Classifying Code Smell Reviews with Semantic Search

Carlos Henrique Gonçalves Ribeiro
Federal University of Campina Grande
Campina Grande, Brazil
carlos.ribeiro@ccc.ufcg.edu.br

João Arthur Brunet Monteiro
Federal University of Campina Grande
Campina Grande, Brazil
joao.arthur@computacao.ufcg.edu.br

ABSTRACT

Background: Code smells refer to patterns in source code that deviate from established design principles. During code review, developers have the opportunity to identify and correct these smells, thereby enhancing the overall quality of the codebase. Further examination of the discussions within code reviews can reveal valuable insights about how code smells are discussed. **Aim:** In order to enable future research to better understand developers behavior regarding code smells, we set out to build a dataset of code-smell related discussions. In practice, we want to classify comments in two categories: code smell comments and non code smell comments. **Method:** To do so, we conducted an experiment that leveraged semantic search as a classification technique. The training data was scraped from three popular open source GitHub repositories and consisted of over 100,000 entries. **Results:** As a result, we have automatically classified 4,058 review comments as being code smell related. Although employing a novel technique and disposing of limited resources we could achieve a precision of 0.41 for the task of classification.

KEYWORDS

Code Smells, Code Review, Semantic Search, Classification.

1 INTRODUCTION

Code smells refer to patterns in source code that deviate from established design principles [2]. These deviations can be an indication of problems in the software development process, making them crucial to be addressed. During code review, developers have the opportunity to identify and correct these smells, thereby enhancing the overall quality of the software. Understanding how developers identify, discuss, and address these code smells can provide valuable insights for future research.

In order to enable future research to better understand developers behavior regarding code smells, we set out to build a dataset of code smell related discussions. This dataset can be used for various purposes, including training machine learning models aimed at automating software development processes and preempting the accumulation of code smells in future source code.

In a recent study by Fregnan et al [3]. it was analyzed how well a machine learning-based technique can automatically classify review changes. Subsequently, in Turzo et al. [9], the authors utilized Deep Neural Network to perform classification. By leveraging code context, comment text, and a set of code metrics they were able to reach the best accuracy of 59.3% with a model using CodeBERT.

Our methodology involved an experiment using semantic search as a classification technique. We gathered training data by scraping reviews from three popular open-source GitHub repositories: Neovim[1], Keycloak[2] and gRPC[3]. Using manually labeled data collected in Vitorino et al. [10], we embedded 3,798 code smell comments into a vector database. For each unclassified entry, we queried the database for the most similar entry and recorded the distance. Entries with distances shorter than a predefined threshold were classified as code smell reviews. Both manual and automatic validation were conducted to acquire results.

In this study, we have assembled a dataset containing 7,856, between manually (3,798) and automatically classified (4,058), code review comments that discuss code smells. Both the dataset of manually classified[4] and ML-classified reviews[5], as well as the scraping script[6] are made publicly available.

2 BACKGROUND

This section serves the purpose of explaining key concepts for this research, such as code smells, code review and semantic search.

2.1 Code Smells

Code smells, according to Martin Fowler [2], are usually the symptoms of underlying problems in software systems. Some of the most frequent code smells are:

- **Long Method:** Methods that are too long, are usually hard to understand, maintain, and reuse.
- **Data Clumps:** Groups of variables that are frequently used together should be encapsulated into their own class.
- **Shotgun Surgery:** When a change in one part of the codebase requires many small changes across multiple classes, indicating poor encapsulation and high coupling.
- **Large Class:** Classes that have too many responsibilities violate the Single Responsibility Principle.
- **Primitive Obsession:** Lack of domain-specific abstractions, can lead to code duplication.
- **Feature Envy:** If method in one class uses excessive data from another class it probably belongs there instead.
- **Switch Statements:** The presence of switch or case statements, might point to a violation of the Open/Closed Principle.
- **Duplicated Code:** Portions of code that are identical, which can difficult to maintenance and lead to inconsistency.
- **Long Parameter List:** Methods or functions that take too excessive parameters, makes code hard to read.

[1]https://github.com/neovim/neovim
[2]https://github.com/keycloak/keycloak
[3]https://github.com/grpc/grpc
[4]https://drive.google.com/file/d/1c4jOnaSmxsIG9ESeWbrs02B4MyVG4nJm/view?usp=sharing
[5]https://drive.google.com/drive/folders/1lQitzzkX5JtdRf0QLmaK5ziTEnNQrTbW?usp=sharing
[6]https://github.com/carloshgr/sniff

Sentence 1	It seems this method is not used and can be removed
Sentence 2	This method is no longer needed.
Cosine Distance	0,231611490249634

Table 1: Cosine distance between sentence embeddings

Here's an example of what a switch statement code smell might look like in Java:

```
public class PaymentProcessor {
    public void processPayment(Payment p) {
        switch (p.getType()) {
            case CREDIT_CARD:
                creditCardPayment(p);
                break;
            case DEBIT_CARD:
                debitCardPayment(p);
                break;
            case PAYPAL:
                payPalPayment(p);
                break;
            default:
                throw new Exception();
        }
    }

    private void creditCardPayment(Payment p) {
        // Logic to process credit card payment
    }

    private void debitCardPayment(Payment p) {
        // Logic to process debit card payment
    }

    private void payPalPayment(Payment p) {
        // Logic to process PayPal payment
    }
}
```

The code above violates the **Open/Closed Principle:** Adding a new payment method requires modifying the PaymentProcessor class, which can lead to it becoming bloated and harder to maintain over time.

However, it's important to note that not all code smells automatically denote problematic code. For example, while a large class might initially appear concerning, its presence may be justified if splitting it into smaller classes would decrease cohesion elsewhere. The identification and interpretation of code smells require nuanced judgment and context-specific analysis.

When code smells are correctly identified and addressed, there is an increase in code readability and maintainability [7, 8]. Moreover a decrease in software failure is to be expected. [5]

2.2 Code Review

Code review is a cornerstone practice in modern software engineering aimed at enhancing the overall quality of code artifacts. Through systematic examination and critique, code review endeavors to unearth bugs, ensure quality assurance, promote standardization, and facilitate knowledge sharing among programmers.

In essence, code review acts as a safeguard against the proliferation of code smells, offering a structured framework for engineers to detect, discuss, and address potential issues in the codebase. By harnessing the collective expertise of team members, code review serves as a crucial mechanism for maintaining code integrity and fostering continuous improvement.

Code Review has achieved it's current form with the rise of version control systems like Git[7] and GitHub[8]. Tools like those allow cooperation to occur through pull requests, where code can be discussed, verified and approved or disapproved. [6]

2.3 Semantic Search

In parallel, the realm of information retrieval has witnessed the emergence of semantic search techniques as a means of enhancing search engine performance. Unlike traditional lexical searches, which rely solely on word matching, semantic searches delve deeper by incorporating meaning and context into query analysis.

Bast et al. [1] divides semantic search in two, based in the kind of data used: search on text (natural language) and search on knowledge bases. Furthermore, the authors also divide semantic search by the type of query: keyword, structured and natural language. This study is focused in search on text using natural language queries.

According to Gao et al. [4] the following steps are encompassed by semantic search: First, the construction of a knowledge base. This involves transforming texts into real number vectors through an embedding model. Subsequently, fetching relevant information from this repository by comparing it with the user's query.

Embeddings represent sentences as a vectors. Ideally, sentences that have the same meanings have similar embeddings. By computing similarities between vectors representing query semantics and document content, semantic search algorithms offer more nuanced and contextually relevant search results.

The similarities between vectors will be calculated using the cosine distance:

$$d(x,y) = 1 - cos(\Theta) = 1 - \frac{x \cdot y}{\parallel x \parallel \parallel y \parallel}$$

In the above formula, **d** is a function that takes two sentence embeddings **x** and **y** and calculates the cosine distance between them. Furthermore, Θ is the angle between the two vectors, a smaller angle meaning that the sentences are more similar.

Table 1 displays two code review comments and the cosine distance calculated between their respective embeddings. Embeddings themselves are not shown due to their high dimensionality.

[7]https://git-scm.com/
[8]https://github.com/

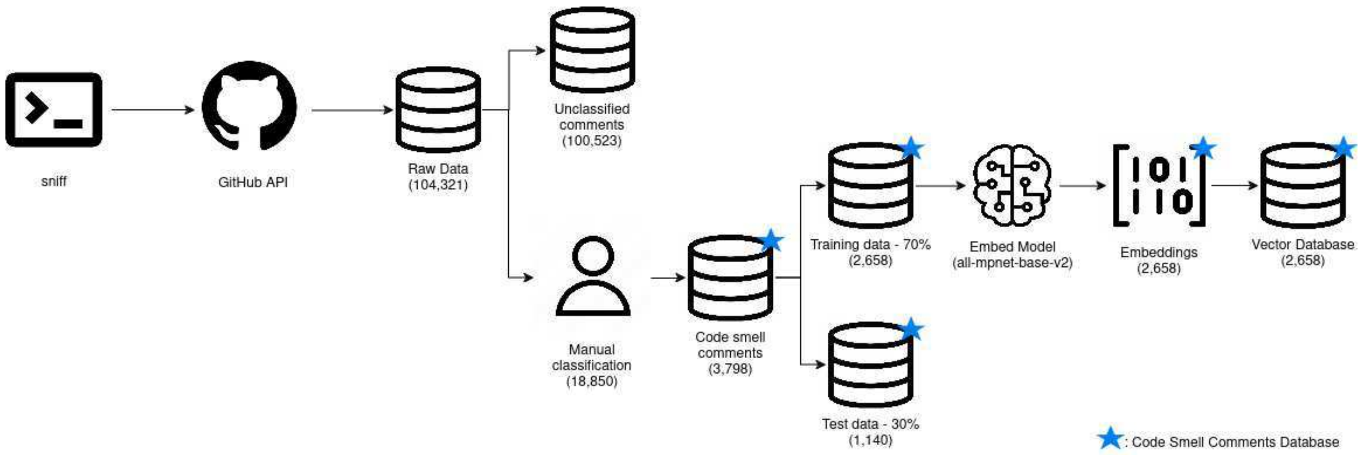
sniff
GitHub API
Raw Data (104,321)
Unclassified comments (100,523)
Manual classification (18,850)
Code smell comments (3,798)
Training data - 70% (2,658)
Test data - 30% (1,140)
Embed Model (all-mpnet-base-v2)
Embeddings (2,658)
Vector Database (2,658)
★: Code Smell Comments Database

Figure 1: Construction of the vector database

3 METHOD

This section describes the methodology employed to build the code smell reviews database, from the data mining to the automatic classification.

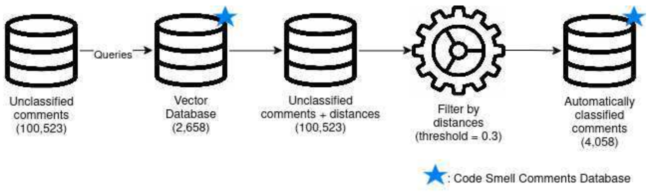
Unclassified comments (100,523)
Queries
Vector Database (2,658)
Unclassified comments + distances (100,523)
Filter by distances (threshold = 0.3)
Automatically classified comments (4,058)
★: Code Smell Comments Database

Figure 2: Automatic classification process

The initial step in this process, as depicted in Figure 1, involved building the vector database. Initially, the comments are gathered from online repositories via GitHub REST API[9]. A subset of these comments undergo manual classification by a team of software engineers. The manually classified comments are embedded into a vector database.

The automatic classification process is depicted in Figure 2. It consists of querying the vector database, saving the distance for each query, and filtering the queries based on a distance threshold. Code smell rich databases are highlighted.

Each following section contains a detailed description of one of the methodology steps.

3.1 Data Collection

Intending to gather raw data for classification, we opted for the GitHub REST API, due to its simplicity and availability. GitHub's API offers three kinds of comment views: comments on a specific commit within the pull request, comments on the pull request as a whole and comments on a specific line within the pull request. Conflicts in the manual classification phase were resolved by inspecting the source code, therefore we focus on comments on a specific line within the pull request.

[9]https://docs.github.com/en/rest?apiVersion=2022-11-28

We collected comments from three open-source repositories: Neovim, Keycloak and gRPC. The projects were selected based on criteria such as popularity, amount of pull requests and variety of programming languages used.

To interact with the API, we developed sniff, a script in Golang that leverages concurrency without incurring in penalties from the API. All of the raw data, consisting of 104,321 review comments, are also made publicly available[10].

3.2 Manual Classification

This phase was conducted as a part of another study, of which the author of this work was a participant. An overview of the steps followed to obtain the manually labeled data is provided below, while the detailed methodology can be found at Vitorino et al. [10] This labeled data will be used within this work to allow the automatic classification.

3.2.1 *Manual Independent Analysis.*

The first step of the manual classification phase consisted of a qualitative study involving 26 developers. The developers were divided in 13 pairs, each independently analyzing a subset of 1,450 comments. By the end of this step, the group classified a total of 18,850 comments, of which 4,563 comments were identified as related to code smells.

3.2.2 *Manual Simultaneous Analysis.*

To confirm previous findings, as well as resolving conflicts, two experienced developers carried out a new qualitative analysis. They examined entire comment threads and source code as necessary to simultaneously and manually analyze the results. At the conclusion of this process a total of 3,798 comments remained.

3.2.3 *Data Limitations.*

The manually classified dataset used in this study was analyzed and compiled in the context of a different study. Therefore, it lacks some information that could be useful to us. For example, all entries in the database belong to the same class: code smell comments.

[10]https://drive.google.com/drive/folders/1eaSEXzyEsjdcJkJz8xVS8vob7VXN77sO?usp=sharing

This limitation prevents us, for example, from automatically measuring precision. The measurement of precision in this article is done manually, by the author. Given the difficulty of measuring performance we could not optimize our classification technique based on the f1-score.

3.3 Automatic Classification

After acquiring the labeled data, we proceeded to classify the remaining entries. For that purpose we chose an innovative approach that leveraged semantic search as a classification technique.

Figure 3 displays a histogram of the distances of the queries for each project. We can see that the three of them follow a normal distribution with a mean around **0.45**.

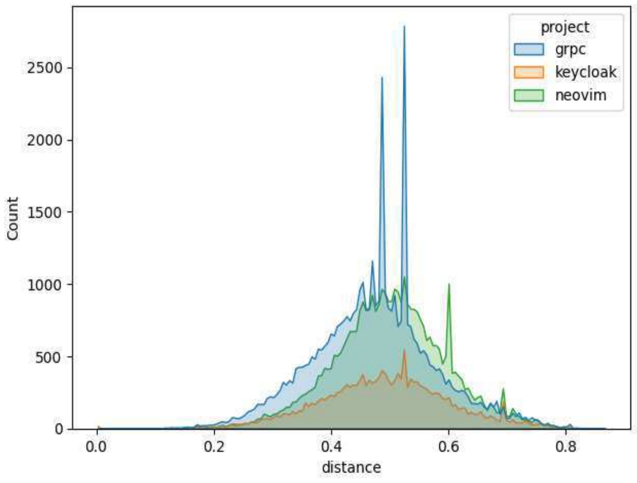

Figure 3: Frequency distribution of distances for each project

We provide the dataset containing all unclassified entries along with their corresponding distances prior to the filtering process as future research might benefit from choosing other threshold values.

3.3.1 Semantic Search.
Initially, we embedded all manually labeled data and imported them into a vector database. Afterwards, for each unclassified comment, we queried the database for the most similar entry. As a result, for each comment requiring classification, we obtained the nearest entry in the database along with the cosine distance between the two.

3.3.2 Threshold Selection.
In order to classify the comments, we needed to separate between two groups, code smell comments and non-code smell comments. To do so, we needed to choose a distance threshold.

Our aim during the threshold selection was to minimize the number of false positives in the dataset, whilst growing it as much as possible. With that in mind, we manually analyzed the distribution of distances between the entries in the dataset. After careful consideration, we identified a threshold value of 0.3 as optimal for our classification task. This threshold value was chosen based on its ability to yield the highest recall rate while maintaining a satisfactory level of precision.

3.3.3 Filtering.
In order to separate the two groups, we filtered entries based on our distance threshold (0.3): entries with distances to their closest entry in the database shorter than the threshold were classified as code smell reviews whereas entries with a distance greater than the threshold were classified as non code smell reviews.

3.3.4 Resources.
Our experiment was executed in Google Colab11[11], utilizing resources available for Colab Pro+ users. Specifically, an NVIDIA A100 40 GB PCIe GPU Accelerator12 along with 84 GB of RAM and 200 GB of disk space.

3.3.5 Tools.
As discussed before, we embedded the comments using a model. The model we chose for this task was all-mpnet-base-v2[12]. all-mpnet-base-v2 is the best pretrained model provided by the Sentence-Transformers library. According to the Sentence-Transformers benchmarks all-mpnet-base-v2 achieves 69.57 points of performance in encoding sentences over 14 tasks from different domains and 57.02 points on 6 diverse tasks of semantic search. Even so, all-mpnet-base-v2 maintains a low resource consumption, being able to encode about 2800 sentences/sec on a V100 GPU and being only 420 MB in size[13]. The code used to run the experiment is made publicly available.[14]

Moreover, for storing and querying the embeddings we utilized Chroma[15], an open-source embedding database. Chroma prioritizes simplicity, making easy embed documents and queries along with storing and searching embeddings.

4 RESULTS

The following section describes the results and analysis of the experiment.

4.0.1 Automatic Validation.
Aiming to measure the quality of the generated dataset, we separated 30% (1,140 entries) of the manually classified data for automatic validation. We then queried the database and recorded the distances, similarly to how we did for the unclassified entries. A total of 155 entries had distances shorter than the selected threshold (0.3) and were correctly classified as code smell comments.

With that in hand we were able to calculate the recall:

$$Recall = \frac{155}{1140}$$
$$Recall \approx 0.13$$

By choosing such a low threshold, recall was greatly affected. It was, however, necessary, given our objective of creating a dataset as free from false positives as possible.

As alluded to above, the entries in the manually classified dataset belong to only one class: code smell comments. This makes it impossible to automatically measure precision for both classes. A

[11]https://colab.research.google.com/
[12]https://huggingface.co/sentence-transformers/all-mpnet-base-v2
[13]https://www.sbert.net/docs/pretrained_models.html
[14]https://colab.research.google.com/drive/1ib0ucfTl71r9dRw9behgZ7535PI9xaX_?usp=sharing
[15]https://www.trychroma.com/

measurement of precision was done manually by the author of this work and is described in the following section.

4.0.2 *Manual Validation.*

With the automatic classified entries in hand, we proceeded to perform a manual validation. The author of this work took a random sample of 10% (404 entries) of the classified entries and checked if they were indeed code smell related. Table 2 displays precision results for each of the three projects.

	Precision
Neovim	0.52
gRPC	0.39
Keycloak	0.35
Aggregate	0.41

Table 2: Precision of our classification technique for each project

This phase was conducted solely by the author of this work. The spreadsheets in which results were annotated are made publicly available[16].

5 DISCUSSION & FUTURE WORK

There are many reasons for why the results may not be optimal. Working on these issues is an opportunity for future research. First, the data is not as clean as it could be. Several comments include snippets of code, URLs, paths to files and so on. Those less meaningful portions might degrade the performance of the model and reduce the quality of the generated embeddings.

Furthermore, it is still not clear if the model we employed for embedding the text is a good fit for our domain. Larger, more robust LLMs such as Meta's Llama 2[17] or Mistral 7B[18] may perform better at the cost of greater resource consumption.

Additionally, the choice of similarity metric utilized in our analysis presents another opportunity for exploration. While we relied on the cosine distance in our experiments, alternative metrics such as the euclidean distance and dot product similarity warrant investigation to assess their potential impact on result quality. Conducting comparative experiments with different similarity measures could elucidate the most suitable approach for our specific task and dataset.

Lastly, while semantic search holds promise for information retrieval tasks, its applicability to classification tasks remains uncertain. Future research could delve into comparative analyses, contrasting the performance of semantic search with traditional machine learning approaches such as K-nearest neighbors (KNN) and decision trees.

6 THREATS TO VALIDITY

The following section points out factors that might undermine the accuracy, reliability, and generalizability of our research findings.

[16]https://drive.google.com/drive/folders/1I87Eu6SfGbmqyswpaEH_ZncF2YZ5kuJx?usp=sharing
[17]https://llama.meta.com/llama2
[18]https://mistral.ai/news/announcing-mistral-7b/

6.1 Internal Validity

Internal validity refers to the extent to which a study accurately demonstrates a causal relationship between variables by ruling out alternative explanations. We can enumerate the following factors as potential threats to the internal validity of our study:

- **Selection bias:** The dataset was constructed using data from only three repositories, that may not represent the whole population of code reviewers.
- **Feature selection bias:** The embeddings may not be a good representation of the textual data.
- **Algorithmic bias:** Semantic search may not be a good fit for classification tasks.

6.2 Construct Validity

Construct validity refers to the degree to which a measurement tool or research study accurately assesses or represents the abstract concept or theoretical construct it claims to measure. One possible threat to the construct validity of our study is:

- **Ground Truth Annotation:** The code smell labels assigned to the manually classified dataset may be subjective and prone to annotation errors or disagreements among human annotators, leading to inconsistencies in the evaluation.

6.3 Statistical Conclusion Validity

Statistical conclusion validity pertains to the accuracy and appropriateness of the statistical analyses conducted in a research study and the subsequent conclusions drawn from those analyses. A potential statistical defect present in our work is:

- **Validation sample:** Our manual validation sample might not be a good representation of the whole population of classified comments.

7 CONCLUSION

We can conclude that code smells are still a rare topic of discussion during code review. Moreover, detecting code smell reviews still proves a hard challenge for machine learning algorithms to solve in the current state of research.

We hope that future research can leverage the dataset we produced for further improvement of machine learning techniques in the context of software engineering. Our dataset can be used for various purposes such as training data for classical machine learning algorithms and fine-tuning data for Large Language Models.

Beyond that, the data we've gathered can be a rich source of insights that future descriptive analysis can explore. Understanding the nuances of engineers behavior during code review is a theme still unexplored by current research.

REFERENCES

[1] Hannah Bast, Björn Buchhold, and Elmar Haussmann. 2016. Semantic Search on Text and Knowledge Bases. *Foundations and Trends® in Information Retrieval* 10, 2-3 (2016), 119–271. https://doi.org/10.1561/1500000032

[2] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[3] Enrico Fregnan, Fernando Petrulio, Linda Di Geronimo, and Alberto Bacchelli. 2022. What happens in my code reviews? An investigation on automatically classifying review changes. *Empirical Software Engineering* 27, 4 (14 Apr 2022), 89. https://doi.org/10.1007/s10664-021-10075-5

[4] Yilin Gao, Sai Arava, Yancheng Li, and James Jr. 2024. Improving the Capabilities of Large Language Model based Marketing Analytics Copilots with Semantic Search and Fine-Tuning. *International Journal on Cybernetics Informatics* 13 (03 2024), 15–31. https://doi.org/10.5121/ijci.2024.130202

[5] Haiyang Liu, Yang Zhang, Vidya Saikrishna, Quanquan Tian, and Kun Zheng. 2024. Prompt Learning for Multi-Label Code Smell Detection: A Promising Approach. arXiv:cs.SE/2402.10398

[6] Stacy Nelson and Johann Schumann. 2004. What makes a code review trustworthy?. In *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the.* IEEE, 10–pp.

[7] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (01 Jun 2018), 1188–1221. https://doi.org/10.1007/s10664-017-9535-z

[8] Zéphyrin Soh, Aiko Yamashita, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2016. Do Code Smells Impact the Effort of Different Maintenance Programming Activities?. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 393–402. https://doi.org/10.1109/SANER.2016.103

[9] Asif Kamal Turzo, Fahim Faysal, Ovi Poddar, Jaydeb Sarker, Anindya Iqbal, and Amiangshu Bosu. 2023. Towards Automated Classification of Code Review Feedback to Support Analytics. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* 1–12. https://doi.org/10.1109/ESEM56168.2023.10304851

[10] Marcelo Vitorino. 2024. *How developers discuss Code Smells during Code Review: A replication.* Master's thesis. Universidade Federal de Campina Grande.