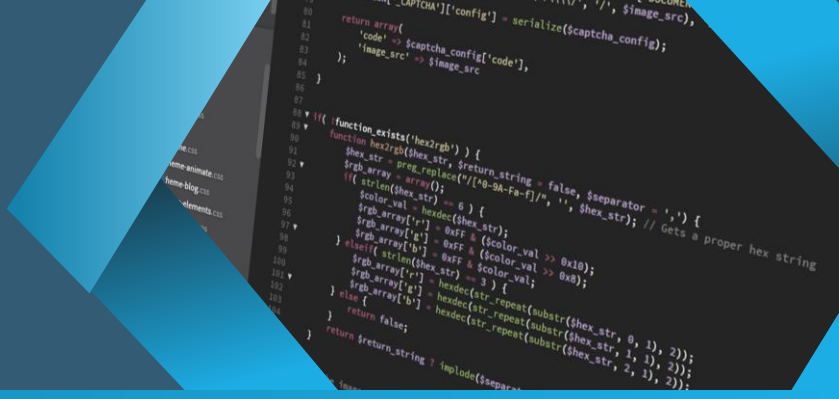
Clean CaDET

Clean Code and Design Educational Tool



REPORT ON THE SELECTED TARGET SMELLS

Executive Summary

In this document, we review the current state-of-the-art Artificial-Intelligence-based detectors for code smell detection. We focus on our target smells: God Class, Long Method, and Feature Envy. The reviewed approaches will serve as a guideline for designing our code smell detection methodologies. We also list the publicly available resources we may use to implement or train our code smell detectors. Finally, we present our plan for a systematic literature review paper on machine learning techniques for code smell detection.

Key Takeaways

1. *We list the publicly available resources for detecting our target smells (God Class, Long Method, and Feature Envy): datasets, tools for preprocessing and feature extraction, and implementations of the existing state-of-the-art detection approaches*
2. *We identified two publicly available manually labeled datasets that will use for evaluation of our smell detection approaches: Palomba et al. and MLCQ*
3. *There is a need for a systematic approach for annotating code smells. Current datasets are problematic due to the considerable disagreement of the annotators and some lack* information that allows *reliable reproduction*
4. *Feature envy may prove hard to tackle with a machine learning approach due to the lack of data. Thus, we will focus on God Class and Long Method smell and leave Feature Envy for the later phase of our project*
5. *We present the plan for a systematic literature review on machine learning techniques for code smell detection that we will conduct*

Contact
Jelena Slivka
slivkaje@uns.ac.rs
https://clean-cadet.github.io/

1 Introduction

In our previous report [1], we have identified God Class, Long Method, and Feature Envy as the most harmful types of code smells we plan to address in our project's first phase. In this document, we review the artificial intelligence approaches for the detection of these smells. We aim to identify:

- the state-of-the-art techniques for the detection of these smells,
- the limitations of the existing methods and possible ways to overcome them,
- publicly available resources we can use for their detection.

We will place a large part of our efforts on developing machine learning (ML) techniques for code smell detection. The last systematic literature review (SLR) on applying ML techniques to the code smell detection problem considers the papers published up to 2017 [2]. Many new articles were published since and evaluated on various datasets. Thus, to derive the state-of-the-art in this area, we need to perform a SLR of newly published approaches and a meta-analysis of their reported performances. In this document, we lay out our plans to conduct such a survey. This report showcases the preliminary results of this research.

In Section 2, we first analyze which information sources are useful for code smell detection in general and review methodological approaches that detect all of our three target smells. Then, in sections 3-5, we give a more detailed overview of the techniques directed at detecting particular target smells. In section 6, we list the publicly available resources for code smell detection we may use in our work. Finally, in Section 7, we present our plan for a systematic literature review of ML-based approaches for code smell detection.

2 Methodological approaches for code smell detection

The existing code smell detection approaches are based on analyzing the following sources of information:

1. *Software metric values*. Most of the code smell prediction models rely on source code metrics as predictors [2][3][4].
2. *Commit changes*. Several papers have shown that analyzing the historical trend of software metric values can help detect the God Class smell [5][6][7].
3. *Textual content [8].*
4. *Abstract Syntax Tree* (AST) [9].
5. *Hybrid approaches* [9][10]. The combination of several sources of information can be beneficial for code smell detection.

Additionally, some approaches detect the existence of code smell based on evaluating the presence of heuristics specific for a particular type of smell [11][12].

As code smell datasets are generally very imbalanced, Pecorelli et al. [13] examine the role of data balancing in ML-based code smell detection. They experimented with different balancing techniques and concluded that SMOTE (Synthetic Minority Over-sampling Technique) yields the best performance. However, in their experiment, data balancing has not dramatically improved models' performance.

In this section, we will explain methodological approaches aimed at detecting all three target smells, and in sections 3-5, we will review the techniques specific to a particular smell.

Fontana et al. [3] detect Data Class, God Class, Feature Envy, and Long Method using object-oriented software quality metrics as predictors. They pose the problem as binary classification, where each code snippet is either a smell or non-smell. They experiment with many ML models and report very high performance (accuracy higher than 96%). However, Di Nucci et al. later critiqued these findings [14]. The most significant limitation is that the test set used in [3] was unrealistically balanced, making the problem of smell detection significantly easier for ML models than it would be in practice. Another limitation is that the authors did not apply feature selection to examine which quality metrics are most relevant for detecting particular code smells. A later study by Di Nucci et al. [14] found that approximately one-third of the total features is relevant. The consequence is that most of the original dataset metrics did not impact code smell detection but may have caused model overfitting.

Palomba et al. [8] perform the code's textual analysis to detect the code smells related to promiscuous responsibilities. The primary assumption in their approach, called TACO, is: "the higher the number of the responsibilities implemented in a code component, the higher the probability that such a component contains heterogeneous identifiers and/or comments." Thus, TACO first performs series of information retrieval normalization steps on source code snippets to produce a vector representation of each code snippet. Then, TACO applies heuristics specific for a particular code smell on the obtained vector representation to detect the smells.

Hadj -Kacem and Bouassida [9] detected the God Class, Feature Envy, and Long Method smells by combining two sources of information: structural and semantic. As structural features, they used software quality metrics extracted using the Metrics tool[1]. They derived the semantic features by applying a Variational Auto Encoder to AST. They applied their approach to the five projects from the Landfill dataset [25]. However, the paper does not list the experimental setup details (i.e., how is the data divided into training and test set).

In [15] and [16], Liu et al. propose a code smell detection approach based on deep learning. Deep learning models typically require large annotated datasets for training. However, as labeling code smells is challenging, the available manually labeled sets may be insufficiently large. Thus, Liu et al. proposed a procedure for automatically generating labeled data. They use code snippets extracted from high-quality open-source applications as negative samples (code samples not suffering from smells) and apply smell-specific augmentation procedures to those samples to obtain positive samples. To create Feature Envy samples, they randomly move the methods to other classes. They make the Long Method samples by finding all method invocations that could be inlined by inline method refactoring and create the God Class samples from pairs of classes that can be merged. This study's limitation is the evaluation procedure performed on the automatically generated dataset that may be noisy. A manual evaluation is performed but in a relatively small case study. However, the proposed procedure holds great potential as a dataset augmentation procedure for deep learning applications in the code smell detection field.

[1] http://metrics.sourceforge.net/

3 God Class detection

Pantiuchina et al. [5] presented COSP (COde Smell Predictor), a tool for preventing code smells, which predicts whether a given class is likely to be affected by code smells within *t* days. One type of code smell that this tool can predict is the God Class. The authors defined the God Class as a procedural-style design class with many different responsibilities, monopolizing the system's logic.

To implement this approach, they used Weka's[2] implementation of the Random Forest. They used the metrics described in Table 1 as predictor values. These 14 metrics were used to characterize the quality of each class from three different perspectives:

- Current code quality (metrics' values at the prediction date)
- Historical code quality (metrics' values in all commits performed until the prediction date; for each metric, they computed regression slope line, which can highlight a continuing degradation of some quality aspect)
- Recent code quality (regression slope line focusing on the last *l* changes)

Described perspectives lead to 42 features used by the Random Forest algorithm to identify classes that would potentially become smelly. To deal with the unbalanced dataset, the authors used the sub-sample re-balancing technique.

Table 1. Class quality metrics used by COSP [5] for detecting the God Class smell

Metric	Description
CBO	Coupling Between Object classes: measures the dependencies a class has [17]
DIT	Depth of Inheritance Tree: the length of the path from a class to its farthest ancestor in the inheritance tree [17]
NOC	Number Of Children (direct subclasses) of a class
NOF	Number Of Fields declared in a class
NOPF	Number Of Public Fields declared in a class
NOSF	Number Of Static Fields declared in a class
NOM	Number Of Methods in a class
NOPM	Number Of Public Methods in a class
NOSM	Number Of Static Methods in a class
NOSI	Number Of Static Invocations of a class
WMC	Weighted Methods per Class: Sums the cyclomatic complexity of methods in a class [17]
RFC	Response For a Class: The number of methods in a class plus the number of remote methods that are called recursively through the entire call tree [17]
ELOC	Effective Lines Of Code: The lines of code excluding blank lines and comments
LCOM	Lack of Cohesion Of Methods: A class cohesion metric based on the sharing of local instance variables by the methods of the class [17]

Pantiuchina et al. evaluated COSP by applying it to six open-source projects for within-project evaluation[3]. A small number of projects can represent a constraint and negatively affect generalization. Another limitation of the study is that the authors used the predictions of the heuristic-based tools DECOR [18] as the ground truth rather than manual labels. Authors report overall precision of 75% and recall of 13%.

[2] https://www.cs.waikato.ac.nz/ml/weka/

[3] In within-project prediction, classifiers are trained to detect code smells using the data in an older version of the project and used to predict code smells in a newer version of the same project. The alternative is the cross-project prediction, where for the prediction of code smells of one project, we train the classifier using the other projects.

Barbez et al. [6] presented CAME[4] (Convolutional Analysis of code Metrics Evolution), a deep-learning approach that relies on the structural metrics' historical values for code smell detection. One type of code smell that this tool can predict is the God Class. The authors defined the God Class as an anti-pattern that refers to how a class grows rapidly with new functionalities. A God Class implements many responsibilities, delegating only trivial operations and accessing many other classes' data.

The authors built and trained ten distinct convolutional neural networks (CNNs) using the source code metrics extracted from commit history. The CNN models differ due to the randomness of their initialization. The final prediction is obtained by boosting, i.e., combining the predictions of the CNN ensemble.

To construct the input matrices for each CNN, the authors used RepositoryMiner[5]. Given a predefined history length *k,* RepositoryMiner calculates a set of quality metrics for each of the last *k* revisions of the projects' source code. Table 2 presents the quality metrics used in the CAME approach to detect the God Class smell.

Barbez et al. evaluated CAME on eight open-source projects listed in Table 3. The authors performed a cross-project prediction, where CAME was trained on five projects and evaluated on the remaining three. Annotations for the dataset were obtained from DECOR [18] and the re-implementation of the HIST [7] and verified by two master students.

Table 2. Structural metrics used by CAME [6] for detecting the God Class smell

Metric	Description
ATFD	Access To Foreign Data: Number of distinct attributes of unrelated classes (i.e., not inner- or super-classes) accessed (directly or via accessor methods) in the body of a class
LCOM5	Lack of COhesion in Methods: Measures cohesion among methods of a class based on the attributes accessed by each method [20] (used in [18])
LOC	Lines Of Code: Sum of the number of lines of code of all methods of a class
NAD	Number of Attributes Declared: Number of attributes declared in the body of a class (used in [18])
NADC	Number of Associated Data Classes: Number of dependencies with data-classes (i.e., data holders without complex functionality other that providing access to their data) (used in [18])
NMD	Number of Methods Declared: Number of non-constructor and non-accessor methods declared in the body of a class (used in [18])
WMC	Weighted Method Count: Sum of the McCabe's cyclomatic complexity [19] of all methods of a class

Table 3. Systems used by CAME [6]

System	Used for	Number of God classes
Android Opt Telephony	Training	10
Android Platform Support	Test	4
Apache Ant	Training	7
Apache Lucene	Training	3
Apache Tomcat	Test	5
Apache Xerces	Training	15
ArgoUML	Training	22
JEdit	Test	5

Barbez et al. compared the performance of the history-based CAME approach against three ML classifiers' performance (Decision Tree, Multi Layers Perceptron, and Support Vector Machine) trained using the metrics computed on the current revision of each studied system. The

[4] https://github.com/antoineBarbez/CAME/
[5] https://github.com/antoineBarbez/RepositoryMiner/

authors highlighted the positive impact of historical metrics on detection, noting that the accuracy improves, although the recall does not. With a metrics history of 500 revisions, the F-measure was increased by 33% $\pm$ 6%, compared to the performances achieved using a single revision.

An important limitation of the study is the size of the dataset and its small number of God Class instances.

In their textual analysis approach for code smell detection [8], Palomba et al. conjecture “that Blob classes are characterized by a semantic scattering of contents.”. Thus, they calculate the similarity of all pairs of methods in the class (using their vector representation). The average value of these pair-wise similarities is used as a measure of class cohesion, and the probability of the class suffering from the God Class smell is one minus this value.

4 Long Method detection

Section 2 explained Palomba et al. textual analysis approach to detect the code smells related to promiscuous responsibilities. Concretely, for Long Method detection, TACO relies on the following conjecture: "a method is affected by [the Long Method] smell when it is composed of sets of statements semantically distant to each other". Thus, TACO segments each into code blocks, i.e., sets of consecutive statements that logically implement an algorithmic step. Then, TACO calculates the textual cohesion as the average similarity between all pairs of vectors representing the methods' segments. The probability of the method suffering from a Long Method smell is calculated as one minus the calculated textual cohesion. Palomba et al. evaluate TACO on ten open-source projects from the Landfill dataset [25] and compare its performance to DÉCOR [18]. The evaluation setup is a within-project evaluation. In the experiment performed in [8], TACO achieves the overall f-measure of 76% for Long Method detection.

Tiwari and Joshi [22] proposed a functionality-based approach for Long Method detection. Functionalities within the method are identified through a novel block-based dependency analysis technique called *Segmentation* that clusters the sets of statements into extract method opportunities (or tasks). Then, the interdependencies among various extract method opportunities are identified. The interdependencies among subtasks of the method are used as a measure of severity of the long method smell.

5 Feature envy detection

In their textual analysis approach for code smell detection [8], Palomba et al. conjecture that “a method more interested in another class is characterized by a higher similarity with the concepts implemented in the envied class, with respect to the concepts implemented in the class it is actually in.” Thus, to detect whether a method suffers from Feature Envy, they calculate the similarity of the methods’ vector representation with vectors representing candidate “envied” classes. The probability of the method suffering from feature Envy smell is calculated by subtracting the highest calculated similarity from the similarity between the method and its belonging class.

Bavota et al. [10] analyze structural and textual information in the source code to identify move method opportunities for removing Feature Envy. They conjecture that the higher the overlap of

terms between comments and identifiers of method *m* and class *C*, the higher the likelihood they implement similar responsibilities. Thus, they develop an approach called Methodbook (an analog to Facebook) in which they treat methods as “people” and their implementations as their “profiles.” They create “friend” relationships between the methods by applying Relational Topics Models (RTM) to analyze the structural and textural properties of their “profiles.” Finally, given the method, Methodbook suggests that its target class is the class that contains the highest number of its “friends.”

Kiss and Mihancea [23] emphasize that detecting the Feature Envy code smell at the method level can be problematic as sometimes only the part of the method suffers from this smell [24]. Thus, they propose decomposing the method body into code blocks and applying the Feature Envy detection strategy to the obtained blocks. To detect code blocks, they recursively split the method body into sub-blocks following the compound statements' nesting structure, such as *if* and *for*. If no such statement exists, the block is split into sibling sub-blocks at the positions of each blank line. Then, to detect whether a specific code block suffers from Feature Envy, Kiss and Mihancea use structural metrics combined with the rule proposed in [24]. Finally, they group consecutive envious blocks to detect if the combined block is smelly or not. Their experiment showed that this technique is feasible and can spot the envious areas within a method, though not with a very high accuracy regarding the boundaries of envious areas.

6 Publicly available resources for code smell detection

This section analyzes the resources available for the detection of Long Method, Feature Envy, and God Class smells:

1. Publicly available datasets we can use for training our models
2. Implementations of the existing state-of-the-art detection approaches we may use as benchmarks
3. Publicly available tools we may use for data preprocessing or feature extraction
4. Publicly available tools able to detect the selected target smells we may use as baselines or utilize as an integral part of our models.

6.1. Publicly available datasets

As Azeem et al. pointed out [2], many researchers evaluated their ML-based approaches for code smell detection using the predictions of automatic heuristic-based detection tools as ground truth. However, this is far from ideal, as these tools suffer from a high rate of false positives (software artifacts wrongly detected as being affected by a smell) [21]. Thus, we opt to use manually labeled datasets for training and testing our ML-based models. In this section, we analyze publicly available manually labeled datasets we can use for training our models. We summarize the general properties of such datasets in Table 1.

Fontana et al. [3] labeled Qualitas corpus. They used five automated detectors to identify a set of code smell candidates. Three MSc students manually examined and labeled each code smell candidate. Finally, they created the dataset by selecting 420 instances per smell, ensuring that one-third of the instances have positive labels (represent code smells), while the remaining instances are negative. They set this distribution to create a relatively balanced dataset, which is preferred for training ML models. However, this artificially created distribution is unrealistic

[26] and might affect the obtained results' generalizability [28]. Thus, we avoid using this dataset to evaluate our approaches.

Palomba et al. labeled the Landfill dataset [25] by manually annotating 20 open-source Java projects to extract five types of code smells. First, one author identified the code smell candidates. After that, the second author checked the candidates and discarded false positive instances. Full manual labeling ensures the dataset has a naturally occurring positive to negative instance ratio. However, this dataset has several limitations. As the authors point out [25], the labeling procedure does not guarantee that the dataset does not contain false negatives. Also, the number of instances of each smell type is relatively small for training ML models (Table 1).

Later, Palomba et al. labeled a much larger dataset [26]. The labeling procedure was semi-automatic. To reduce human effort in the manual annotation of code smells, they first apply automatic heuristic-based code smell detection tools to filter instances. They adapted detection tools to have a very high recall, sacrificing precision. The filtered samples are automatically labeled as negative examples as they are highly improbable to suffer from the considered code smell. The rest of the instances are manually labeled.

Datasets [25][26] present an excellent approach for deriving the realistic distribution of code smells in software project and present a valuable resource for training ML models for code smell detection. However, these datasets have several important limitations:

- The annotation approach was not systematic. Smells were labeled by very few annotators without precise guidelines for code smell annotation.
- There is no guarantee that the dataset does not contain false negative instances [25][26].
- Although [26] is publicly available, it is not published in the form that allows completely reliable reproduction. We could not find the source code for all referenced releases and found inconsistences for some we could find[6]. As pointed out in [27], datasets should ideally be supplemented with information such as full class paths, commit hashtags, URLs, etc.

Thus, we plan to use the larger dataset [26] for evaluation of our ML-based approaches, but keep in mind that other datasets might be needed to prove the generalizability of our results.

MLCQ datasets [27] is the most diverse in the sense that it contains the largest number of projects. However, unlike [25][26], where whole projects are examined, in MLCQ, a random subset of code snippets is selected for manual annotation. A total of 26 annotators with different levels of experience annotated the projects. We plan to use the MLCQ dataset for the evaluation of our approaches. However, we need to bear in mind that the annotation procedure was not systematic. Firstly, not all instances are labeled by multiple annotators. Therefore, some annotations are susceptible to annotator subjectivity. Furthermore, annotators have different expertise levels and have received no training or guidelines on how to annotate code smells. Consequently, there is a considerable disagreement among annotators, and deriving the final annotation is not straightforward[7].

[6] For example, for some releases we downloaded, classes had different numbers of code lines than listed in the dataset. We had to discard inconsistent releases, which reduced the dataset size.

[7] For example, we have found examples where the same instance was labeled as ‘none,’ ‘none,’ ‘major,’ and ‘minor’ code smell violation by different annotators.

From Table 1, we may also observe that ML-based Feature Envy detection might be challenging due to the small number of available annotations. Thus, we will focus our efforts on developing ML methods for God Class and Long Method detection, which proved to be more harmful than Feature Envy [1]. Later, we will try to tackle Feature Envy by applying heuristic rules or developing a more comprehensive dataset.

In summary, we plan to use the MLCQ dataset and the dataset compiled by Palomba et al. [26] to evaluate our ML-based approaches for the detection of God Class and Long Method. However, we may conclude that a more systematic approach to dataset annotation is needed and a more comprehensive dataset that allows for reliable reproduction.

Table 1. Summary of publicly available manually labeled datasets for code smell detection

Dataset	Labeled smells	Label type	Projects	Number of positive instances
Qualitas corpus [3] https://essere.disco.unimib.it/machine-learning-for-code-smell-detection/	God Class Long Method Feature Envy Data Class	Severity (0 – 3)	112 open-source Java projects	420 labeled instances per smell
Landfill [25] http://soft.vub.ac.be/landfill/	5 types of code smells including God Class and Feature Envy	Binary	20 open-source Java projects	God Class: 92 Feature Envy: 86
Palomba et al. [26] https://dibt.unimol.it/staff/fpalomba/reports/badSmell-analysis/index.html	13 types of code smells including God Class, Long Method, and Feature Envy	Binary	395 releases of 30 open-source Java projects	[8]God Class: 1149 Long Method: 14767 Feature Envy: 511
MLCQ [27] https://zenodo.org/record/3666840#.YDN2HOhKg2x	God Class Long Method Feature Envy Data Class	Severity (0 – 4)	792 open-source Java projects	[9]God Class: 155 Long Method: 277 Feature Envy: 85

6.3. Publicly available tools for preprocessing and feature extraction

Process step	Tool	Functionality
Preprocessing	CodeSensor https://github.com/fabsx00/codesensor	Converts source code to Abstract Syntax Tree.
	Tokenizer https://github.com/dspinellis/tokenizer	Tokenizes source code into integer vectors, symbols, or discrete tokens.
	PyDriller https://github.com/ishepard/pydriller	Provides easy extraction of information from a Git repository: extraction of the commit message, the number of developers, modifications, diffs, source code of a commit, structural metrics (LOC and complexity).
	MethodDefragmenter https://github.com/arpadkiss29/MethodDefragmenter	Splits the methods' body into a hierarchical structure of blocks as proposed in [29].

[8] We will not be able to use all these annotations, as some referenced source code commits are missing, and we found inconsistencies in annotations and the source code for some commits we could download.
[9] Some instances are labeled by multiple annotators, and individual annotations differ. In such cases, we have determined the final label by a majority vote.

Feature extraction	CK metrics suite https://github.com/mauricioaniche/ck	Calculates class-level and method-level code metrics in Java projects (30 metrics).
	Metrics tool http://metrics.sourceforge.net/	Calculates class-level and method-level metrics in Java projects (17 metrics).
	Repository Miner https://github.com/antoineBarbez/RepositoryMiner/	Calculates class-level and method-level history of Java projects (12 metrics).
	JCodeOdor https://essere.disco.unimib.it/jcodeodor/	Calculates class-level and method-level metrics in Java projects (43 metrics).
	POM https://wiki.ptidej.net/doku.php?id=pom	Calculates class-level and method-level metrics in Java projects (60 metrics).
	Code2Vec https://github.com/tech-srl/code2vec	Represents a snippet of source code as a vector.
	Code2Seq https://github.com/tech-srl/code2seq	Represents a snippet of source code as a vector.
	CuBERT https://github.com/google-research/google-research/tree/master/cubert	Represents a snippet of source code as a vector.
Automatic Refactoring	Jdeodorant https://marketplace.eclipse.org/content/jdeodorant/	Provides *extract class* and *extract method* refactoring.

6.4. Publicly available tools for smell detection

Smell detection method	**Tool**	**Detected smells**		
		Feature Envy	Long Method	God Class
Heuristic-based	JDeodorant https://marketplace.eclipse.org/content/jdeodorant/	✓	✓	✓
	DÉCOR https://github.com/simgam/cASpER		✓	✓
	PMD https://pmd.github.io/			✓
	JSpIRIT https://sites.google.com/site/santiagoavidal/projects/jspirit	✓	✓	✓
	JCodeOdor https://essere.disco.unimib.it/jcodeodor/			✓
	MethodDefragmenter [30] https://github.com/arpadkiss29/MethodDefragmenter	✓		
History-based	CAME [6] and HIST [5] https://github.com/antoineBarbez/CAME/			✓
Text-based	TACO [8] https://github.com/simgam/cASpER	✓	✓	✓

7 Systematic literature review planning

The last Systematic Literature Review (SLR) on applying ML techniques to the code smell detection problem considers the papers published up to 2017 [2]. As many new articles were published, we decided to conduct a SLR on ML methodological approaches for code smell detection covering the papers published between January 2018 and January 2021. The research questions (RQs) we want to answer with this survey are:

RQ1. Which ML methodological approaches have been proposed in the literature for code smell detection?

RQ2. What are the key properties of data used to train and evaluate machine learning models?

RQ3. How was the proposed machine learning strategy evaluated?

For our research methodology, we use the SLR guidelines proposed by Kitchenham and Charters [31]. Figure 1 shows our selection process. We have collected the papers from several databases and are currently applying the exclusion criteria.

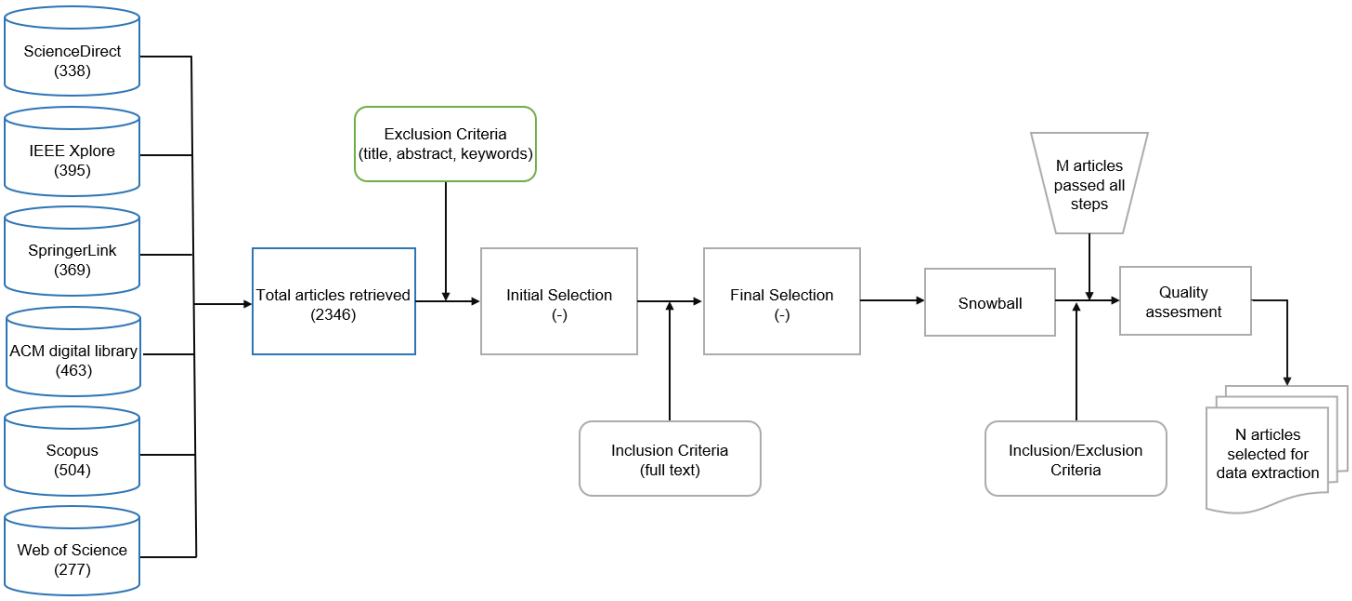


Figure 1. Article selection process

8 References

[1] Deliverables for the Clean-CaDET project funded by the Science Fund of the Republic of Serbia, Grant No. 6521051, AI - Clean CaDET https://github.com/Clean-CaDET/Deliverables/blob/main/work_package_1/D1.1.%20Report%20on%20the%20selected%20target%20smells.pdf

[2] Azeem, M.I., Palomba, F., Shi, L. and Wang, Q., 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology.

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

[4] Pecorelli, F., Di Nucci, D., De Roover, C. and 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, p.110693.

[5] Pantiuchina, J., Bavota, G., Tufano, M. and Poshyvanyk, D., 2018, May. Towards just-in-time refactoring recommenders. In Proceedings of the 26th Conference on Program Comprehension (pp. 312-315). ACM.

[6] Barbez, A., Khomh, F. and 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.

[7] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2013. Detecting bad smells in source code using change history information. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, 2013, pp. 268-278.

[8] Palomba, F., Panichella, A., De Lucia, A., Oliveto, R. and Zaidman, A., 2016, May. A textual-based technique for smell detection. In 2016 IEEE 24th international conference on program comprehension (ICPC) (pp. 1-10). IEEE.

[9] Hadj -Kacem, M. and Bouassida, N., 2019, December. Improving the Identification of Code Smells by Combining Structural and Semantic Information. In International Conference on Neural Information Processing (pp. 296-304). Springer, Cham.

[10] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. 2014. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. Software Engineering, IEEE Transactions on 40, 7 (July 2014), 671–694. https://doi.org/10.1109/TSE.2013.60

[11] Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y. and Zhang, L., 2019. Deep learning based code smell detection. IEEE Transactions on Software Engineering.

[12] Tiwari, O. and Joshi, R.K., 2020, February. Functionality Based Code Smell Detection and Severity Classification. In Proceedings of the 13th Innovations in Software Engineering Conference on Formerly known as India Software Engineering Conference (pp. 1-5).

[13] Pecorelli, F., Di Nucci, D., De Roover, C. and 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, p.110693.

[14] Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A. and 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.

[15] Liu, H., Xu, Z. and Zou, Y., 2018, September. Deep learning based feature envy detection. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (pp. 385-396). ACM.

[16] Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y. and Zhang, L., 2019. Deep learning based code smell detection. IEEE Transactions on Software Engineering.

[17] Shyam R. Chidamber and Chris F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. IEEE Trans. Softw. Eng (1994), 476–493.

[18] N. Moha, Y. Gueheneuc, L. Duchien, and A. Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Trans. Softw.Eng. 36, 1 (Jan. 2010), 20–36.

[19] Thomas J McCabe. A complexity measure. IEEE Transactions on software Engineering, (4):308–320, 1976.

[20] Brian Henderson-Sellers. Object-oriented metrics: measures of complex-ity. Prentice-Hall, Inc., 1995.

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

[22] Tiwari, O. and Joshi, R.K., 2020, February. Functionality Based Code Smell Detection and Severity Classification. In Proceedings of the 13th Innovations in Software Engineering Conference on Formerly known as India Software Engineering Conference (pp. 1-5).

[23] Á. Kiss and P. F. Mihancea, "Towards Feature Envy Design Flaw Detection at Block Level," 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 2018, pp. 544-548, doi: 10.1109/ICSME.2018.00064.

[24] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice. Springer Verlag, 2006.

[25] Palomba, F., Di Nucci, D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D. and 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.

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

[27] Madeyski, L. and Lewowski, T., 2020. MLCQ: Industry-Relevant Code Smell Data Set. In Proceedings of the Evaluation and Assessment in Software Engineering (pp. 342-347).

[28] Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A. and 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.

[29] L. Yang, H. Liu, and Z. Niu, “Identifying fragments to be extracted from long methods,” in Proceedings of the 16th Asia-Pacific Software Engineering Conference. IEEE, 2009, pp. 43 – 49

[30] Kiss, Á. and Mihancea, P.F., 2018, September. Towards Feature Envy Design Flaw Detection at Block Level. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 544-548). IEEE.

[31] B. Kitchenham, S. Charters, Guidelines for Performing Systematic Literature Re- views in Software Engineering, School of Computer Science and Mathematics, Keele University, UK, 2007.

The Project is funded by the Science Fund of the Republic of Serbia, Grant No. 6521051, AI - Clean CaDET

Science Fund of the Republic of Serbia