best-performing approach.

5.1. Heuristic-based detection

This section presents the performance of heuristic-based detectors listed in section 4.1 on the MLCQ dataset. We discuss heuristics results on MLCQ in the context of the original studies that proposed them and show that comparing heuristics on the same large-scale manually labeled dataset is our studies' significant contribution.

Table 5 presents the performance of heuristic-based rules on the task of God Class detection, and Table 6 presents the performance of heuristic-based rules on the task of Long Method detection. As the application of heuristic-based detectors does not require training, we report the performance of heuristic-based approaches on the training set, the test set, and the whole dataset (training and test set combined). Note that, in a realistic scenario, we would need to choose the best-performing algorithm using the training data exclusively. Thus, the best performing individual rule for God Class detection is GC_8 (Table 5) and the best performing individual rule for Long Method detection is ANY (Table 6). For an easier comparison of the heuristic-based detectors on the test set, we summarize the performances from Tables 5 and 6 in Figs. 4 and 5, respectively.

For the God Class smell (Table 5), our aggregations of the individual rules could not improve the performance compared to the individual rule performance. As expected, the ALL approach has the best precision of all heuristic-based approaches, but its recall is extremely low. In contrast, ANY approach has low precision but the highest recall. The Weighted Vote approach has the same performance as the rule GC_8. The GC_8 vote dominates the Weighted Vote as this rule achieves a significantly higher F-measure on the training set than other rules. As GC_8 is a more simplistic approach than Weighted Vote and achieves the same performance, we consider GC_8 the best performing heuristic for God Class detection in our experiment. In Table 6, we observe the same trends for our aggregation approaches for the Long Method smell.

Table 5
Performance of heuristic-based approaches for God Class detection. We report the performance on the training set, the test set, and the whole dataset (All). We test the performance of both individual rules (GC_1 to GC_8) and three rule aggregations (ALL, ANY, Weighted Vote). The performance is reported in terms of precision (P), recall (R), and F-measure (F) of the positive (smell) class.

	Rule specification	Training set			Test set			All		
		P	R	F	P	R	F	P	R	F
GC_1	[ATFD] $>$ 2 & [WMC] $\geq$ 47 & [TCC] $<$ 0.33	0.26	0.03	0.06	**0.75**	0.06	0.10	0.35	0.03	0.06
GC_2	[WMC] $\geq$ 47 & [TCC] $<$ 0.3 & [ATFD] $>$ 5	0.42	0.02	0.05	**0.75**	0.06	0.10	0.50	0.03	0.06
GC_3	[NOM] $>$ 15 $\mid$ [NOF] $>$ 15	0.33	0.42	0.37	0.44	0.52	0.47	0.35	0.44	0.39
GC_4	[CLOC] $>$ 750 $\mid$ [NOM] + [NOF] $>$ 20	0.35	0.46	0.40	0.39	0.44	0.42	0.36	0.46	0.40
GC_5	[NOM] + [NOF] $>$ 20	0.35	0.46	0.40	0.39	0.44	0.42	0.36	0.46	0.40
GC_6	[LCOM] $\geq$ 0.725 & [WMC] $\geq$ 34 & [NOF] $\geq$ 8 & [NOM] $\geq$ 14	0.31	0.56	0.40	0.34	0.61	0.43	0.31	0.57	0.40
GC_7	[NOM] $>$ 20 $\mid$ [NOF] $>$ 9 $\mid$ [CLOC] $>$ 750	0.36	0.43	0.39	0.39	0.39	0.39	0.37	0.42	0.39
GC_8	[CLOC] $>$ 100 $\mid$ [WMC] $>$ 20	0.34	0.66	**0.45**	0.37	**0.70**	**0.49**	0.35	0.67	**0.46**
Rule agg.	ALL	**0.50**	0.02	0.04	**0.75**	0.06	0.10	**0.58**	0.03	0.05
	ANY	0.30	**0.70**	0.42	0.31	**0.70**	0.43	0.30	**0.70**	0.42
	Weighted Vote	0.34	0.66	**0.45**	0.37	**0.70**	**0.49**	0.35	0.67	**0.46**

Size metrics: NOM (Number of Methods), NOF (Number of Fields), CLOC (Lines of Code in a class).
Complexity metrics: WMC (Weighted Method Complexity).
Cohesion metrics: TCC (Tight Class Cohesion), LCOM (lack of cohesion).
Coupling metrics: ATFD (Access To Foreign Data).

Table 6
Performance of heuristic-based approaches for Long Method detection. We report the performance on the training set, the test set, and the whole dataset (All). We test the performance of both individual rules (LM_1 to LM_8) and three rule aggregations (ALL, ANY, Weighted Vote). The performance is reported in terms of precision (P), recall (R), and F-measure (F) of the positive (smell) class.

	Rule specification	Training set			Test set			All		
		P	R	F	P	R	F	P	R	F
LM_1	MLOC $>$ 50	0.93	0.26	0.40	**1.00**	0.20	0.34	0.94	0.24	0.39
LM_2	MLOC $>$ 30 & VG $>$ 4 & NBD $>$ 3	0.88	0.24	0.37	0.91	0.18	0.31	0.89	0.23	0.36
LM_3	MLOC $>$ 50 $\mid$ VG $>$ 10	0.86	0.40	0.55	0.89	0.31	0.46	0.87	0.38	0.53
Rule agg.	ALL	**0.96**	0.11	0.20	**1.00**	0.11	0.20	**0.97**	0.11	0.20
	ANY	0.85	**0.47**	**0.61**	0.86	**0.33**	**0.48**	0.85	0.44	**0.58**
	Weighted Vote	0.86	0.40	0.55	0.89	0.31	0.46	0.87	0.38	0.53

Size metrics: MLOC (Lines of Code in a method).
Complexity metrics: VG (McCabe's complexity), NBD (Nested Blocks Depth).

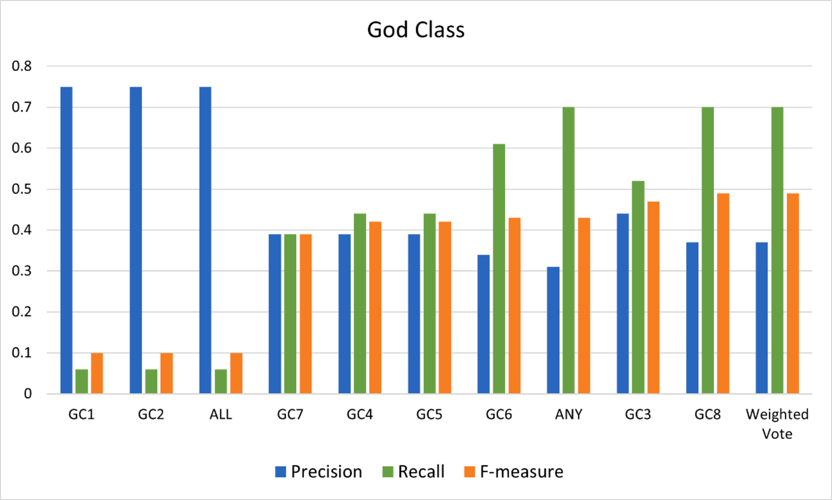

Fig. 4. Performance of heuristic-based approaches for God Class detection on the test set. We sort the performances from left to right according to the F-measure, our primary performance metric.

However, the ANY model has a significantly higher F-measure on the training set than other approaches. We consider the ANY approach the best performing heuristic for Long Method detection in our experiment.

A surprising result for the God Class smell (Table 5) is that, although the ALL approach has the highest precision of all heuristic-based strategies, its precision on the whole dataset is still relatively low (58%). We would expect ALL to have near-perfect precision as, for the samples it labels as God Class, there is a perfect agreement of diverse God Class detection rules. For comparison, the ALL approach for Long Method has an extremely high precision of 97%, as expected. To understand the modest precision of ALL approach for God Class detection, we asked the domain expert to analyze classes this rule mislabeled as suffering from

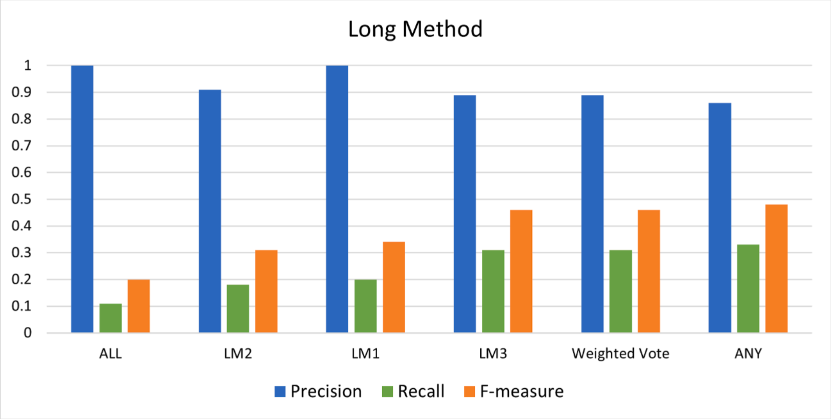

Fig. 5. Performance of heuristic-based approaches for Long Method detection on the test set. We sort the performances from left to right according to the F-measure, our primary performance metric.

the God Class code smell. The ALL approach labeled 12 classes as the God Class smell, and five of these classes are labeled as non-smell by the annotator majority vote in the MLCQ dataset. Our domain expert inferred that these five examples are mislabeled in the MLCQ dataset – he would label these examples as major to critical. To further investigate this issue, we looked at individual annotations in the MLCQ dataset and the annotator experience[8]. We found that three out of these five annotations had low annotator agreement. It is interesting to observe that the MLCQ annotator with a lot of professional software engineering experience and the largest number of annotated samples in the MLCQ dataset labeled these examples as "critical." The two remaining code samples were labeled as "none" by a single annotator and were not cross-checked. This finding indicates that the MLCQ dataset has shortcomings that might affect the training and evaluation of the ML-based God Class detectors. A more systematic approach to code smell labeling might be required to overcome this issue.

Unfortunately, it is hard to compare the results reported in Tables 5 and 6 to the results reported in the existing literature. We need to evaluate each using the same experimental setting to compare different approaches fairly. Code smell literature reviews unanimously call for a benchmark dataset on which different code smell detection approaches might be compared (Azeem et al., 2019; Lewowski & Madeyski, 2022; Menshawy, Yousef, & Salem, 2021; AbuHassan, Alshayeb, & Ghouti, 2021). Section 3.1 argues that the MLCQ dataset is currently the best benchmark option as it is the largest, fully manually labeled, fully reproducible dataset. To the best of our knowledge, our study is the first to evaluate heuristics from Tables 5 and 6 on MLCQ.

Nevertheless, comparing the performances from Tables 5 and 6 to those reported in the original studies would be helpful to prove the validity of our results. However, Table 7 shows that this comparison is impossible because our experiment differs significantly from those performed in the original studies. We can summarize these differences in three main points:

1. Many original studies do not report performance measures. Instead, they perform case studies that analyze the results qualitatively or provide motivating examples. Some studies do not measure precision and recall directly through manually labeling data but instead calculate the correlation of their detection results with other signals such as the number of issued bug reports. Thus, we cannot state whether we obtained similar performance to these studies. Rasool and Arshad (2015) also observed the lack of thorough evaluation of the existing tools in their systematic literature review. They write: "The accuracy of a code smell detection tool is a key aspect for its validity. Most publications found in the literature did not mention the accuracy of their applied tool. Our review reveals that approximately 30% of the tools spotlight the accuracy, that is, precision and recall, of their technique or tool".
2. Comparing our results to the studies that use manually labeled data as the ground truth needed to compute precision and recall is also problematic due to the following reasons:
   a. Codebases used in the studies vary and are very different from the MLCQs' codebase. While many studies use popular Java open-source projects as their codebase, the number of projects they use ranges from 1 to 12. In contrast, the MLCQ dataset encompasses 792 active industry-relevant, contemporary Java open-source projects. It is a much larger, more current, and more diverse codebase that contains projects of varying sizes, coding styles, and problem domains.
   b. Even if two studies use the same codebase, ground truth annotations may vary. Software engineers largely disagree on which code snippets suffer from a particular code smell (Hozano et al., 2018). Each original study used different annotators. Most studies do not comment on the annotator experience or how they perceive code smells (i.e., they omit the annotation guidelines). Only a single study addressed the issue of distinct code smell understanding by using multiple annotators and resolving disagreements by a majority vote. Finally, the issue of using different ground truths is especially problematic when the dataset is small as the ground truth becomes highly subjective.
3. We cannot reproduce the experiments from the original studies. Although most studies use open-source projects that we could retrieve as a codebase, we cannot obtain the labels they used as the ground truth as they are not publicly available.

Due to these reasons, the performances reported in Tables 5 and 6 differ from those reported in the original studies.

Fernandes et al. (2016) address the same issue of fairly comparing different God Class and Long Method detection tools using the same experimental setting. We compare their results to ours in Table 8. Like Fernandes et al. (2016), we found that the heuristics achieve low to medium rates for God Class detection. Fernandes et al. (2016) also report that the two tools they applied had maximum precision. However, we did not observe high precision rates for the God Class detection

[8] 8 We derive the annotator experience from the accompanying annotator survey. We considered both the years of experience in software development and the number of provided annotations as important experience factors. Software engineering experience contributes to the quality of code smell detection. Providing many annotations indicates that the annotator has developed a consistent strategy for labeling code smells in diverse projects (Luburić et al., 2021).

Table 7
Comparing the performance of metric-based heuristics achieved on the test portion of the MLCQ dataset (our experiment) to the performances (P - precision and R - recall) reported in the original studies.

	Approach	Experiment description	MLCQ dataset		Original study	
			P	R	P	R
GC_1	$[\text{ATFD}] > 2 \;\&\; [\text{WMC}] \geq 47 \;\&\; [\text{TCC}] < 0.33$ Trifu and Marinescu (2005)	A single open-source project. The authors do not report performance measures. Instead, they select a few top code smells (detected by their approach) and discuss why they constitute code smells.	0.75	0.06	N/A	N/A
GC_2	$[\text{WMC}] \geq 47 \;\&\; [\text{TCC}] < 0.3 \;\&\; [\text{ATFD}] > 5$ Macia et al. (2012)	Three open-source projects. The original developers annotated the smells. The authors inform the precision and recall for each project separately. However, as they have reported the exact number of TP, FP, and FN per project, we calculated the aggregated precision and recall, which we present here. Annotations are not publicly available.	0.75	0.06	0.5	0.38
GC_3	$[\text{NOM}] > 15 \mid [\text{NOF}] > 15$ Kiefer et al. (2007)	A single open-source project. The authors do not report performance measures. Instead, they perform a case study by measuring the number of bug reports issued for the code smells detected by their approach and show that the largest code smell candidate has the largest number of filed bug reports.	0.44	0.52	N/A	N/A
GC_4	$[\text{CLOC}] > 750 \mid [\text{NOM}] + [\text{NOF}] > 20$ Fard et al. (2013)	The authors consider nine JavaScript applications chosen for manual checking as they do not have a large codebase. The authors report a total of 15 God Classes in this codebase. The dataset is manually annotated. However, the authors do not specify if multiple annotators examined each code sample. They do not state the annotator's expertise or understanding of the code smells. The annotations are not publicly available.	0.39	0.44	0.78	0.94
GC_5	$[\text{NOM}] + [\text{NOF}] > 20$ Moha et al. (2009)	The dataset consists of 11 manually annotated open-source systems. The dataset is manually annotated. Five annotators examined a single system using Brown's and Fowler's books as references. The authors report that their approach achieves 100% recall and 89.6% precision on this single system. The authors only report the precision for the remaining ten systems as it is too costly to calculate recall. As the authors report the precision per system, we report the average precision here. The paper states that the dataset is publicly available; however, we could not access the dataset link from the paper when we performed our experiments.	0.39	0.44	0.89	N/A
GC_6	$[\text{LCOM}] \geq 0.725 \;\&\; [\text{WMC}] \geq 34 \;\&\; [\text{NOF}] \geq 8 \;\&\; [\text{NOM}] \geq 14$ Souza et al. (2017)	The authors examined 12 software systems. The authors compare their approach to the results obtained by the existing tools JDeodorant and JSPiRIT. Additionally, annotators examined two of those systems manually using Fowler's book as a reference. They do not specify whether multiple annotators analyzed the same code snippets. Here, we report the average precision and recall for those two systems. The dataset is not publicly available.	0.34	0.61	0.75	0.45
GC_7	$[\text{NOM}] > 20 \mid [\text{NOF}] > 9 \mid [\text{CLOC}] > 750$ Danphitsanuphan and Suwantada (2012)	The dataset consisted of 323 Java classes. The authors report they tested the used heuristics by using known bad smell source code from the "Refactoring: Improving the Design of existing Code" book. The approach had 100% accuracy for some smells for this source code. However, they do not report the accuracy for the Large Class code smell.	0.39	0.39	N/A	N/A
GC_8	$[\text{CLOC}] > 100 \mid [\text{VG}] > 20$ Liu et al. (2011)	The authors present a motivating example in which they apply their approach to a single systems' source code.	0.37	0.70	N/A	N/A
LM_1	$\text{MLOC} > 50$ Fard and Mesbah (2013)	The authors consider nine JavaScript applications chosen for manual checking as they do not have a large codebase. The authors report a total of 25 Long Methods in this codebase. The dataset is manually annotated. However, the authors do not specify if multiple annotators examined each code sample. They do not state the annotator's expertise or understanding of the code smells. The annotations are not publicly available.	1	0.2	1	1
LM_2	$\text{MLOC} > 30 \;\&\; \text{VG} > 4 \;\&\; \text{NBD} > 3$ Souza et al. (2017)	The authors examined 12 software systems. The authors compare their approach to the results obtained by the existing tools JDeodorant and JSPiRIT. Additionally, annotators examined two of those systems manually using Fowler's book as a reference. They do not specify whether multiple annotators analyzed the same code snippets. Here, we report the average precision and recall for those two systems. The dataset is not publicly available.	0.91	0.18	0.29	1
LM_3	$\text{MLOC} > 50 \mid \text{VG} > 10$ Liu et al. (2011)	The authors present a motivating example in which they apply their approach to a single systems' source code.	0.89	0.31	N/A	N/A

in our study. Like us, Fernandes et al. (2016) report better performances for Long Method detection.

In contrast to our study, Fernandes et al. (2016) only consider four tools and use a vastly smaller codebase consisting of a single Java application to calculate precision and recall. To the best of our knowledge, we are the first to compare the performances of many proposed heuristic-based detectors on the same large-scale, manually labeled dataset. Thus, we consider the results reported in Tables 5 and 6 as one of the significant contributions of this study.

5.2. *Comparing the performance of different approaches for code smell detection*

This section compares the performance of all approaches for code smell detection we considered in this study. Tables 9 and 10 summarize the performances of different approaches measured on the test set for God Class and Long Method, respectively: the first column lists how we denote each considered code smell detection approach; the second column lists the used source code representation; the third column presents the used algorithm; the last three columns report the precision, recall, and F-measure of the minority (smell) class measured on the test set. Fig. 6 graphically represents the F-measures reported in tables 9 and 10

Table 8

Comparison of the performance of metric-based heuristics reported in our paper and those reported by Fernandes et al. (2016). Fernandes et al. reported the performance of four tools on a single application and, here, we show the average performance for those four tools. We report the average performance of eight God Class and three Long Method detectors applied the whole MLCQ codebase (denoted ALL in Tables 5 and 6).

	God Class		Long Method	
	Average precision [%]	**Average recall [%]**	**Average precision [%]**	**Average recall [%]**
Our study	36.9 ± 5.6	38.5 ± 23.4	90.0 ± 3.6	28.3 ± 8.4
Fernandes et al. (2016)	64.8 ± 44.0	14.0 ± 0.0	74.3 ± 39.3	45.8 ± 16.3

for easier comparison.

For each approach, we evaluate only the best-performing model. For the heuristic-based models, the best performing models are GC_8 for God Class detection and ANY for Long Method detection (section 5.1). For ML-based models, we select the best-performing model after hyper-parameter tuning, according to the F-measure of the minority class evaluated through a 5-fold-cross validation procedure on the training dataset.

From Tables 9 and 10, we can see that, in our experiment, the best performing approach for both God Class and Long Method detection proved to be a classifier trained using CuBERT source code embeddings as code snippet representation, ML_CuBERT. The next two best-performing approaches are the ML classifier trained using source code metrics ML_metrics and the ML classifier trained using the combination of code metrics and votes of heuristic-based detectors as features ML_metrics&votes.

Excluding the ML classifier trained using the code2vec (*ML_code2vec*) and ML classifier trained using code2seq features (*ML_code2seq*), ML classifiers outperformed the heuristic-based approaches in our experiments. This result contradicts the study (Pecorelli et al., 2020) that found that heuristic techniques perform slightly better than ML techniques but agrees with Azeem et al. (2019) *meta*-analysis. As discussed in section 4.1, applying heuristic-based approaches is brittle due to the ambiguity of code metrics definitions and possible differences from the implementations of the same metric in different code metrics extraction tools. The significant advantage of ML-based detectors is that we can use the source code metrics as indicators, rather than imposing hardcoded thresholds that might need readjusting depending on the tool used for calculating code metrics.

Combining the heuristic-based detectors votes and code metrics for representing the analyzed source code (*ML_metrics&votes*) did not improve the performance compared to representing the source code using code metrics alone (*ML_metrics*). The improvement is slight for the God Class detection (1% improvement), while it hinders the performance for the Long Method detection (4% decrease).

ML_code2vec achieved the worst performance on both tasks. We attribute this to the low generalizability of code2vec embeddings. Code2vec is a representative *state-of-the-art* among embedding models (Kang et al., 2019), and it has excellent performance on the method name prediction task Alon et al. (2019) designed it for. However, the learned code embeddings may not be generalizable to other tasks. Kang et al. (2019) investigated the generalizability of code2vec embeddings to three different software engineering tasks: code comment generation, code authorship identification, and code clones detection. They have demonstrated that code2vec's token embeddings do not always significantly increase models' performance for these tasks and cannot be used readily to improve simpler models.

Code2seq is an improvement of code2vec, proposed by Alon et al. (2018). In our experiments, *ML_code2seq* outperformed *ML_code2vec* on both tasks. However, *ML_metrics* based on the hand-crafted features outperformed both embeddings. We may attribute this to the fact that these embeddings predominantly focus on the syntactic and lexical properties of the code with little to no semantic implication (Wang et al., 2021). On the other hand, CuBERT embeddings seem to capture the semantics needed for code smell detection.

In the case of God Class detection, the performance of *ML_CuBERT* is not significantly better than classifiers trained using source code metrics *ML_metrics* (2% improvement) or their combination with votes of heuristic-based detectors as features *ML_metrics&votes* (1% improvement). However, ML_CuBERT performs significantly better on the Long Method detection task than ML_metrics (8% improvement) and ML_metrics&votes (12% improvement).

We observe a similar trend when comparing *ML_CuBERT* to the heuristic-based approach *H_metrics*: the improvement is slight for the God Class detection (4% increase) but substantial for the Long Method detection (27% increase). Note that *H_metrics* and *ML_metrics* both rely on hand-crafted source code metrics.

This result indicates that the CuBERT features provide better source code representation for code smell detection than source code metrics and have the additional benefit of being automatically inferred. However, as the performance improvement for the God Class detection task is slight, we provide a deeper analysis of whether CuBERT features provide a better semantic representation of the analyzed source code than the source code metric in section 5.3.

According to the performance achieved by all tested approaches on the MLCQ dataset, God Class detection poses a much more challenging task than the Long Method detection. We may attribute this to one of the following reasons:

- In the MLCQ dataset, the annotators had significantly more disagreement for class-level smells than method-level smells (Madeyski & Lewowski, 2020). Thus, a more systematic approach for labeling code smells that would produce higher-quality labels is needed to tackle the God Class detection problem successfully. As pointed out in section 3.2, the NLP field has well-defined guidelines and best practices for creating consistent and high-quality annotations. Unfortunately, to the best of our knowledge, current publicly available manually labeled datasets for code smell detection were not subjected to the same stringent annotation guidelines.
- The worse performance of God Class detectors may be due to God Class being a class level smell, while the Long Method is a method level smell. Therefore, the God Class detector must deal with a more significant input variability than the Long Method detector, and more labeled data is needed to tackle this smell.

Finally, Table 11 compares the performance of our ML-based approach, *ML_metrics*, to similar schemes presented in the literature.

As Table 11 shows, the reported code smell detection performance is very dataset-dependent. Fontana et al. (2016) report that ML algorithms achieve a near-perfect performance in their study. However, DiNucci et al. (2018) showed the reported high-performance results from the used dataset characteristics. DiNucci et al. (2018) achieved significantly lower performance with a more realistic setup on the same dataset. Compared to DiNucci et al. (2018), we achieved a 6% lower F-measure for God Class detection and 19% higher F-measure for Long Method detection when applying the similar ML-based approach to MLCQ. The difference may be due to one or more of the following factors:

- Codebase differences: Qualitas corpus is considerably smaller than MLCQ. MLCQ dataset is also more diverse (i.e., it encompasses samples from a broader range of open-source projects) and more recent (i.e., it encompasses samples that represent the contemporary coding styles)
- The feature set differences: DiNucci et al. used an extensive set of code metrics provided in the Qualitas Corpus. However, they performed feature selection and used only the metrics that impact the prediction (approximately one-third of the original metrics). We

Table 9
Performance of different approaches for God Class detection. We report the precision, recall, and F-measure of the minority (smell) class evaluated on the test set.

Denotement	Features	Approach	Precision	Recall	F-measure
ML_code2vec	code2vec features	Random Forest + SMOTEENN	0.16	0.69	0.26
ML_code2seq	code2seq features	Bagging (SVM classifier) + SMOTE	0.34	0.50	0.41
H_metrics	Code metrics	Heuristic detector (GC_8)	0.37	0.70	0.49
ML_metrics	Code metrics	Random Forest + SMOTEENN	0.41	0.67	0.51
ML_metrics&votes	Code metrics + votes of heuristic detectors (GC_3 – GC_8)[1]	Random Forest + SMOTEENN	0.43	0.69	0.52
ML_CuBERT	CuBERT features	Bagging (SVM classifier) + SMOTEENN	0.48	0.58	**0.53**

[1] We excluded the votes of GC_1 and GC_2 as they achieved extremely low F-measure on the training set.

used a smaller set of metrics (46 class-level and 26 method-level metrics). A limitation of our study is that our choice of metrics was dictated by the capabilities of the metric extraction tools we used. We argue the choice of these metric extraction tools in Section 4.1. We did not perform feature selection as we used the ML classifiers robust to irrelevant features.
- Feature value differences: we relied on different metric extraction tools than those used for creating the Qualitas corpus. As we argued in Section 2.2, metrics definitions contain ambiguities that lead to metrics extraction tools producing different results for the same code snippet. However, the ML approach treats metric values as potentially noisy signals rather than the exact values. Thus, this should not have a significant impact on our performance. We chose ML classifiers robust to noisy features to mitigate this factor.

The study of Pecorelli et al. (2020) is most similar to ours as it compares the performance of ML-based detectors and heuristics for code smell detection. Considering the overall performance on all smell types, the authors report that heuristics perform slightly better than the ML-based approach. Compared to Pecorelli et al., we achieve a significantly higher performance in the case of both ML-based and heuristics-based detection. In ML-based detection, we achieve an 11% increase of F-measure for God Class and a 44% increase for Long Method. Our best heuristic for God Class detection achieves 33% higher F-measure, and our best heuristic for Long Method detection achieves 4% higher F-measure. We may attribute these differences to one or more of the following reasons:

- Codebase differences: MLCQ dataset encompasses code samples from a broader range of open-source projects and is more recent. It is also fully manually labeled.
- The feature set differences: Pecorelli et al. (2020) use a considerably smaller code metrics set than us. Their choice of code metrics is dictated by the heuristic-based approaches they use. It is possible that using metrics not considered by the heuristics is beneficial as the ML approach might uncover the less obvious relationships between metrics values and code smells.
- Differences in considered heuristics: Our study considered different detection rules extracted from a recent systematic literature survey on code smell heuristics. However, as discussed in section 4.1, we were limited to the rules for which we could find precise metric definitions to match the metrics extracted by the tools. This is a limitation of our study and heuristics-based detection in general. On the other hand, Pecorelli et al. (2020) considered a single heuristic and might have benefited from exploring more heuristic options.
- Feature value differences: we did not use the same metric extraction tools as Pecorelli et al. (2020). This might have impacted heuristic-based detection but should not have impacted ML-based detection significantly.

5.3. Performance analysis

This section presents the error analysis we performed to analyze the reasons for different performances of CuBERT automatically inferred features and hand-crafted code metrics features. We applied *ML_CuBERT*, *ML_metrics*, and *H_metrics* to the examples in the test set and collected all examples where these approaches made a wrong decision. Then, we asked the domain expert to analyze these code samples. Our goal is to manually identify code characteristics that influence the performance of these models. In the remainder of this section, we list our conclusions.

The performance improvement for God Class detection was slight, but our error analysis indicates CuBERT features can capture the semantics needed to detect code smells. *ML_CuBERT* correctly labeled God Classes that have multiple responsibilities that metric-based approaches mislabeled. To ensure that the metrics-based approaches detect these samples, we need to develop metrics that measure the number of responsibilities and can be automatically calculated, which is challenging. In contrast to *ML_metrics, ML_CuBERT* also correctly labeled atypical classes. For example:

- classes with many auto-generated comments;
- classes that have many methods and many fields but semantically do not suffer from the God Class smell as they are relatively simple and do not handle multiple responsibilities;
- classes that follow design patterns[9];
- classes that have a single complex method but are simple otherwise.

On the other hand, we concluded that *ML_CuBERT* might benefit from including the RFC metrics in the source code representation as it mislabeled several classes with large RFC that *ML_metric* correctly detected.

The performance improvement of *ML_CuBERT* over *ML_metrics* for the Long Method code smell was substantial, and we would need to design additional metrics to improve the performance of *ML_metrics*. Designing new code metrics is a tedious task that requires domain expertise and is problem-specific. This result highlights the benefit of using the CuBERT approach of automatically inferring features over hand-crafting features for different code smell detection tasks.

A disadvantage of using code metrics as features is that this approach relies upon non-trivial metric extraction implementations. As discussed in section 4.1, metric extraction is complex, and metric implementation may not necessarily follow the metrics designer's reasoning and intention as metric definitions are often incomplete (Lincke et al., 2008). For example, it is ambiguous should the tool consider the inner classes when calculating the metrics of the outer class. If the inner classes should be considered, this adds to the complexity of the metric calculations. The implication is that the thresholds used in metric-based heuristics are tool-dependent and therefore brittle.

Moreover, the metric calculation might depend on including or excluding library classes, and calculating metrics for a particular code snippet might require parsing the whole project. For example, it was very time-consuming to extract the metrics needed to represent the code samples from the MLCQ dataset. MLCQ includes roughly six samples per

[9] 9 Design patterns typically violate clean code principles but are not considered smells.

Table 10
Performance of different approaches for Long Method detection. We report the precision, recall, and F-measure of the minority (smell) class evaluated on the test set.

Denotement	Used features	Approach	Precision	Recall	F-measure
ML_code2vec	code2vec features	Random Forest + SMOTE	0.33	0.19	0.24
H_metrics	Code metrics	Heuristic-based approach (ANY)	0.86	0.33	0.48
ML_code2seq	code2seq features	Bagging (SVM classifier) + SMOTE	0.50	0.60	0.55
ML_metrics&votes	Code metrics + votes of heuristic detectors (LM_1 – LM_3)	Random Forest + SMOTEENN	0.63	0.63	0.63
ML_metrics	Code metrics	Random Forest + SMOTEEENN	0.63	0.70	0.67
ML_CuBERT	CuBERT features	XGBoost + SMOTEENN	0.70	0.81	**0.75**

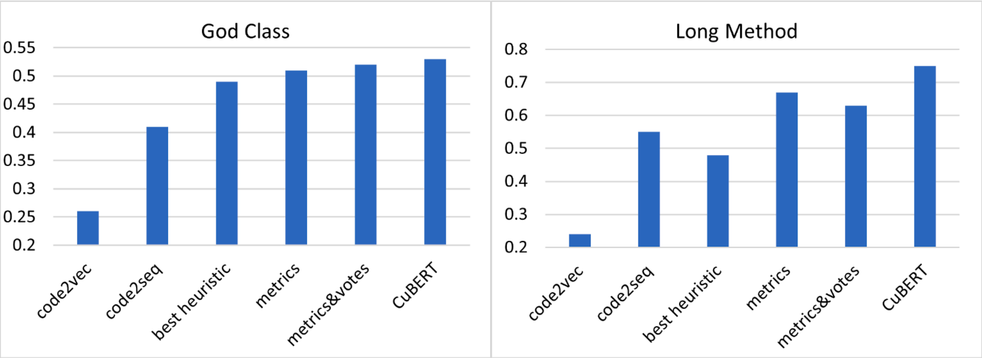

Fig. 6. F-measure of the minority (smell) class evaluated on the test set.

project. Regardless, we needed to examine the whole project to calculate some code metrics[10] accurately. Furthermore, we needed to include library classes, which was impossible for some projects due to parsing errors. Thus, some metrics may not have been accurately calculated for some code samples, affecting the metric-based detectors.

For example, in contrast to *H_metrics and ML_metrics, ML_CuBERT* correctly detects Long Methods characterized by too many words and wide expressions. While *H_metrics* does not rely on the number of unique words, *ML_metrics* used this metric as one of the predictor variables. However, our domain expert recognized that this quantity was miscalculated (had a low value) for a part of these samples.

Consequently, some metrics may not have been accurately calculated for some code samples, affecting the metric-based detectors. In contrast, CuBERT embeddings require only the analyzed code samples as input. Thus, calculating the CuBERT embedding is more straightforward and less bug-prone.

We identified a need for a systematic approach to code smell annotations. For example, there is no consensus on whether inner classes should be included in the outer class analysis. It is unclear how the MLCQ annotators handled this issue. Additionally, our domain expert stated some potentially relevant guidelines for annotating code smells during error analysis. For example, if the classes' complexity stems from a single Long Method, he would not label this class as a God Class. However, if the class has several Long Methods, he would label it as a God Class. Another guideline example would be to treat classes that follow design patterns differently when deciding on their smelliness. Developing a consensus on such guidelines, as is standard in the NLP field, would significantly improve the quality of the code smell datasets and consequently the state-of-the-art in this field.

Another related issue is that not all dataset examples are cross-checked and that there is a significant disagreement between the annotators, especially for the class-level smells. Additionally, our domain expert disagreed with the labels assigned to some of the code samples. These issues further indicate that we need more studies to derive a "science of annotation" in the code annotation field.

To summarize, we conclude that in our experiment, the best source code representation for code smell detection was CuBERT features for the following reasons:

- CuBERT features better captured the semantics needed for God Class and Long Method detection than code metrics
- Metrics extraction implementations are non-trivial as metric definitions can be ambiguous
- Metric extraction is time-consuming
- Metrics values may be miscalculated, which is hard to detect.

6. Threats to validity

Here we discuss the factors that might affect our conclusions: internal, external, and construct validity.

6.1. Threats to internal validity

Threats to internal validity are related to the correctness of our experimental results.

MLCQ dataset was labeled by 26 professional software developers of different backgrounds. However, not all instances were annotated by multiple developers, and, therefore, some assigned labels may be subjective. Another factor that might affect our experimental conclusions is the label aggregation strategy we used for the instances in the MLCQ dataset with multiple labels assigned by different annotators. In such cases, we opted to derive the ground truth for each instance by the majority vote, which is a widely accepted way to assign a label in these situations (Ide & Pustejovsky, 2017).

Though implementing the rules is simple, we may have misinterpreted metric descriptions or threshold values from the original paper or metric-extraction tools documentation. This issue could be mitigated by using publicly available and tested tools that perform code smell detection. However, the issue cannot be resolved entirely as programming languages construct constantly evolve. Consequently, code metric and heuristic designers cannot anticipate all possibilities and precisely define the metrics for every possible case.

[10] [10] The examples of metrics that cannot be calculated accurately by examining the code snippet alone but require examination of other related code snippets are DIT (Depth of Inheritance Tree), CBO (Coupling Between Objects), and RFC (Response for a Class).

Table 11
Comparing the F-measure of metric-based classifiers on the MLCQ dataset (our experiment) to performances reported in the original studies. In our study on MLCQ, the best F-measure ML_metrics achieved was 0.52 for God Class (GC) and 0.67 for Long Method (LM). The best heuristic for God Class detection achieved 0.49F-measure, and the best heuristic for Long Method detection achieved 0.48F-measure on the test set.

Proposed approach	Dataset[1]	Original study
Fontana et al. (2016): The authors used many code metrics as features (61 class-level and 82 method-level metrics). The authors also experimented with a wide range of ML classifiers. Here, we report the best performance. Experiment: 10-fold-cross validation.	Qualitas corpus – 420 instances per smell sampled from 74 open-source systems. Labeling procedure: semi-automatic. Cross-checked labels: yes. Annotator training: yes. The authors enforced an unrealistic ratio of code smells to non-smells on both the training and the test set.	**ML approach:** GC: 0.98 LM: 1.0
DiNucci et al. (2018): The authors replicate the study by Fontana et al. (2016) but correct the identified limitations.	The authors use the same dataset as Fontana et al. (2016) but correct the artificial smell to the non-smell ratio on the whole dataset.	**ML approach:** GC: 0.58 LM: 0.48
Pecorelli et al. (2020): This study compares the performance of ML-based and heuristic detectors. In the ML-based approach, authors use 17 code metrics as features. These were the same metrics used in heuristics. These were the same metrics used in heuristics. The authors provide an online appendix for experiment replication. However, the appendix contains the extracted metrics without the source code needed to run our experiments. Though the dataset consists of open-source projects, we lacked the details necessary for reliable matching with the source code. Experiment: 10-fold-cross validation.	Palomba et al. (2018a) – 125 releases of 13 open-source projects. Labeling procedure: semi-automatic. Annotator training: no. Cross-checked labels: yes.	**ML approach:** GC 0.41 LM: 0.23 **Heuristics:** GC: 0.16 LM: 0.44

[1] "Semi-automatic" labeling procedure denotes that heuristic detectors provided a list of code smell candidates that annotators manually checked for false positives. Such a procedure does not ensure there are no false-negative instances. "Cross-checked labels" denotes whether multiple annotators examined each instance to avoid labeling subjectivity. "Annotator training" refers to whether the annotators received guidelines, training, and practiced a proof-of-concept annotation before annotating code smells.

To evaluate our ML models, we performed the stratified single hold-out validation. We recognize that the repeated hold-out validation would yield the robust performance estimate, less variant to splitting the data into training and test sets. However, such a procedure would be highly time-consuming, and thus it would be infeasible to test all model variants considered in our experiment. We, however, note that our approach of having a single benchmark (test) set to compare different approaches is a standard ML model validation setting. For example, in Computer Vision and NLP, public leaderboards are considered necessary incentives for active development of the field as they provide standard benchmarks for fair model comparison. We note that some SuperGLUE benchmarks used for language understanding are comparable in size to the MLCQ dataset we use in our study (Wang et al., 2019).

Finally, a single domain expert performed the error analysis, and therefore, our error analysis conclusions are susceptible to subjectivity.

6.2. *Threats to external validity*

Threats to external validity regard the generalizability of our experimental results. We used the MLCQ dataset that encompasses many active industry-relevant, contemporary projects. These projects are of varying sizes and encompass different domains. These factors mitigate the issue of generalizability. However, all projects are open-source and written in Java programming language, and we cannot guarantee that our conclusions generalize to other industry projects written in other programming languages.

6.3. *Threats to construct validity*

Construct validity concerns the relationship between the theory and empirical observations.

In our experiments, we used multiple metric-based heuristics to show the effectiveness of the ML approach. As discussed in section 4.1, applying metric-based heuristics is brittle due to possible differences between metrics definitions and their implementations. Threshold values defined by the heuristic authors might be inadequate if the metric definition differs from its implementation, consequently affecting the measured performance of heuristics. We mitigated this threat by choosing metric extraction tools that are open-source and documented. Thus, we could examine whether the metric implementation corresponds to its definition. We eliminated the heuristic where authors did not clearly define the metrics they relied upon. However, metrics definitions have ambiguities that affect their implementations. For example, it is unclear whether the properties of an inner class should affect the metrics calculated for the outer class.

To calculate the required metrics, we used publicly available open-source metric extraction tools used by other researchers in the field. However, the MLCQ dataset encompasses many projects (792), and some metric values may have been miscalculated due to parsing errors. Indeed, while performing error analysis, our domain expert identified a few instances for which the number of unique words was miscalculated. As we used the tools tested and accepted by the community, we hope that the number of miscalculated metric values is insignificant.

7. Conclusion

In this paper, we performed ML-based detection of God Class and Long Method code smells. We experimented with training ML classifiers on multiple source code representations: (1) pre-trained code2vec, code2seq, and CuBERT embeddings (2) a traditionally used representational vector of hand-crafted code metrics, and (3) a combination of source-code metrics and votes of popular smell detection heuristics. To test the effectiveness of our ML-based approach, we also included popular smell detection heuristics as baselines. We evaluated our approach on the MLCQ dataset, the largest, manually labeled, and fully reproducible dataset for code smell detection. We also performed an in-depth error analysis to understand the advantages and limitations of the considered approaches.

Our results indicate that pre-trained code2vec and code2seq embedding do not generalize well to the task of code smell detection. On the other hand, the ML model trained on the pre-trained CuBERT embeddings outperformed all other approaches. We also found that the ML model trained using the hand-crafted code metrics outperformed heuristic metric-based approaches.

Our error analysis argued the advantages of using automatically inferred CuBERT features to represent source code over the metric-based representation for code smell detection. We found the metric extraction process time-consuming, brittle, and unable to capture the semantics needed to detect code smells as well as CuBERT features.

To the best of our knowledge, this study is the first to evaluate the usefulness of pre-trained neural source code embeddings for code smell detection. This approach allows us to address two major limitations of the traditionally employed ML-based approach in which code metrics are used as features: (1) the metric extraction process is error-prone, time-consuming, and unscalable; (2) we are able to leverage the power of transfer learning – to the best of our knowledge, this is the first study to explore whether the knowledge mined from code understanding models can be transferred to code smell detection.

Our study is also the first to compare different code smell detectors on the same dataset. Almost every study that proposes a detection approach tests this approach on the dataset unique for the study – both in terms of the used codebase and the assigned labels. Thus, we cannot directly compare the reported performances to derive the best-performing approach. For example, as a consequence, researchers are still debating whether the ML approach for code smell detection genuinely outperforms a heuristic-based approach (Azeem et al., 2019; Pecorelli et al., 2020). Our study gives further proof in favor of the ML approach. To effectively measure the progress in the field, we need a single reproducible benchmark dataset to compare approaches fairly. In the paper, we argued that the MLCQ dataset we used in our study is the best available option.

Unfortunately, our findings also imply that MLCQ has two serious limitations. Firstly, not all labeled instances were cross-checked by different annotators. Secondly, the disagreement between the annotators is considerable as no instruction or annotation training was provided for the annotators. The authors omitted the annotator guidelines purposefully to infer developers' understanding of code smells. However, for ML, as taught from other fields such as NLP, datasets should have high-quality annotations which are acquired through providing well-defined guidelines, examples and training the annotators. Therefore, our findings imply that the field of ML-based code smell detection would benefit from a systematic approach to labeling code smells. We direct our future efforts towards developing a systematic approach for code smell labeling, inspired by the guidelines in NLP.

Funding

This research was supported by the Science Fund of the Republic of Serbia, Grant No 6521051, AI-Clean CaDET. Our funders had no involvement in the study design, collection, analysis, and interpretation of the data, writing of the report, or the decision to submit the article for publication.

CRediT authorship contribution statement

Aleksandar Kovačević: Conceptualization, Methodology, Software, Validation, Formal analysis, Writing – original draft, Writing – review & editing, Supervision. **Jelena Slivka:** Conceptualization, Methodology, Software, Formal analysis, Writing – original draft, Writing – review & editing, Visualization, Funding acquisition. **Dragan Vidaković:** Conceptualization, Methodology, Software, Validation, Writing – original draft. **Katarina-Glorija Grujić:** Software. **Nikola Luburić:** Conceptualization, Formal analysis, Project administration, Writing – review & editing. **Simona Prokić:** Conceptualization. **Goran Sladić:** Writing – review & editing.

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.

Acknowledgements

This research was supported by the Science Fund of the Republic of Serbia, Grant No 6521051, AI-Clean CaDET.

References

Abbes, M., Khomh, F., Gueheneuc, Y. G., & Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering* (pp. 181–190). IEEE.

AbuHassan, A., Alshayeb, M., & Ghouti, L. (2021). Software smell detection techniques: A systematic literature review. *Journal of Software: Evolution and Process, 33*(3), e2320.

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR), 51*(4), 1–37.

Alon, U., Brody, S., Levy, O., & Yahav, E. (2018). code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400.* Implementation of code2seq provided by the authors is available at https://github.com/tech-srl/code2seq Accessed: July 27, 2021.

Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, *3*(POPL), 1-29. Implementation of code2vec provided by the authors is available at https://github.com/tech-srl/code2vec Accessed: July 27, 2021.

Aniche, M. (2015). Java code metrics calculator (CK). Retrieved from https://github.com/mauricioaniche/ck/ Accessed July 27, 2021.

Arumugam, L. (2020). *Semantic code search using Code2Vec: A bag-of-paths model.* University of Waterloo. Master's thesis.

Azeem, M. I., Palomba, F., Shi, L., & Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology, 108*, 115–138.

Bafandeh Mayvan, B., Rasoolzadegan, A., & Javan Jafari, A. (2020). Bad smell detection using quality metrics and refactoring opportunities. *Journal of Software: Evolution and Process, 32*(8), e2255.

Bakarov, A. (2018). A survey of word embeddings evaluation methods. *arXiv preprint arXiv:1801.09536*.

Barbez, A., Khomh, F., & Guéhéneuc, Y. G. (2019, September). Deep Learning Anti-patterns from Code Metrics History. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 114-124). IEEE.

Ben-Nun, T., Jakobovits, A. S., & Hoefler, T. (2018). Neural code comprehension: A learnable representation of code semantics. *arXiv preprint arXiv:1806.07336*.

Boutaib, S., Bechikh, S., Palomba, F., Elarbi, M., Makhlouf, M., & Said, L. B. (2021). Code smell detection and identification in imbalanced environments. *Expert Systems with Applications, 166*, Article 114076.

Briand, L. C., Wüst, J., Ikonomovski, S. V., & Lounis, H. (1999). May). Investigating quality factors in object-oriented designs: An industrial case study. In *Proceedings of the 21st international conference on Software engineering* (pp. 345–354).

Cairo, A. S., Carneiro, G., de Resende, A. M. P., & Brito, F. (2019). The influence of god class and long method in the occurrence of bugs in two open source software projects: An exploratory study. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering* (pp. 199–204).

Chen, T., & Guestrin, C. (2016). August). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (pp. 785–794).

Child, M., Rosner, P., & Counsell, S. (2019). A comparison and evaluation of variants in the coupling between objects metric. *Journal of Systems and Software, 151*, 120–132.

Coimbra, D., Reis, S., Abreu, R., Păsăreanu, C., & Erdogmus, H. (2021). On using distributed representations of source code for the detection of C security vulnerabilities. *arXiv preprint arXiv:2106.01367*.

Compton, R., Frank, E., Patros, P., & Koay, A. (2020). June). Embedding java classes with code2vec: Improvements from variable obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories* (pp. 243–253).

Danphitsanuphan, P., & Suwantada, T. (2012, May). Code smell detecting tool and code smell-structure bug relationship. In *2012 Spring Congress on Engineering and Technology* (pp. 1-5). IEEE.

DeFreez, D., Thakur, A. V., & Rubio-González, C. (2018). Path-based function embedding and its application to specification mining. *arXiv preprint arXiv:1802.07779*.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv: 1810.04805*.

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., & De Lucia, A. (2018, March). Detecting code smells using machine learning techniques: are we there yet?. In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)* (pp. 612-621). IEEE.

Fard, A. M., & Mesbah, A. (2013, September). Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (pp. 116-125). IEEE.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv: 2002.08155*.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., & Figueiredo, E. (2016). June). A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering* (pp. 1–12).

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

Fowler, M. (2018). Refactoring: Improving the design of existing code. *Addison-Wesley*. Professional.

Hoang, T., Kang, H. J., Lo, D., & Lawall, J. (2020). June). Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 518–529).

Hovy, E. (2010). Annotation. *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*.

Hozano, M., Garcia, A., Fonseca, B., & Costa, E. (2018). Are you smelling it? Investigating how similar developers detect code smells. *Information and Software Technology, 93*, 130–146.

Hussain, Y., Huang, Z., Zhou, Y., & Wang, S. (2020). Deep transfer learning for source code modeling. *International Journal of Software Engineering and Knowledge Engineering, 30*(05), 649–668.

Ide, N., & Pustejovsky, J. (Eds.). (2017). *Handbook of linguistic annotation (Vol. 1)*. Berlin: Springer.

Kanade, A., Maniatis, P., Balakrishnan, G., & Shi, K. (2020, November). Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning* (pp. 5110-5121). PMLR. Implementation provided by the authors is available at https://github.com/google-research/google-research/tree/master/cubert Accessed: July 27, 2021.

Kang, H. J., Bissyandé, T. F., & Lo, D. (2019). In *November). Assessing the generalizability of code2vec token embeddings* (pp. 1–12). IEEE.

Karampatsis, R. M., & Sutton, C. (2020). Scelmo: Source code embeddings from language models. *arXiv preprint arXiv:2004.13214*.

Kiefer, C., Bernstein, A., & Tappolet, J. (2007, May). Mining software repositories with isparol and a software evolution ontology. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)* (pp. 10-10). IEEE.

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

Lacerda, G., Petrillo, F., Pimenta, M., & Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software, 167*, Article 110610.

Lemaître, G., Nogueira, F., & Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research, 18*(1), 559–563.

Lewowski, T., & Madeyski, L. (2022). Code smells detection using artificial intelligence techniques: A business-driven systematic review. *Developments in Information & Knowledge Management for Business Applications*, 285–319.

Lincke, R., Lundberg, J., & Löwe, W. (2008). July). Comparing software metrics tools. In *Proceedings of the 2008 international symposium on Software testing and analysis* (pp. 131–142).

Liu, H., Ma, Z., Shao, W., & Niu, Z. (2011). Schedule of bad smell detection and resolution: A new way to save effort. *IEEE transactions on Software Engineering, 38*(1), 220–235.

Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y., & Zhang, L. (2019). Deep learning based code smell detection. *IEEE transactions on Software Engineering*.

Lozoya, R. C., Baumann, A., Sabetta, A., & Bezzi, M. (2021). Commit2vec: Learning distributed representations of code changes. *SN Computer Science, 2*(3), 1–16.

Luburić, N., Prokić, S., Grujić, K. G., Slivka, J., Kovačević, A., Sladić, G., & Vidaković, D. (2021). Towards a systematic approach to manual annotation of code smells.

Madeyski, L., & Lewowski, T. (2020). MLCQ: Industry-relevant code smell data set. In *Proceedings of the Evaluation and Assessment in Software Engineering* (pp. 342-347). Zenodo. https://zenodo.org/record/3590102#.YG2VzegzY2w Accessed July 28, 2021.

Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., & von Staa, A. (2012). March). Are automatically-detected code anomalies relevant to architectural modularity? An exploratory analysis of evolving systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development* (pp. 167–178).

Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Pearson Education.

Mashhadi, E., & Hemmati, H. (2021). Applying CodeBERT for Automated Program Repair of Java Simple Bugs. *arXiv preprint arXiv:2103.11626*.

Menshawy, R. S., Yousef, A. H., & Salem, A. (2021). In *May). Code Smells and Detection Techniques: A Survey* (pp. 78–83). IEEE.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

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 Transactions on Software Engineering, 36*(1), 20–36.

Neamtiu, I., Foster, J. S., & Hicks, M. (2005). May). Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories* (pp. 1–5).

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., & De Lucia, A. (2014). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering, 41*(5), 462–489.

Palomba, F., Di Nucci, D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D., & De Lucia, A. (2015, May). Landfill: An open dataset of code smells with public evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (pp. 482-485). IEEE.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., & De Lucia, A. (2018a). On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. *Empirical Software Engineering, 23*(3), 1188–1221.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., & De Lucia, A. (2018b). A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology, 99*, 1–10.

Pan, C., Lu, M., & Xu, B. (2021). An Empirical Study on Software Defect Prediction Using CodeBERT Model. *Applied Sciences, 11*(11), 4793.

de Paulo Sobrinho, E. V., De Lucia, A., & de Almeida Maia, M. (2018). A systematic literature review on bad smells—5 W's: Which, when, what, who, where. *IEEE Transactions on Software Engineering*.

Pecorelli, F., Di Nucci, D., De Roover, C., & De Lucia, A. (2020). A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. *Journal of Systems and Software, 169*, Article 110693.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., … Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *the Journal of machine Learning research, 12*, 2825–2830.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*.

Piotrowski, P., & Madeyski, L. (2020). Software defect prediction using bad code smells: A systematic literature review. *Data-Centric Business and Applications*, 77–99.

Pour, M. V., Li, Z., Ma, L., & Hemmati, H. (2021, April). A Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)* (pp. 36-46). IEEE.

Rabin, M. R. I., Mukherjee, A., Gnawali, O., & Alipour, M. A. (2020). November). Towards demystifying dimensions of source code embeddings. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages* (pp. 29–38).

Rabin, M. R. I., Bui, N. D., Wang, K., Yu, Y., Jiang, L., & Alipour, M. A. (2021). On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology, 135*, Article 106552.

Rasool, G., & Arshad, Z. (2015). A review of code smell mining techniques. *Journal of Software: Evolution and Process, 27*(11), 867–895.

Sabetta, A., & Bezzi, M. (2018, September). A practical approach to the automatic classification of security-relevant commits. In 2018 IEEE International conference on software maintenance and evolution (ICSME) (pp. 579-582). IEEE.

Sharma, T., & Spinellis, D. (2018). A survey on software smells. *Journal of Systems and Software, 138*, 158–173.

Sharma, T., Efstathiou, V., Louridas, P., & Spinellis, D. (2019). On the feasibility of transfer-learning code smells using deep learning. *arXiv preprint arXiv:1904.03031*.

Souza, P. P., Sousa, B. L., Ferreira, K. A., & Bigonha, M. A. (2017). September). Applying software metric thresholds for detection of bad smells. In *In Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse* (pp. 1–10).

Trifu, A., & Marinescu, R. (2005, November). Diagnosing design problems in object oriented systems. In *12th Working Conference on Reverse Engineering (WCRE'05)* (pp. 10-pp). IEEE.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998-6008).

Yamashita, A., & Moonen, L. (2013). In *May). Exploring the impact of inter-smell relations on software maintainability: An empirical study* (pp. 682–691). IEEE.

Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., … Bowman, S. R. (2019). December). SuperGLUE: A stickier benchmark for general-purpose language understanding systems. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems* (pp. 3266–3280).

Wang, S., Huang, L., Ge, J., Zhang, T., Feng, H., Li, M., ... & Ng, V. (2020). Synergy between Machine/Deep Learning and Software Engineering: How Far Are We?. *arXiv preprint arXiv:2008.05515*.

Wang, Y., Gao, F., & Wang, L. (2021). Demystifying code summarization models. *arXiv preprint arXiv:2102.04625*.

Wu, J. (2021). Literature review on vulnerability detection using NLP technology. *arXiv preprint arXiv:2104.11230*.