Automatic detection of Feature Envy and Data Class code smells using machine learning

Milica Škipina [1], Jelena Slivka [2], Nikola Luburić [1], and Aleksandar Kovačević [1]

[1]Affiliation not available
[2]University of Novi Sad

October 30, 2023

Abstract

A code smell is a surface indication that usually corresponds to a deeper problem in the system. Detecting and removing code smells is crucial for sustainable software development. However, manual detection can be daunting and time-consuming. Machine learning (ML) is a promising approach towards the automation of code smell detection. The first ML-based methods were classifiers trained on feature vectors comprising software metrics extracted by off-the-shelf tools. Determining the optimal set of metrics is a complex problem that requires both ML and software engineering expertise. Recently source code embedding models emerged as a viable feature-inferring alternative. However, their potential is yet to be fully explored. To that aim, we compare state-of-the-art source code embedding models (CuBERT and CodeT5) with the models trained on metrics returned by the CK Tool and RepositoryMiner tools. We focus on detecting the Data Class and Feature Envy code smells within a large-scale, manually labeled, publicly available dataset. After extensive experiments (51 test/train splits), we found that source code embedding models have comparable performances with software metrics, a that they indeed can capture important characteristics of the source code. We discuss our findings in detail in the paper.

Automatic detection of Feature Envy and Data Class code smells using machine learning

Milica Škipina, Jelena Slivka, Nikola Luburić, Aleksandar Kovačević*

Faculty of Technical Sciences, University of Novi Sad, Serbia

Author Note

Correspondence concerning this article should be addressed to Aleksandar Kovačević, Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovića 6, 21 000 Novi Sad, Serbia. Email: kocha78@uns.ac.rs

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.

Abstract

A code smell is a surface indication that usually corresponds to a deeper problem in the system. Detecting and removing code smells is crucial for sustainable software development. However, manual detection can be daunting and time-consuming. Machine learning (ML) is a promising approach towards the automation of code smell detection. First ML-based methods were classifiers trained on feature vectors comprising software metrics extracted by of-the-shelf tools. Determining the optimal set of metrics is a complex problem that requires both ML and software engineering expertise. Recently source code embedding models emerged as viable feature inferring alternative. However, their potential is yet to be fully explored. To that aim, we compare state-of-the-art source code embedding models (CuBERT and CodeT5) with the models trained on metrics returned by the CK Tool and RepositoryMiner tools. We focus on detection of the Data Class and Feature Envy code smells within a large-scale manually labeled publicly available dataset. After extensive experiments (51 test/train splits) we found that source code embedding models have comparable performances with software metrics, a that they indeed can capture important characteristics of the source code. We discuss our findings in detail in the paper.

Keywords: code smell detection, neural source code embeddings, code metrics, machine learning, software engineering

1 Introduction

Code smells represent symptoms of poor design and implementation choices. The presence of code smells is positively corelated with software defects [1], and over time has a negative impact on understandability, extensibility, and maintainability of software projects [2].

Typically, code refactoring is a solution to the design problem coming from code smells [3]. However, detection of poorly written code is one of the major problems in the refactoring process [2]. Many different approaches have been proposed for code smell detection, from guided manual inspection to fully automated methods based on heuristics relying on various metrics derived from the source code or the development history [4]. Manual detection is difficult, time-consuming, prone to errors, and not feasible on the large-scale [2]. The problem with heuristic-based code smell detectors is that they require fine-tuned thresholds for the computed metrics. The selection of these thresholds strongly influences their accuracy, and the detected smells can be subjectively interpreted by developers [5].

To overcome the sensitivity related to thresholds of heuristic-based methodologies, researchers began experimenting with Machine Learning (ML) [5]. A crucial part of the application of ML models to detect code smells is creating a vector representation of the analyzed source code. Majority of the ML-based approaches rely on software metrics as the components (features) of the vector representation [5]. Additionally, researchers have tried to use the syntactic structure of source code (obtained from abstract syntax tree - AST) to produce alternative vector representations. Notable examples are code2vec [6] and code2seq [7]. Recently, inspired by the field of natural processing, pre-trained contextual code embedding models such as CuBERT [8] and CodeT5 [9] have emerged. These models aim to capture the semantics directly from the source code by treating it as a natural language. However, the effectiveness of the alternative source code representations has not been fully explored in the code smell detection task.

In our recent work we assessed the effectiveness of code2vec, code2seq, and CuBERT against source code metrics for the detection of God Class and Long Method code smells [10]. We found that ML model trained on the pre-trained CuBERT embeddings shows promising results by outperforming all other representations. Inspired by our previous findings, in this paper, we present a further study on the effectiveness of contextual source code embedding models on code smell detection. In this case we study the Data Class and Feature Envy code smells. Feature Envy is a high severity smell [12] that denotes a method that relies more on members of other classes than its own, while Data class is a class that is just a container for data, without any functionality. We consider these two smells because they are often found in combination and significantly increase the fault-proneness of the source code [14]. Besides CuBERT we explore the most recently published CodeT5 [9] source code embedding model. We used the same dataset as in our previous study, the MLCQ dataset [13] which is large-scale dataset containing 792 industry-relevant projects, manually labeled by developers with professional experience. We evaluated the ML models extensively with 51 different train-test splits and obtained inconsistent results. In case of the Feature Envy code smell the ML models trained with CodeT5 embeddings outperformed models trained using software metrics, while the results were opposite in case of the Data Class code smell. We provide a complete replication package to ensure the reproducibility of our findings[1].

The rest of the article is organized as follows. Section 2 describes the used methodology including selected target code smells, different ways of feature extraction, ML classifiers, and the experiment we designed

[1] Our replication package is avalilable at https://github.com/milica-skipina/ML-code-smell-detection.

to compare presented code smell detection approaches. Section 3 provides and discusses the results, while section 4 concludes our paper.

2 Experiment design

We compare the performance of the different source code vector representations using the same experimental setting. In this section, we present our experiment design. Section 2.1 explains why we chose the Data Class and Feature Envy code smells. Then, in section 2.2, we briefly describe the MLCQ dataset. Finally, in section 2.3 we explain our experiment design and chosen performance measure.

2.1 Targeted source code smells

According to a recent systematic literature review [11], God Class and Long Method, Feature Envy, and Data Class are the most frequently detected code smells. God Class and Data class are class-level smells, while Long Method and Feature Envy belong to the method-level bad code smells. Feature Envy is a high severity smell [12] and refers to a method that is more interested in members of other classes than its own. When this happens, it is a clear sign that it is in the wrong class [2]. The fault-proneness of the source code affected by Feature Envy is increases by a factor of 8 [14]. On the other hand, Data class smell refers to a class that serves only as a container for data, without any functionality. Generally, other classes are responsible for manipulating their data, which is then a case of Feature Envy [2]. Thus, these two smells are similar in a way that they are both the consequence of inappropriate use of class level attributes and methods between two classes [15].

2.2 The MLCQ dataset

We chose the MLCQ dataset [13] as it has number of advantages over other publicly available datasets. We discussed those advantages in detail in our previous study [10] while we briefly report them here. The MLCQ dataset is large (~15000 samples) as opposed to the dataset introduced by Fontana et al. [16] [17] with only several hundred samples per smell. It is also fully manually annotated in contrast to the Qualitas corpus [18] that is annotated automatically using heuristics. The balance of code smells is not artificially enforced which is a serious flaw of the Qualitas corpus [16].

The MLCQ data set was created by software developers with professional experience who reviewed 792 industry-relevant, contemporary Java open-source projects. The authors gathered code smell annotations from a total of 26 developers with professional experience (8 hired as senior developers, 4 hired as regular developers, 8 hired as junior developers and 6 with unknown background). The annotators manually assessed each code snippet on a four-level severity scale (critical, major, minor, and none). Four smells were considered, two on the class level (God Class and Data Class) and two at the method level (Feature Envy and Long Method).

2.2.1 Dataset pre-processing

Some samples were annotated by multiple annotators, and in some cases, there was a disagreement between their opinions. In such cases, we labeled these samples with the label which had the majority votes. We than filtered the dataset to include only Data Class and Feature Envy which resulted in 2159 distinct classes and 2242 distinct methods. Figure 1 shows the distribution of the severity of the samples for the two code smells.

As Figure 1 illustrates, the dataset is highly imbalanced in terms of severity. Majority of samples for both Data Class (86.85%) and Feature Envy (97.06%) are labelled with severity 'none', indicating the absence of code smell. Severity labels indicating the presence of code smells, i.e., ‘minor’, ‘major’ and ‘critical’ have a very low percentage in the dataset, making the task of training a multi-label classifier infeasible. Thus, we decided to aggregate all the code smell indicating labels to one label e.g., marked with 1 and mark all the samples 'none' with 0. In this way, we framed the code smell detection task as a binary classification problem.

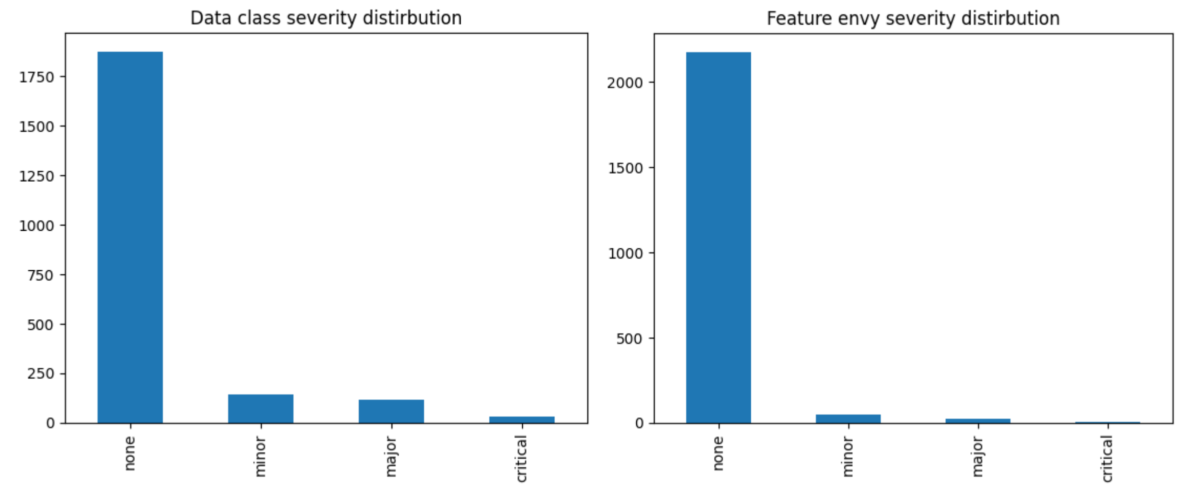

Figure 1 Distribution of the code smell severity of the samples for the Data Class and Feature Envy code smells

2.3 Feature extraction

As input to our models, we explored feature vectors comprising: (1) source code metrics extracted with the CK Tool [19] and RepositoryMiner [20] tools, as well as (2) neural source code embeddings extracted by CuBERT [8] and CodeT5 [9] models.

2.3.1 Source code metrics

In previous studies which were focused on code smell detection, code metrics were the most popular features for smell detection [5]. An advantage of using the code metrics as features for ML models compared to neural source code embeddings is that predictions are easily interpretable. Decisions made by ML models based on metrics can be analyzed given the values of feature importance as returned by the ML model. On the other hand, the disadvantage is that choosing the adequate metrics subset represents a challenging process that is time-consuming, requires domain expertise, and is problem-dependent [21]. In addition, the extraction of a particular subset usually requires applying and aggregating the results of multiple tools.

As noted in our previous work [10] we selected the CK Tool and RepositoryMiner for metric extraction based on the following reasons. The selected tools are open-source, well-documented and allow batch computation. They can perform metric extraction through static analysis to avoid compilation issues. The resulting output includes the Chidamber and Kemerer (CK) metrics [22] determined as most commonly used by recent systematic reviews [11] [5].

We used all extracted metrics from both aforementioned tools, which include a total of 46 class-level metrics for Data Class and 29 method-level metrics for Feature Envy code smells.

2.3.2 Neural source code embeddings

By neural source code embeddings we consider real valued vector representations of the source code generated by different types of neural networks. Recently, researchers showed that pre-trained language models based on the transformer architecture, such as BERT [23], transfer well to programming languages and largely benefit a broad set of source code related tasks [9]. One such example is CuBERT, a code understanding model inspired by BERT, proposed by Kanade et al. [8].

In our previous study [10], we compared ML models trained with CuBERT embeddings with models trained with two state-of-the-art embeddings popular at the time (code2vec [6] and code2seq [7]). The comparison was performed on the task of automatic detection of Long Method and God Class code smells. As CuBERT significantly outperformed the other two neural source code embedding models, we decided that, for the purpose of this study, to include only CuBERT in our experiments.

Most recently, Wang et al. introduced the CodeT5[2] [9]. It was inspired by the T5 architecture from the NLP field [24], and pre-trained on 8.35M functions in 8 programming languages (Python, Java, JavaScript, PHP, Ruby, Go, C, and C#). It achieved state-of-the-art results on 14 sub-tasks on the CodeXGLUE [9] code intelligence benchmark [25]. Given its state-of-the-art performance and that it hast yet been tested on the code smell detection task, we decide to include the CodeT5 model in our experiments.

2.3.2.1 Aggregation of embeddings

In general, the pre-trained neural embedding models in NLP produce one vector representation for each sentence. To represent a passage or a document, the sentence level representations need to be aggregated to one vector, usually using the mean aggregation. Similarly, for source code, embeddings are returned at the level of one line of code. So, we experimented with mean and sum aggregation to produce the embedding vector for a class (in case of the Data Class smell) and for a method (in case of the Feature Envy smell). Additionally, we empirically found that CodeT5 can process a whole class or method a single input sequence (which is not typical) and decided to add embeddings produced in such way to our experimental setup. Finally, the authors of CodeT5 offered two variants based on the number of weights in the neural network, the CodeT5-base, and CodeT5-small. The embeddings included in our experiments are summarized in Table 1

Table 1 Different configurations of embedding models and aggregations of code line or code snippet level embeddings we consider in our experiment.

Model	**Aggregation**	**Vector length**	**Denoted as**
CuBERT	Mean	1024	CuBERT_avg
CuBERT	Sum	1024	CuBERT_sum
CodeT5-base	/	768	CodeT5_base
CodeT5-base	Mean	768	CodeT5_base_avg
CodeT5-base	Sum	768	CodeT5_base_sum
CodeT5-small	/	512	CodeT5_small
CodeT5-small	Mean	512	CodeT5_small_avg
CodeT5-small	Sum	512	CodeT5_small_sum

[2] https://github.com/salesforce/CodeT5

2.3.3 Additional post-processing

After the extraction and aggregation stages, all features (both metrics and embeddings) were scaled using z-normalization. Finally, to address the class label imbalance problem, we resampled the training portions of our data (see Section 2.5). We used a combined strategy of SMOTE (Synthetic Minority Over-sampling Technique) over-sampling and Edited Nearest Neighbor (ENN) under-sampling, realized with the SMOTEENN method from the imbalanced-learn package [26].

2.4 Machine learning classifiers

Based on the experiments in our previous study [10] and a general trend of ensembles and gradient boosting ensembles outperforming other models [11], we decided to experiment with the following models: Random Forest [27], Gradient Boosted Trees [28] and CatBoost [29] as well as with Bagging (using Support Vector Machine as the base algorithm) [30]. We used the classifier implementations from the Python libraries: Scikit-learn [31], XGBoost [32], and CatBoost [29].

Hyper-parameters of each model were optimized on the training set (see Section 2.5). We used 100 trials of Bayesian optimization, with stratified 5-fold-cross validation and macro F1-measure (see Section 2.5). The macro F1-measure is computed using the unweighted mean of F1-measures across the class label values. This method treats all class labels equally regardless of their frequency in the dataset.

2.5 Model evaluation

While cross-validation is a typically used model evaluation strategy, in a study by Tantithamthavorn et al., it was empirically found to be among the most biased and unreliable in the context of software defect prediction [33]. The most likely reason is the randomness with which the folds are sampled, which strongly impacts the resulting performance [34]. An alternative, more reliable evaluation strategy is to average the performances obtained in multiple trials where, in each trial, the data is split into different training and test sets [33] [35]. As recommended in [35], we opted for this strategy and used 51 trials, applying the 80/20 train/test split in each trial. The split is performed using random stratified sampling, as the class label is highly imbalanced. Table 2 presents the class label ratio in training and test sets.

Table 2 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.

Code smell	No. instances (pos/neg)	No. instances train (pos/neg)	No. instances test (pos/neg)
Data Class	221/1497	177/1197	44/300
Feature Envy	64/2167	51/1734	13/433

Due to high class label imbalance, we chose to measure the performance of machine learning models with the F1-measure of the positive class:

$$F1 = 2 \cdot \frac{precison \cdot recall}{precision + recall},$$

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

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.

3 Results

Table 3 and Table 4 summarize the performance of different feature extraction approaches. We only show the results of the best performing ML model, according to our evaluation strategy.

Table 3 The performance of ML models trained on different source code representations for the Data Class code smell detection. We report the precision, recall, and F-measure of the minority (smell) class label on the test set averaged over 51 trials.

Representation	**Model**	**Precision**	**Recall**	**F1-measure**		
				min	**max**	**mean ± std**
Code_metrics	Random Forest	0.58 ± 0.06	**0.67 ± 0.08**	0.47	0.70	**0.62 ± 0.05**
CodeT5_base	Bagging (SVM)	0.58 ± 0.09	0.60 ± 0.09	0.40	0.72	0.58 ± 0.08
CodeT5_base_avg	Bagging (SVM)	0.54 ± 0.08	0.58 ± 0.09	0.44	0.66	0.55 ± 0.05
CodeT5_base_sum	Bagging (SVM)	0.53 ± 0.06	0.59 ± 0.09	0.44	0.66	0.55 ± 0.05
CodeT5_small	Bagging (SVM)	0.58 ± 0.08	0.64 ± 0.07	0.49	0.74	0.60 ± 0.06
CodeT5_small_avg	Bagging (SVM)	**0.61 ± 0.07**	0.61 ± 0.08	0.49	0.71	0.61 ± 0.05
CodeT5_small_sum	Bagging (SVM)	0.59 ± 0.07	0.62 ± 0.08	0.47	0.70	0.60 ± 0.05
CuBERT_sum	Bagging (SVM)	0.51 ± 0.06	0.63 ± 0.07	0.48	0.67	0.56 ± 0.04
CuBERT_avg	Bagging (SVM)	0.55 ± 0.08	0.56 ± 0.10	0.37	0.68	0.55 ± 0.07

Table 4 The performance of ML models trained on different source code representations for the Feature Envy code smell detection. We report the precision, recall, and F-measure of the minority (smell) class averaged over 51 experiment runs on the test set.

Representation	**Model**	**Precision**	**Recall**	**F1-measure**		
				min	**max**	**mean ± std**
Code_metrics	Random Forest	0.16 ± 0.06	0.28 ± 0.14	0.00	0.33	0.20 ± 0.08
CodeT5_base	Random Forest	**0.17 ± 0.04**	0.50 ± 0.17	0.13	0.40	0.25 ± 0.06
CodeT5_base_avg	Random Forest	0.17 ± 0.06	0.34 ± 0.13	0.06	0.38	0.22 ± 0.08
CodeT5_base_sum	Bagging (SVM)	0.14 ± 0.05	0.41 ± 0.14	0.06	0.40	0.20 ± 0.06
CodeT5_small	Random Forest	**0.17 ± 0.04**	**0.55 ± 0.14**	0.14	0.41	**0.26 ± 0.06**
CodeT5_small_avg	Random Forest	0.15 ± 0.06	0.29 ± 0.13	0.04	0.36	0.20 ± 0.07
CodeT5_small_sum	Random Forest	0.16 ± 0.04	0.40 ± 0.11	0.11	0.34	0.22 ± 0.05
CuBERT_sum	Random Forest	0.14 ± 0.04	0.41 ± 0.18	0.08	0.35	0.21 ± 0.07
CuBERT_avg	Random Forest	0.13 ± 0.06	0.17 ± 0.07	0.00	0.32	0.17 ± 0.07

In case of the Data Class code smell (Table 3), the ML model trained with code metrics as features achieved the highest recall (0.67) and F1-measure (0.62). It is closely followed by the ML model trained with embeddings returned by the small variant of the CodeT5 embedding model. Surprisingly, models trained with features from large (in terms of trainable parameters) neural source code embedding models

(CuBERT and CodeT5_base) had ~6% lower F1-measure when compared to other feature extraction techniques.

In terms of the Feature Envy code smell, the ML model trained with embeddings from the small variant of the CodeT5 had the highest F-measure (0.26) and recall (0.55), closely followed by the base variant (F1-measure of 0.25). In contrast to Data Class, the model trained with software metrics had lower F1-measure (0.20) then almost all the other models.

Additionally, our results show a big difference in F1-measures between the two considered smells. While the ML models performed relatively well in case of the Data Class with F1-measure of 0.62, detection of Feature Envy proved to be more challenging with F1-measure of just 0.26.

In terms of the ML models, Bagging with SVM (for Data Class) and Random Forest (for Feature Envy) had consistently performed better than other for almost all the features.

4 Discussion

The primary goal of our study was to evaluate the potential of state-of-the-art neural source code embeddings as features for ML-based code smell detection. In Section 4.1, we analyze our findings regarding this goal. Next, in Section 4.2, we interpret the models based on source code metrics to find which metrics are most relevant for detecting Data Class and Feature Envy code smells. Finally, we found the Feature Envy smell much harder to detect than other smells (Table 5). In section 4.3, we analyze why this is the case.

4.1 Are neural source code embeddings a viable alternative to manually engineered features?

Based on the results of our experiments, we can conclude that neural source code embeddings are a promising alternative to traditionally used software metrics. The ML model trained with the embeddings generated by the CodeT5 model outperformed the ML model trained with software metrics by 6% in the case of the Feature Envy code smell and had only ~1% lower F1-measure than the metrics-based model, in the case of the Data Class code smell.

These results are in accordance with our previous studies [10] and [36]. In [10], on the same MLCQ dataset [13], the embeddings returned by the CuBERT model significantly outperformed code metrics for God Class and Long Method detection in Java. In [36], we used our code smell dataset [37] to compare code metrics and CodeT5 embeddings for God Class and Long Method detection in C#. Here, code metrics slightly outperformed CodeT5 embeddings. Looking at all our experiments, performance-wise, there is no clear winner between using code metrics or code embedding features for code smell detection. Thus, we must also consider additional criteria (other than performance) concerning extracting and using these features.

The advantage of code metrics features is straightforward model interpretation. However, we also found many disadvantages to using code metrics features. As we discussed in [10], we found the process of extracting code metrics extremely time-consuming – it was necessary to process a whole project to extract metrics for a few annotated code snippets from that project. More concerning, as the programming language constructs are constantly evolving, code metrics extraction tools need to be updated to account for the new constructs. When processing the MLCQ dataset, we encountered parsing errors and wrongly calculated metric values, which made this process brittle. Moreover, to update their code metrics extraction tools, developers must make pragmatic choices when implementing their algorithms, as metrics definitions need to be updated to accommodate new programming constructs. The community recognizes the problem that metrics tools show inconsistent results even for well-established code metrics [38].

We note that recently Madeyski and Lewowski [39] trained ML models using code metrics on the MLCQ dataset used in this study and outperformed our results (Table 5)[3]. The difference in performance most likely stems from using different sets of metrics – study [39] used a wider set of metrics then ours. In [10], we emphasized how code metrics did not capture all semantic necessary to detect code smells. Study [39] shows that including more metrics has a positive effect. However, the improvement is not huge – the reported performance is still too low for the detectors to be useful in practice[4]. Moreover, designing new metrics that can capture the necessary semantics and whose calculation can be automated is a challenging and time-consuming process. This process is also very problem-dependent and, therefore, not scalable.

As the developed smell detection models were trained using small datasets with very few smell instances, adding more labeled data is a promising way to improve the performance. Research across multiple fields has shown that traditional (shallow) ML methods often perform better than deep learning (DL) models when the available data is limited. However, with adding more data, shallow machine learning performance is known to plateau, while, in contrast, DL methods performance does not plateau and continues increasing [40]. We might expect better performance from neural code embeddings on future larger-scale datasets. In contrast to code metric features, DL models like neural code embeddings automatically infer features and should be able to adapt to the evolving programing constructs automatically.

Table 5 Comparison of our results to performance presented in [39].

Code smell	**Best approach from** [39]	**Our best approach**	**Best F-measure from** [39]	**Our best F-measure**	**Difference**
Feature Envy	PMD and CODEBEAT code metrics + Random Forest	CodeT5_small features + Random Forest	0.32	0.26	0.06
Data Class	PMD and CODEBEAT code metrics + Random Forest	CK tool and RepositoryMiner code metrics + Random Forest	0.63	0.62	0.01
Long Method	PMD and CODEBEAT code metrics + Random Forest	CuBERT features + XGBoost [10]	0.77	0.75	0.02
God Class	PMD and CODEBEAT code metrics + Random Forest	CuBERT features + Bagging [10]	0.57	0.53	0.04

Overall, based on our studies, we can conclude that features (embeddings) generated by pre-trained large transformer models represent a viable alternative to manually engineered features, i.e., software metrics. Feature inferring neural networks and, more recently, pre-trained language models are the preferred feature extraction models in computer vision, natural language processing, and reinforcement learning.

[3] To the best of our knowledge, the only other study that detects code smells using the same MLCQ dataset is [44]. However, our results are not directly comparable to theirs. They used a different severity aggregation technique, defined their binary classification problem differently (another grouping of severity classes), and relied on accuracy and precision as performance metrics. In their study, Madeyski and Lewowski [39] tried to compare their approach to [44] but could not replicate their results.
[4] In [36], on our C# dataset for God Class and Long Method smells, we estimated that the desirable performance would be around 0.9 F-measure, as this is the estimated human performance.

As our study shows, these models will likely be the state-of-the-art feature generation technique of the future for the code smell detection task.

From Table 3 and Table 4, we can see that in almost all experiments, the vector representations generated by the CodeT5 model proved to be of higher quality when compared to the ones returned by the CuBERT model. This result is expected, as CodeT5 is a more recent transformer-encoder architecture[5] that uses a bimodal input (source code and natural language), is trained on multiple programming languages, and more than one task is used for its pre-training[6]. In contrast, CuBERT is a transformer architecture that uses a unimodal input (source code), is trained on a single programming language, and uses a single pre-training task. In general, encoder-only transformer architectures perform worse on classification tasks. However, combining different modalities may improve the models' understanding of the code, and multilingual pre-training was shown to improve results compared to monolingual training [41]. These may be the reasons why CodeT5 yields a better code representation than CuBERT on the code smell detection task.

We also note that in our experiments, embedding whole methods proved better than embedding code snippets by combing code line-level embeddings. This finding indicates that CodeT5 can better describe the source code by observing the entire method than analyzing its parts. Finally, we observed that CodeT5 small embeddings mostly outperform models trained with CodeT5 base embeddings. A recent study, which investigated how well the pre-trained code models capture various source code properties, tested the influence of model size [42]. In most cases, they found that larger models outperform their smaller variants. However, they found that performance differences were slight in most cases.

4.2 Which code metrics most influence the Data Class and Feature Envy detection models?

This section analyses the feature (code metrics) importance for the Data Class and Feature Envy code smell detection models. Figure 2 presents the top 10 most important features for Data Class detection and Figure 3 presents the top 10 most important features for Feature Envy detection. The definitions of the featured code metrics are listed in Table 6.

Table 6 Definitions of the top 5 most important source code metrics for detecting Data Class and Feature Envy smells. *These metrics were extracted using the CK Tool [19].*

Metric	**Definition**
staticFieldsQty	Number of static fields
rfc *(Response for a Class)*	Number of unique method invocations
publicMethodsQty	Number of public methods
protectedFieldsQty	Number of protected fields
cbo *(Coupling between objects)*	Number of dependencies a class has (e.g., field declaration, method return types, variable declarations)
methodsInvokedQty	Number of all directly invoked methods
loc *(Lines of code)*	Number of the lines of code, ignoring empty lines and comments
variablesQty	Number of variables

[5] Transformer-encoder refers to the encoder part of the transformer architecture.
[6] CodeT5 pretraining tasks are Natural Language Processing (NLP) based, Code-Aware (CA), and Cross-Modal-Aware (CMA) [41].

Figure 2 shows that the *staticFieldsQty* metric was among the top five most important features for detecting Data Class for each of the 51 models (trained using different train/test splits). The second most important feature for the Data Class code smells was the *rfc* metric. Data Classes do not implement enough functionality but contain only fields and methods for accessing those fields (getters and setters). Therefore, our result is not surprising – the most important features for detecting Data Classes calculate the number of defined fields (*staticFieldsQty*, *protectedFieldsQty*, *cbo*) or methods (*rfc*, *publicMethodsQty*). This result assures us that our models consider the relevant features when deciding whether a class is a Data Class.

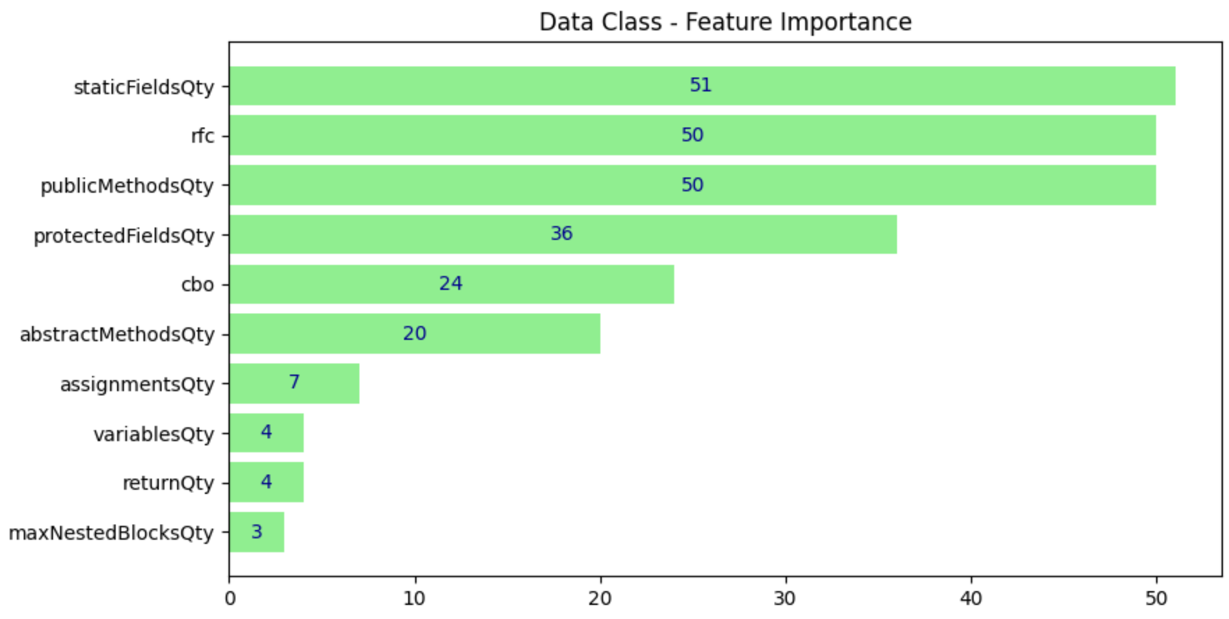

Figure 2 Feature (code metrics) importance for Data Class smell detection for the top 10 most important features. The numbers on each bar represent the number of times the metric was in the top 5 most important features for the best-performing model (in 51 training/test set splits).

For the Feature Envy code smell, *methodsInvokedQty* was among the top five most important features in all 51 experiments (3), followed by *rfc* (48 times amongst the top five most important metrics) and *cbo* (44 times amongst the top five most important metrics). Out of 51 trials, *methodsInvokedQty* was the most important feature in 44 trials, and *rfc* in 6, and, for the remaining model, the *code metric line*[7] was the most important. As the *code metric line* appeared important in a single trial, we might attribute this result to the specificity of the used (randomly sampled) training set and not consider it an important feature for Feature Envy detection in general. Methods that exhibit Feature Envy are those more interested in members of other classes than their own. It is reasonable that these methods will have a large value for the *methodsInvokedQty* metric, as they may invoke many methods from a different class. The *rfc* metric measures the number of unique method invocations. Upon closer inspection, we found that CK Tool returned identical values of *methodsInvokedQty* and *rfc* metrics in this dataset for all instances.

[7] *Code metric line* refers to the line of the class in which the method's definition starts.

We also note that none of the code metrics calculated by CK Tool and RepositoryMiner measured the times the method accessed fields and methods from another class[8], which should be important metrics for Feature Envy detection. As we already emphasized, using code metrics as predictors heavily relies on the code metrics extraction tools. We chose to use the code metrics extraction tools adopted by the community as they are thoroughly tested by the community in many different scenarios (code snippets containing various programming constructs). However, we found they may lack important metrics and may miscalculate metric values due to parsing errors.

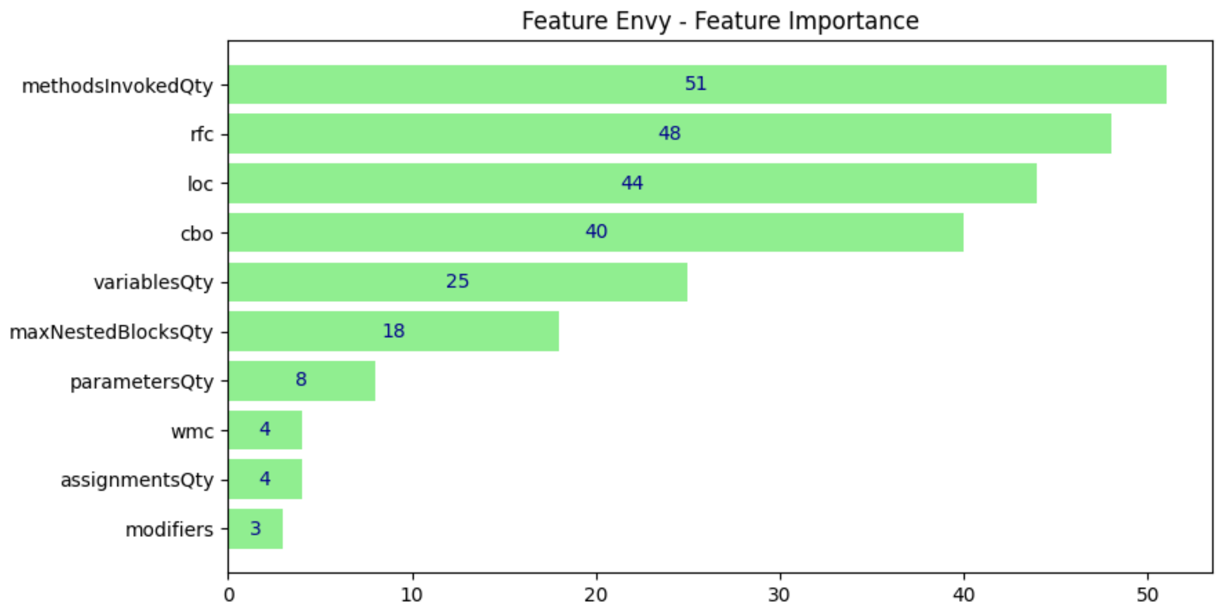

Figure 3 Feature (code metrics) importance for Feature Envy smell detection for the top 10 most important features. The numbers on each bar represent the number of times the metric was in the top 5 most important features for the best-performing model (in 51 training/test set splits).

4.3 Why is Feature Envy hard to detect?

For Feature Envy detection, the performance of our best-performing model, CodeT5_small, varies from 0.14 to 0.41 F1-score, depending on the generated train/test split. Low performance measured on particular test sets indicates that some smell instances are hard to detect. On the other hand, some specific training sets may be better as they contain samples that contribute to more accurate detection of Feature Envy.

The Code_metrics model achieved poor performance for the Feature Envy smell (0.20 ± 0.08 F1-score). We can explain this result by examining the most important feature of this model – the number of invoked methods (methodsInvokedQty and rfc metrics). With the increase of these metrics, a method is more likely to be classified as Feature Envy. However, these metrics count the number of methods an examined method invokes, regardless of whether the invoked methods are members of the same class as the examined method. As explained in Section 4.2, for Feature Envy detection, a better indicator would be to count solely the number of invocations of the methods belonging to other classes.

[8] The existing metrics are counts of method invocations regardless of invoked methods' class membership.

The next most important metric for Feature Envy detection is the number of lines the examined metric has (loc metric). This result can be explained by the correlation between the loc metric and *methodsInvokedQty* and *rfc* metrics –the more methods a method invokes, the more lines of code it has.

Finally, the annotation quality may be a significant contributor to ML models' performance [43]. Soomlek et al. [44] analyzed the annotator disagreement of the Feature Envy code smell on the MLCQ dataset used in this study. They found the disagreement was extremely high. For example, samples independently annotated by four annotators were assigned labels ranging from *none* to *major*. Soomlek et al. [44] created a ground truth label by assigning a numerical severity score to labels (*critical* = 3, *major* = 2, *minor* = 1, *none* = 0) and calculating the average severity score for the code sample. They found that the standard deviation of the Feature Envy label is 0.97 (calculated on 70 samples). The high average standard deviation shows high disagreement among human experts that annotated the dataset, which hurt the performance of the trained ML models. We encountered the same problem for God Class and Long Method detection on the MLCQ dataset [10].

A larger, more diverse dataset of smelly samples with high label quality is necessary to improve the automatic Feature Envy detection performance. The aim of the MLCQ study [13] was to elicit different viewpoints on code smell detection, which is why the authors purposefully omitted code smell annotation guidelines. While this approach yielded valuable insights into how developers of diverse backgrounds perceive code smells, such an annotation procedure needs to be revised for training high-performing ML models. It is essential to create a domain-specific dataset with high-quality annotations to advance the performance of automatic code smell detectors. In our work [45], we proposed an annotation methodology to develop such a dataset inspired by the well-defined corpus acquisition guidelines from the Natural Language Processing field [46].

5 Conclusion

In this study, we assessed the effectiveness of source code embedding models against the software metrics for the detection of Data Class and Feature Envy code smells. We experimented with two state-of-the art embedding models, CodeT5 and CuBERT. We used the MLCQ dataset, the largest, manually labeled, publicly available dataset for code smell detection. We analyzed the obtained results in-depth to understand the advantages and disadvantages of the considered code representation methods.

Our results show that, for the problem of detecting Data Class and Feature Envy code smells, ML models trained with neural source code embeddings code smell outperform or have very similar performance when compared to the models trained on software metrics. Taking into account the results of our previous studies [10] and [34], we have further confirmed the effectiveness of neural source code embeddings as a most likely feature generation technique alternative to the traditionally used software metrics.

In addition, our results confirm that some code smells are much harder to detected than others, which is in accordance with our previous study [10] and a most recent study [37] conducted on the same dataset. This is especially the case for Feature Envy where our best model achieved an F-measure of 0.26 while the authors in [37] reported an F-measure of 0.32. Further analysis revealed that additional efforts must be directed towards: (1) feature extraction, as both our features and the ones used in [37] were not enough to capture the variability of this smell, and (2) careful manual annotation, because Feature Envy was the source of most disagreement between the annotators of the MLCQ dataset [44].

Besides expanding the software metrics feature set and annotating more data our future work will be focused on unlocking the potential of the neural code embedding models through transfer learning and data augmentation. We plan to apply transfer learning by changing the architecture of the embedding models so that their primary becomes code smell detection, and then fine-tuning them on datasets such as MLCQ. We also plan to research ways of using these models to generate synthetic 'smelly' source code to expand (augment) our datasets.

6 Acknowledgements

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

7 References

[1] P. Piotrowski and L. Madeyski, Software Defect Prediction Using Bad Code Smells: A Systematic Literature Review, Springer International Publishing, 2020.

[2] G. Lacerda, F. Petrillo, M. Pimenta and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software,* vol. 167 , p. 110610, 2020.

[3] M. Fowler, Refactoring: Improving the Design of Existing Code, Springer Berlin Heidelberg, 2018.

[4] M. S. Haque, J. Carver and T. Atkison, "Causes, impacts, and detection approaches of code smell: a survey," in *Proceedings of the ACMSE 2018 Conference*, 2018.

[5] M. Azeem, F. Palomba, L. Shi and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis, Information and Software Technology," vol. 108 , pp. 115-138, 2019.

[6] U. Alon, M. Zilberstein, O. Levy and E. Yahav, "code2vec: Learning distributed representations of code," in *Proceedings of the ACM on Programming Languages, 3(POPL)*, 2019.

[7] U. Alon, S. Brody, O. Levy and E. Yahav, "code2seq: Generating Sequences from Structured Representations of Code," in *International Conference on Learning Representations*, 2018.

[8] A. Kanade, P. Maniatis, G. Balakrishnan and K. Shi, "Learning and evaluating contextual embedding of source code," in *International Conference on Machine Learning*, 2020.

[9] Y. Wang, W. Wang, S. Joty and S. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021.

[10] A. Kovačević, J. Slivka, D. Vidaković, K. G. Grujić, N. Luburić, S. Prokić and G. Sladić, "Automatic detection of Long Method and God Class code smells through neural source code embeddings," *Expert Systems with Applications,* vol. 204, p. 117607, 2022.

[11] A. Al-Shaaby, H. Aljamaan and M. Alshayeb, "Bad smell detection using machine learning techniques: a systematic literature review," *Arabian Journal for Science and Engineering,* vol. 45, no. 4, pp. 2341-2369., 2020.

[12] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014.

[13] L. Madeyski and T. Lewowski, "MLCQ: Industry-relevant code smell data set," in *Proceedings of the Evaluation and Assessment in Software Engineering*, 2020.

[14] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Software Engineering,* 2017.

[15] W. Wake, Refactoring workbook, Addison-Wesley Professional, 2004.

[16] F. Arcelli Fontana, M. Mäntylä, M. Zanoni and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering,* vol. 21 , pp. 1143-1191, 2016.

[17] F. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems,* vol. 128 , pp. 43-58, 2017.

[18] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia pacific software engineering conference*, 2010.

[19] M. Aniche, "Java code metrics calculator (CK)," 2015. [Online]. Available: https://github.com/mauricioaniche/ck/. [Accessed 27 07 2021].

[20] A. Barbez, F. Khomh and Y. G. Guéhéneuc, "Deep Learning Anti-patterns from Code Metrics History," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019.

[21] S. Wang, L. Huang, J. Ge, T. Zhang, H. Feng, M. Li, H. Zhang and V. Ng, "Synergy between machine/deep learning and software engineering: How far are we?," *arXiv preprint arXiv:2008.05515,* 2020.

[22] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering,* vol. 20 , pp. 476-493, 1994.

[23] J. Devlin, M. W. Chang, K. Lee and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805,* 2018.

[24] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li and P. Liu, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," *Journal of Machine Learning Research,* vol. 21, pp. 1-67, 2020.

[25] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang and G. Li, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[26] G. Lemaître, F. Nogueira and C. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *The Journal of Machine Learning Research,* vol. 18, no. 1, pp. 559-563, 2017.

[27] L. Breiman, "Random forests," *Machine learning,* vol. 45, no. 1, 2001.

[28] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016.

[29] L. Prokhorenkova, G. Gusev, A. Vorobev, A. Dorogush and A. Gulin, "CatBoost: unbiased boosting with categorical features," *Advances in neural information processing systems,* vol. 31, 2018.

[30] L. Breiman, "Bagging predictors," *Machine learning,* vol. 24, no. 2, pp. 123-140, 1996.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg and J. Vanderplas, "Scikit-learn: Machine learning in Python," *Journal of machine Learning research,* vol. 12, pp. 2825-2830, 2011.

[32] J. Chen, S. Burleigh, Chennupati, N. and B. Gudapati, "A Scalable Boosting Learner Using Adaptive Sampling," in *International Symposium on Methodologies for Intelligent Systems*, 2015.

[33] C. Tantithamthavorn, S. Mcintosh, A. Hassan and K. M. Matsumoto, "An Empirical Comparison of Model Validation Techniques for Defect Prediction," *IEEE Transactions on Software Engineering,* vol. 43, pp. 1-18, 2017.

[34] T. Hall, S. Beecham, D. Bowes, D. Gray and S. Counsell, "Developing fault-prediction models: What the research can show industry," *IEEE software,* vol. 28, no. 6, pp. 96-99, 2011.

[35] M. Kuhn and K. Johnson, Applied predictive modeling, vol. 26, New York: Springer, 2013.

[36] A. Kovačević, N. Luburić, J. Slivka, S. Prokić, K. Grujić, D. Vidaković and G. Sladić, "Automatic detection of code smells using metrics and CodeT5 embeddings: a case study in C#," *TechRxiv preprint,* 2022.

[37] N. Luburić, S. Prokić, K. Grujić, J. Slivka, A. Kovačević, G. Sladić and D. Vidaković, "Towards a systematic approach to manual annotation of code smells," *TechRxiv prerpint,* 2022.

[38] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software,* vol. 138, pp. 158-173, 2018.

[39] L. Madeyski and T. Lewowski, "Detecting code smells using industry-relevant data," *Information and Software Technology,* p. 107112, 2022.

[40] N. Sharma, R. Sharma and N. Jindal, "Machine learning and deep learning applications-a vision," *Global Transitions Proceedings,* vol. 2, no. 1, pp. 24-28, 2021.

[41] C. Niu, C. Li, B. Luo and V. Ng, "Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code," *arXiv preprint arXiv:2205.11739,* 2022.

[42] S. Troshin and N. Chirkova, "Probing Pretrained Models of Source Code," *arXiv preprint arXiv:2202.08975,* 2022.

[43] Y. Roh, G. Heo and S. E. Whang, "A survey on data collection for machine learning: a big data-ai integration perspective," in *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[44] C. Soomlek, J. Rijn and M. Bonsangue, "Automatic human-like detection of code smells," in *International Conference on Discovery Science*, 2021.

[45] N. Luburić, S. Prokić, K. Grujić, J. Slivka, A. Kovačević, G. Sladić and D. Vidaković, "Towards a systematic approach to manual annotation of code smells," *TexhRcxiv preprint,* 2022.

[46] N. Ide and J. Pustejovsky, Handbook of linguistic annotation, vol. 1, Dordrecht: Springer, 2017.