Code Smell Detection using Features from Version History

Ulrike Engeln
Institute for Software Systems, Hamburg University of Technology

1 Introduction

Code smells are indicators for bad quality of source code. In 1999, Fowler and Beck introduce the concept of smells to ease identification of refactoring opportunities in software. They define about 20 structures in code that commonly require re-engineering.

Manual identification of smells is usually time-consuming and costly since deep understanding of the whole software project is necessary to, for example, identify dependencies in the code. Therefore, there exist several attempts of automated code smell detection in literature. Most of them are based upon static properties of the software, e.g., code metrics. However, not all design flaws are of structural nature. There exist three different kinds of code smells. They target coding style, responsibilities, and interdependencies of classes or methods. While smells that target coding style are of structural nature only, smells that describe interdependencies, e.g., Feature Envy, cannot be detected by structural properties. For identification of such smells, Palomba et al. propose using the version history of the code as source of information. Smells targeting responsibilities affect both, code metrics and version history. For example, God Classes usually have many lines of code and appear frequently in the version history.

A well suited approach for the development of a smell detector are machine learning techniques that learn based on features, i.e., measurable properties of the software under investigation, e.g., code metrics. If we apply machine learning techniques to automate smell detection, then we expect classifiers trained on code metrics and information from version history to produce orthogonal results since they focus on different aspects of smells. To improve performance in code smell detection and enable a classifier to detect all three kinds of smells, Barbez et al. introduce a hybrid, ensemble learning-based smell detection technique, which combines classifiers using code metrics and information from version history. We propose a different approach of combining the two sources of information, which, rather than combining existing detectors, directly learns identification of smells from the two sources of information.

Since in their work Palomba et al. do not draw features from version history but apply heuristic rules, one major issue of our machine learning approach is to decide how to express information from the version history by features.

2 Features from Version History

The introduced method of feature extraction from version history builds the core of our work. The general idea is to measure how often files or methods change simultaneously.

We introduce three design parameters, which determine which parts of version history are considered as simultaneous change.

- **Direction** specifies whether prior or future changes are evaluated.
- **Window size** specifies the number of history entries that are considered.
- **Weighting** specifies how changes are weighted as function of their distance.

For the weighting, we define a constant, a linear, and an exponential strategy as follows:

$$w_{\text{const}} = \begin{cases} 1, \text{ if } distance \leq window\_size. \\ 0, \text{ otherwise.} \end{cases}$$

$$w_{\text{lin}} = \begin{cases} 1 - \frac{distance}{window\_size}, \text{ if } distance \leq window\_size. \\ 0, \text{ otherwise.} \end{cases}$$

$$w_{\text{exp}} = \begin{cases} 2^{-distance}, \text{ if } distance \leq window\_size. \\ 0, \text{ otherwise.} \end{cases}$$

Figure 1 illustrates the general idea of measuring simultaneous changes. For each file, we create a vector containing entries for all available files. We identify all commits containing the file and distribute points for all files that appear within the defined window (here: forward oriented of size 3 starting from commit 2) according to the chosen weighting strategy.

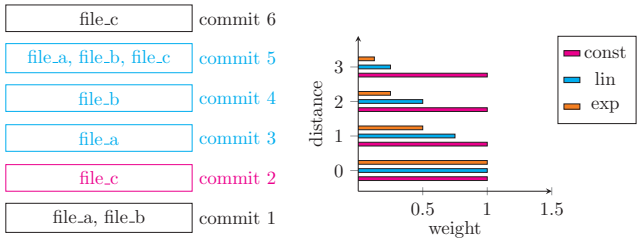

Figure 1: Concept for distributing weights from git history for *file_c*. Relevant commits are colored. Right: different weighting concepts.

We use statistical description of the normalized weight vectors, among others maximum, median and mean, as historical features.

3 Evaluation of Historical Features

For evaluation of the introduced historical features we investigate the following research question.

> **RQ1**: Do the historical features introduced in this work lead to better code smell detection compared to using only code metrics?

We answer the formulated question through three hypotheses, which we test on the code smells God Class and Feature Envy. We select those two smells since they target responsibility and interdependencies, respectively. They belong to the two kinds of smells that we expect to show in version history. Further, for the two smells, labeled data is available in existing work.

With the first hypothesis,

> **RQ1.H1**: There exists information about smells in version history.

we seek to verify whether the designed historical features can capture information about smells from version history at all.

The second hypothesis investigates the impact of the design parameters:

> **RQ1.H2**: For Feature Envy, most information is gained from history if metrics focus on the close past.

We assume that, especially for smells that target interdependencies, the choice of design parameters (see Sec. 2) matters.

The last hypothesis addresses the core of our work. We explore whether historical metrics complement code metrics assuming that:

> **RQ1.H3**: In particular for detection of smells that target interdependencies, information from version history adds value compared to using only code metrics.

For verification of the hypotheses, we perform experiments following the workflow from Figure 2. As ground truth, we apply the available data from Madeyski et al. and Palomba et al. [2, 3]. We have 111 instances of God Class and 125 instances of Feature Envy available. For each smell, we add good instances such that we obtain a data set of $\frac{1}{11}$ smelly instances and $\frac{10}{11}$ good instances. We extract code metrics and historical features of all instances from the data sets of the two smells. Next, we split the available data into test and training part where we ensure that data from a project either appears in the test or the training data to avoid correlation in the data sets. The training data is used to train a random forest of 100 decision trees, which we evaluate by different performance measures. For more reliable evaluation results, we repeat the training process applying 10-fold cross-validation.

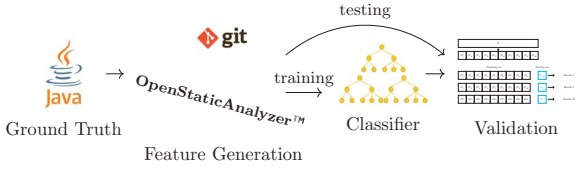

Figure 2: General workflow of the experiments.

4 Results

Evaluation of RQ1.H1 shows that historical features contain information about code smells. For RQ1.H2, surprisingly, we are not able to identify specific design parameters that show best performance. The results of RQ1.H3, finally, are given in Table 1. For God Class, we observe improvement of 0.3% to 1.8% in all performance measures but precision, which decreases by about 0.4%. For Feature Envy, we observe a significant improvement, e.g., F1-score increases from 43.03% to 54.36%.

Table 1: Performances with and without historical features.

	Feature Envy		God Class	
	code metrics	with history	code metrics	with history
Accuracy	90.79%	92.18%	95.95%	96.22%
F1-Score	43.03%	54.36%	74.87%	75.72%
Precision	57.20%	70.20%	77.22%	76.83%
Recall	42.37%	49.94%	73.96%	75.74%
AUC	0.6885	0.7323	0.8651	0.8737

5 Conclusion and Future Work

In our work, we introduce a method to draw historical features that improve smell detection. Results from Section 4 show that historical features allow for better detection of smells targeting interdependencies, e.g., Feature Envy. For God Class, the observed differences are too small to assume an impact.

Future work should focus on four points:

- impact of the design parameters of the features,
- value of historical features for smells targeting responsibilities,
- extension to other smells,
- extension to other programming languages.

A major challenge of any empirical evaluation, and thus for all open points, is the availability of ground truth since there barely exist labeled data of code smells. In our work, we observe that results highly depend on the selected training and test set. Thus, a wider data set could lead to more insights for the first two points. Extension to other smells requires further data sets. For the fourth point, an alternative to training on new data is the application of transfer learning, which we investigate in the master thesis that belongs to this paper [1].

References

[1] Ulrike Engeln. *Transfer learning code smells using version history.* Master's thesis, Hamburg University of Technology, Forthcoming 2023.

[2] L. Madeyski and T. Lewowski. Mlcq: Industry-relevant code smell data set. EASE '20, page 342–347. ACM, 2020.

[3] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. de Lucia. Mining version histories for detecting code smells. *IEEE Trans. Softw. Eng*, 41(5):462–489, 2015.