Preprint of a chapter: Hanna Grodzicka, Arkadiusz Ziobrowski, Zofia Łakomiak, Michał Kawa, and Lech Madeyski, "Code Smell Prediction Employing Machine Learning Meets Emerging Java Language Constructs", in Data-Centric Business and Applications: Towards Software Development (Volume 4) (A. Poniszewska-Maranda, N. Kryvinska, S. Jarzabek, and L. Madeyski, eds.), vol. 40 of book series Lecture Notes on Data Engineering and Communications Technologies, pp. 137–167, Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-030-34706-2_8 [BibTeX]
Draft: http://madeyski.e-informatyka.pl/download/GrodzickaEtAl20LNDECT.pdf

Code smell prediction employing machine learning meets emerging Java language constructs

Hanna Grodzicka, Arkadiusz Ziobrowski, Zofia Łakomiak, Michał Kawa, Lech Madeyski (✉)

Abstract — ***Background:*** Defining code smell is not a trivial task. Their recognition tends to be highly subjective. Nevertheless some code smells detection tools have been proposed. Other recent approaches incline towards machine learning (ML) techniques to overcome disadvantages of using automatic detection tools.
Objectives: We aim to develop a research infrastructure and reproduce the process of code smell prediction proposed by Arcelli Fontana et al. We investigate ML algorithms performance for samples including major modern Java language features. Those such as lambdas can shorten the code causing code smell presence not as obvious to detect and thus pose a challenge to both existing code smell detection tools and ML algorithms.
Method: We extend the study with dataset consisting of 281 Java projects. For driving samples selection we define metrics considering lambdas and method reference, derived using custom JavaParser-based solution. Tagged samples with new constructions are used as an input for the utilized detection techniques.
Results: Detection rules derived from the best performing algorithms like J48 and JRip incorporate newly introduced metrics.
Conclusions: Presence of certain new Java language constructs may hide *Long Method* code smell or indicate a *God Class*. On the other hand, their absence or low number can suggest a *Data Class*

Key words: Code smells detection, replication study, machine learning

Hanna Grodzicka
Faculty of Computer Science and Management
Wroclaw University of Science and Technology
Wroclaw, Poland
ORCiD: 0000-0002-0142-3270
e-mail: 226154@student.pwr.edu.pl

Arkadiusz Ziobrowski
Faculty of Computer Science and Management
Wroclaw University of Science and Technology
Wroclaw, Poland
ORCiD: 0000-0003-3173-2258
e-mail: 229728@student.pwr.edu.pl

Zofia Łakomiak
Faculty of Computer Science and Management
Wroclaw University of Science and Technology
Wroclaw, Poland
ORCiD: 0000-0002-9908-2450
e-mail: 226190@student.pwr.edu.pl

Michał Kawa
Faculty of Computer Science and Management
Wroclaw University of Science and Technology
Wroclaw, Poland
ORCiD: 0000-0001-7035-4734
e-mail: 228007@student.pwr.edu.pl

Lech Madeyski
Faculty of Computer Science and Management
Wroclaw University of Science and Technology
Wroclaw, Poland
ORCiD: 0000-0003-3907-3357
e-mail: lech.madeyski@pwr.edu.pl

1 Introduction

Ever-shifting environment of software development introduces a variety of factors that can affect software quality. Changing requirements, high time pressure and sometimes lack of experience may contribute to creation of code of inferior quality. Code affected by these factors may bring an attentive developer to a conclusion that it "smells". As it is beneficial to be able to grasp the subtleties of such issues, an efficient detection of poor programming habits is a subject of intensive scientific research.
Code smell term was spread by Fowler [3]. He defined smells as something easy to spot (just like real smells) and indicators of a problem since the smells are not inherently bad. Since then, many publications concerning the subject of code smells have been released. A considerable amount of them refer to tools and algorithms for their detection in software projects.

We performed preliminary review of literature on code smell prediction and existing open access datasets of code smells.
One recent publication in this field was published by Arcelli Fontana et al. [1] in the article *Comparing and experimenting machine learning techniques for code smell detection*. In the study attention was paid to four code smells: *Data Class*, *God Class*, *Feature Envy* and *Long Method*. They conduct an extensive comparison of 16 different machine learning algorithms to aid the detection of code smells. They discovered that the best performance was obtained by J48 and Random Forest models. Detection of code smells employing these techniques can provide a very high accuracy – over 96%. There are some premises for imprecision of their research though – these issues will be referred to in the upcoming sections of this paper.

Palomba et al. [10] used the `LANDFILL` dataset presented in another publication (Palomba et al. [11]) and proposed Historical Information for Smell deTection (HIST) approach exploiting change history information to detect instances of five different code smells (*Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob*, and *Feature Envy*). The results indicate that the precision of HIST is between 72 and 86 percent, while its recall is between 58 and 100 percent.
Many papers have introduced a number of tools for detection of code smells. For instance, Arcelli Fontana et al. [1] used iPlasma, PMD, Fluid Tool, Marinescu and Antipattern Scanner. In other publication, Arcelli Fontana et al. [2] additionally used JDeodorant and StenchBlossom to compare tools for code smells detection. Another way to discover presence of smells is *Textual Analysis*, which was described by Palomba [9].
Palomba et al. [11] contributed the dataset containing 243 instances of five smell types from 20 open source projects manually verified by two MSc students. They also presented `LANDFILL`, a web-based platform aimed to promote the collection and to share code smell datasets.
There is yet another tool for detection of code smells. Designite was created and developed by Sharma [12]. It focuses on C# code, but the author provided a version for Java as well. In our research we have used it to verify its detection capabilities on Qualitas Corpus datasets (Tempero et al. [13]) and within new Java structures. This will be described in one of the following chapters.

Tempero et al. [13] created a web page containing open source Java projects. Arcelli Fontana et al. [1] used 74 projects from this set in order to conduct their research. The biggest drawback of this collection is that the newest version comes from 1 September 2013.

The paper consists of six sections, that are organized in the following manner: in Section 2 we describe the reference work and analyze the approach of Arcelli Fontana et al. [1]. Then we discuss the reproduction of Arcelli Fontana et al. [1] experiment. In Section 3 we go through the empirical study conducted by our team. In Section 4 we present the obtained results which are then augmented by an analysis in the Section 5. We conclude the research and indicate future directions in Section 6.

2 The reference work

Instead of using automated tools for code smell prediction Arcelli Fontana et al. [1] decided on machine learning approach that we aim to reproduce.

2.1 Collecting the datasets

We began the reproduction of the Arcelli Fontana et al. [1] experiment with obtaining used datasets. Arcelli Fontana et al. [1] sourced datasets from Qualitas Corpus (Tempero et al. [13]), a curated collection of various software systems. As of the date of writing this paper, the current release of Qualitas Corpus is 20130901. Datasets' last modification dates were retrieved. The results are presented below.

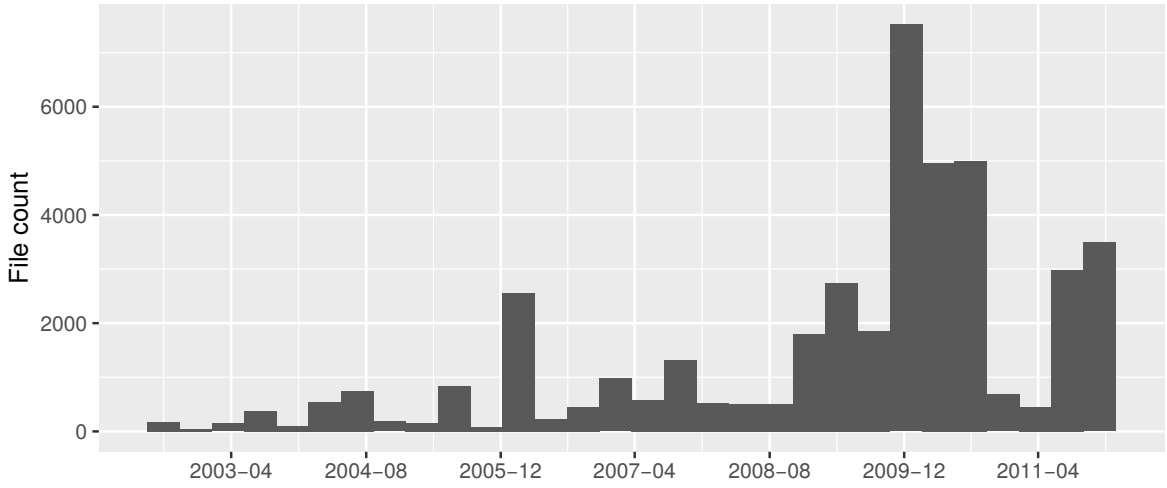


Fig. 1: Histogram presenting last modification dates for the Qualitas Corpus dataset

Arcelli Fontana et al. [1] used Qualitas Corpus distribution from 2012-01-04 which contains source code even from 2002.
State of the art software systems could have completely diverged from design concepts, that were widespread during the creation of such systems. Therefore in the following sections of this paper, we attempted to reason with the usage of such systems as giving reliable results.
We had to filter out Qualitas Corpus systems, which did not appear in the Arcelli Fontana's experiment and augment the source code collection with missing ones. To obtain these results, a set of bash scripts has been written. Individual scripts attempt to tackle problems such as filtering out the incorrect datasets and generating a list of missing systems. We have found that six systems were missing from the newest distribution of the Qualitas Corpus: *freecol*, *jmeter*, *jpf*, *junit*, *lucene*, *weka*.
Following systems had differing versions: *freecol*, *jmeter*, *junit*, *lucene* and *weka*, but *jpf* was not present in the Qualitas Corpus newest distribution, despite being attached there in the version, that was used by Arcelli Fontana et al. [1].
The missing systems were manually retrieved from available repositories hosted on Sourceforge or system manufacturer's website. We were successful in obtaining every single one of the systems used in Arcelli Fontana's experiment in the proper version. The archive containing mentioned datasets is available online along

Table 1: Summary of Qualitas Corpus last modification dates

	Date
Min.	2002-09-02 17:33:14
1st Qu.	2008-08-19 02:07:16
Median	2010-01-03 13:30:36
Mean	2009-05-19 17:18:00
3rd Qu.	2010-08-04 01:26:36
Max.	2011-12-15 21:26:14

with the summary of missing systems (Table 1) prior to their manual supplementation. Additionally, a version of the archive without the test source code has been prepared and shared online likewise[1].

2.2 Tooling approach

Arcelli Fontana et al. [1] introduced some inconsistencies in their paper. One of the most vital challenges for the reproduction of the results were choosing the proper version of tools. Arcelli Fontana et al. [1] did not specify advisors nor *weka* versions. Therefore it was needed for us to come up with the most approximate approach.
We decided to choose R language to build the reproduction infrastructure with usage of `caret` and `RWeka` packages, since they are ones of the most popular tooling[2] choices. As for advisors, we used PMD which date of release was closest to the expected date of creation of Arcelli Fontana et al. [1].
Another downside is the lack of parameter specification, that were obtained by the means of hyperparameter optimization, for the models employed by Arcelli Fontana et al. [1]. Therefore it is not trivial to put the obtained results in the proper context.

2.3 `caret` vs `RWeka` approach

Both `caret` and `RWeka` [6] R packages have been used during Arcelli Fontana et al. [1] experiment reproduction. Following subsections describe approaches in use of each of the packages.

2.3.1 `caret`

Package `caret` (short for *Classification And Regression Training*) is a set of functions that attempt to streamline the process for creating predictive models. The package significantly simplifies data-splitting, pre-processing, feature selection, model tuning using resampling, variable importance estimation and others.
In order to reproduce Arcelli Fontana et al. [1] experiment, the `caret` package has been used. Unfortunately, not all of the models used in the experiment are available in the package, especially `caret` does not include models with the boosting technique AdaBoostM1.
Nevertheless, `caret` has been used because of an ease of cross-validation and grid search (Hsu et al. [7]) parameter estimation technique usage, which has been employed in Arcelli Fontana et al. [1] experiment. Parameters obtained by means of grid search in Arcelli Fontana et al. [1] have not been published in the paper. Table 2 shows models used from the `caret` package. The rest of algorithms have not been run in `caret` due to limitations of their availability in the package. Parameters presented in the `Tuning parameters` column in Table 2 have been computed with the default `caret` grid search settings. Tune grid used for grid search is

[1] Datasets – *sources.tar.gz* for full datasets

[2] `https://www.kdnuggets.com/2015/06/top-20-r-machine-learning-packages.html`, access: 2019-04-09

derived exclusively for each of the algorithms, based on the model tuning parameters provided by the library, which includes the model.
The granularity in the tuning parameter grid is based on `tuneLength` parameter in the `train` function, which defaults to the division of parameter value space equally on three values. One can specify it manually, but we decided to omit the manual specification of `tuneLength` parameter for sake of performance. Moreover choosing the best tuning parameter is driven by the `best` strategy of `train` function, that selects parameters associated with the best performance in terms of area under the ROC (Receiver Operating Characteristics) curve.

Table 2: Models from `caret`

#	Model	Method Value	Type	Library	Tuning Parameters
1	C4.5-like Trees	J48	Classification	RWeka	C=0.01, M=3
2	Naive Bayes	naive_bayes	Classification	naivebayes	laplace=0, usekernel=FALSE, adjust=1
3	Random Forest	rf	Classification, Regression	randomForest	mtry=1011
4	Rule-Based Classifier	JRip	Classification	RWeka	NumOpt=3, NumFolds=3, MinWeights=2

2.3.2 `RWeka`

Package `RWeka` is an interface to machine learning library `Weka`, a collection of machine learning algorithms for data mining tasks written in Java, containing tools for data pre-processing, classification, regression, clustering, association rules, and visualization (`RWeka` [14], Hall et al. [5]).
In order to reproduce Arcelli Fontana et al. [1] experiment, the `RWeka` package has been used as well. The package allows to use the exact algorithms Arcelli Fontana et al. [1] used, including their boosted versions. Downside of using the package directly is a fact, that it does not provide a simple method for `grid search`, which has been applied as a parameter optimization technique in the experiment we have reproduced.
Table 3 includes models used from `RWeka`. The rest of algorithms have not been run with `RWeka` due to their unavailability in the package.

Table 3: Models from `RWeka`

#	Algorithm name	Default parameters
1	B-J48 Pruned	iterations=10, C=0.25, M=2
2	B-J48 Unpruned	iterations=10, C=0.25, M=2
3	B-J48 Reduced Error Pruning	iterations=10, C=0.25, M=2
4	B-JRIP	iterations=10, F=3, N=2.0, O=2
5	B-SMO RBF Kernel	iterations=10, C=250007, G=0.01
6	B-SMO Poly Kernel	iterations=10, C=250007, E=1.0
7	J48 Pruned	C=0.25, M=2
8	J48 Unpruned	C=0.25, M=2
9	J48 Reduced Error Pruning	C=0.25, M=2
10	JRIP	F=3, N=2.0, O=2
11	SMO RBF Kernel	C=250007, G=0.01
12	SMO Poly Kernel	C=250007, E=1.0

2.4 Reproduction strategy overview

We attempted to reproduce the results with the approach adequate to Arcelli Fontana et al. [1]. Manually evaluated datasets provided by Arcelli Fontana et al. [1], alongside with available models from `caret` and `RWeka` packages were used.
Firstly, the hyperparameter optimization was done by means of grid search. Method `trainControl` from `caret` employs this method by specifying its `search` parameter as `grid`.
Models were trained using 10-fold cross-validation – for every model there were ten iterations of 10-fold cross-validation, that were later averaged.
ARFF files contain missing values for various metrics – they are marked with explanation mark within a file.

Table 4: Missing values overview in Arcelli Fontana et al. [1] datasets

File	No. of missing values
data-class.arff	75
feature-envy.arff	92
god-class.arff	76
long-method.arff	92

Unfortunately, we do not know the strategy to deal with missing values in our reproduction, since Arcelli Fontana et al. [1] does not specify any particular method. It could be therefore crucial to recognize a proper technique to supplement missing values in order to fully reproduce the original results. Based on the trained model, a confusion matrix was computed, thanks to which an accuracy and F-measure were obtained and compared with the results from Arcelli Fontana et al. [1]. The results were augmented with standard deviation and area under ROC as well.

2.5 Classifier comparison

For reproduction we used ARFF files provided by Arcelli Fontana et al. [1] as an input.
Altogether, 18 algorithms were tested: 16 from RWeka[3] and 6 from Caret[4]. The common classifiers for both libraries are JRip and J48 Unpruned.
Results marked green are those that came out better compared to Arcelli Fontana et al. [1].
In RWeka, for the accuracy the results differ no more than 2% except SMO RBF Kernel (**bold** font) that is unquestionably worse, by 6.53%. Standard deviation for accuracy is always lower for our results – mostly rounds up to zero. In original work it was between 1 and 3 for the given classifiers.
F-measure has similar outcome to accuracy. Our results tend to be nearly the same. The most notable difference is 1.16% excluding SMO RBF Kernel that has lower score by 4.81%. Similarly to the accuracy, F measure standard deviation is rounded up to 0 in nearly every case.
Area under the ROC has very similar results to the original ones. In few cases it is even better. Its standard deviation is approximately two times lower than in Arcelli Fontana et al. [1] research.
The best algorithms from Table 5 are B-J48 and B-JRip, what agrees with Arcelli Fontana's results. All the SMO-based classifiers came out worse in reproduction.
As for `caret` library, J48 Unpruned, JRip and Random Forest resemble scores achieved by Arcelli Fontana et al. [1]. SVM-based classifiers showed considerable deterioration. Whereas Naive Bayes seems unreliable, it is giving us similarly bad results every time (Table 6).
In conclusion, the reproduction has given consistent results of cross validation for classifiers based on J48, JRip and Random Bayes. SMO and SVM went worse than expected. Naive Bayes gave clearly bad results. The results can be affected by the choice of the libraries and following default behaviour of its components.

[3] `https://cran.r-project.org/web/packages/RWeka/index.html`, access: 2019-04-09

[4] `http://topepo.github.io/caret/index.html`, access: 2019-04-09

Table 5: RWeka results for *Long Method* (grey) compared with Arcelli Fontana's (white)

#	Classifier	Accuracy	Std dev.	F measure	Std dev.	AUROC	Std dev.
1	B-J48 Pruned	99.20%	0.00	99.40%	0.00	0.9913	0.0083
2	B-J48 Pruned	99.43%	1.36	99.49%	1.00	0.9969	0.0127
3	B-J48 Unpruned	99.41%	0.00	99.56%	0.00	0.9962	0.0042
4	B-J48 Unpruned	99.20%	1.18	99.63%	0.99	0.9969	0.0126
5	B-J48 Reduced Error Pruning	98.99%	0.00	99.24%	0.00	0.9967	0.0027
6	B-J48 Reduced Error Pruning	99.19%	1.31	99.39%	0.87	0.9967	0.0100
7	B-JRip	99.05%	0.00	99.29%	0.00	0.9890	0.0058
8	B-JRip	99.03%	1.26	99.50%	0.94	0.9937	0.0144
9	B-Random Forest	98.99%	0.00	99.25%	0.00	0.9996	2e-04
10	B-Random Forest	99.23%	1.17	99.57%	1.91	0.9998	0.0006
11	B-Naive Bayes	95.52%	0.00	96.57%	0.00	0.9780	0.0035
12	B-Naive Bayes	97.86%	2.37	98.35%	2.02	0.9950	0.0084
13	B-SMO RBF Kernel	95.49%	0.01	96.56%	0.01	0.9900	0.0022
14	B-SMO RBF Kernel	97.00%	2.49	97.75%	2.38	0.9930	0.0116
15	B-SMO Poly Kernel	97.42%	0.00	98.07%	0.00	0.9704	0.0047
16	B-SMO Poly Kernel	98.67%	1.76	99.00%	2.17	0.9852	0.0208
17	J48 Pruned	98.94%	0.00	99.21%	0.00	0.9938	0.0034
18	J48 Pruned	99.10%	1.38	99.32%	1.04	0.9930	0.0151
19	J48 Unpruned	98.92%	0.00	99.19%	0.00	0.9933	0.0035
20	J48 Unpruned	99.05%	1.51	99.28%	1.54	0.9925	0.0168
21	J48 Reduced Error Pruning	98.07%	0.01	98.55%	0.00	0.9887	0.0068
22	J48 Reduced Error Pruning	98.40%	2.02	98.80%	1.13	0.9868	0.0222
23	JRip	98.89%	0.00	99.17%	0.00	0.9880	0.0047
24	JRip	99.02%	1.62	99.26%	1.79	0.9884	0.0181
25	Random Forest	99.23%	0.00	99.42%	0.00	0.9996	1e-04
26	Random Forest	99.18%	1.20	99.54%	1.62	0.9998	0.0011
27	Naive Bayes	93.35%	0.00	94.80%	0.00	0.9649	0.0027
28	Naive Bayes	96.24%	2.39	97.09%	1.72	0.9921	0.0086
29	SMO RBF Kernel	90.44%	0.00	93.29%	0.00	0.8583	0.0066
30	SMO RBF Kernel	97.57%	2.02	98.17%	1.61	0.9732	0.0235
31	SMO Poly Kernel	97.06%	0.00	97.81%	0.00	0.9653	0.0060
32	SMO Poly Kernel	98.67%	1.76	99.00%	1.69	0.9852	0.0208

Comparison tables for other code smells can by found in Appendix (Section 6).

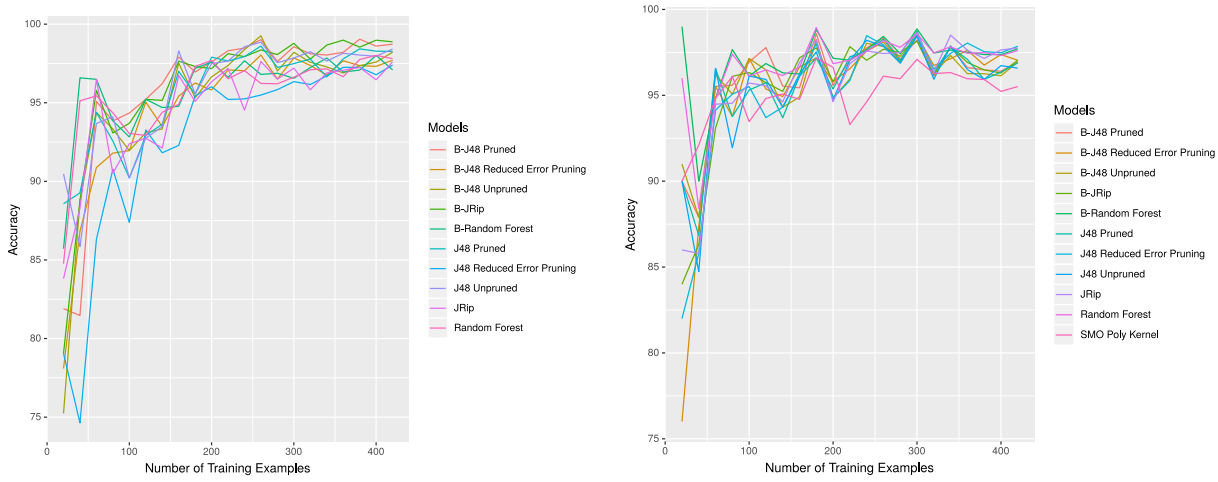
2.6 Learning curves

Learning curves present the behaviour of models' accuracy with incremental change in number of training examples. Figures 2a, 2b, 3a and 3b present results obtained with use of the `RWeka` package.

Table 6: Caret results for *Long Method* (grey) compared with Arcelli Fontana's (white)

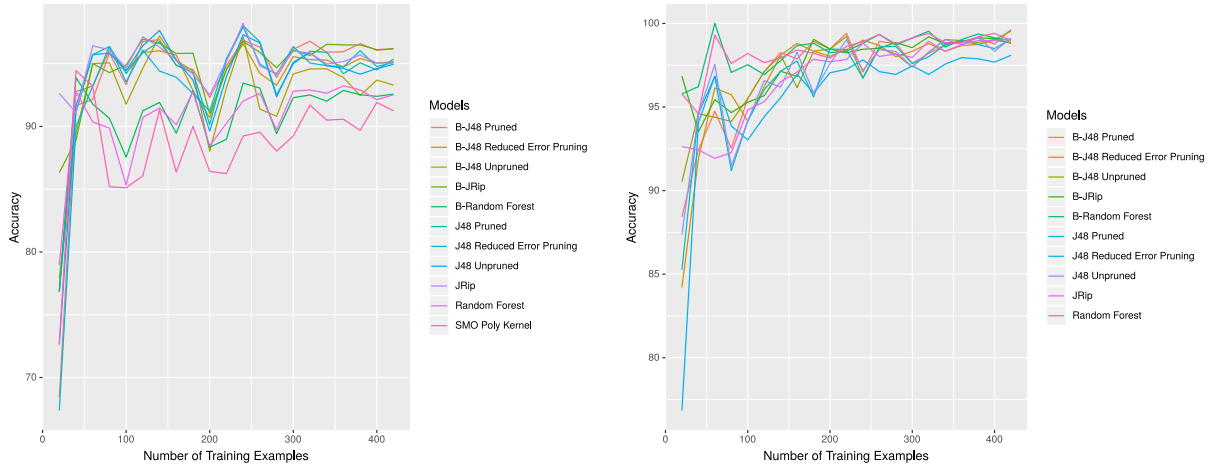
#	Classifier	Accuracy	Std dev.	F measure	Std dev.	AUROC	Std dev.
1	SVM C-SVC Linear Kernel	**89.70%**	0.02	**92.62%**	0.01	**0.9225**	0.0228
2	LibSVM C-SVC Linear Kernel	97.31%	2.22	97.97%	1.88	0.9978	0.0044
3	SVM C-SVC Radial Kernel	**71.16%**	0.01	**81.54%**	0.01	**0.7562**	0.0255
4	LibSVM C-SVC Radial Kernel	97.43%	2.09	98.05%	1.32	0.9972	0.0045
5	J48 Unpruned	98.07%	0.01	98.56%	0.00	0.9842	0.0067
6	J48 Unpruned	99.05%	1.51	99.28%	1.54	0.9925	0.0168
7	Random Forest	98.90	0.00	99.17%	0.00	0.9997	0.0003
8	Random Forest	99.18%	1.20	99.54%	1.62	0.9998	0.0011
9	Naive Bayes	**32.65%**	0.00	NA	NA	**0.7450**	0.0141
10	Naive Bayes	96.24%	2.39	97.09%	1.72	0.9921	0.0086
11	JRip	98.70	0.00	99.03%	0.00	0.9861	0.0014
12	JRip	99.02%	1.62	99.26%	1.79	0.9884	0.0181

Method used for generating the learning curves was equivalent to the one used by Arcelli Fontana et al. [1]. Starting from the dataset of size 20, ten iterations of 10-fold cross validation has been made with constant increment in the number of training examples after each iteration. Accuracy of a model for a given dataset was computed as a mean from mentioned ten iterations.
Obtained learning curves are non-monotonic, although the trend seems to be similar to the results of Arcelli Fontana et al. [1]. Additionally, there appears to be more significant jitter and deviation from the Arcelli Fontana's results. A deciding factor for such learning curves behaviour might be the tooling approach or the dataset partitioning method, that was used to produce incremental datasets.



(a) *Data Class* (b) *God Class*

Fig. 2: Learning curves for *Data Class* and *God Class* code smells



(a) *Feature Envy* (b) *Long Method*

Fig. 3: Learning curves for *Feature Envy* and *Long Method* code smells

2.7 Extracted Rules

For sake of completeness we show the extracted rules from decision-tree based model J48 and from rule-based JRip.
All appearing differences in extracted rules might be due to different grid search approach, since `caret` default grid search explores only a limited space of parameters and no grid search has been employed for `RWeka`.
Metrics' abbreviations can be found in Section 3.4.

2.7.1 Long Method

For *Long Method*, J48 Pruned generates a decision tree that can be expressed as the following set of rules:

$$LOC > 79 \text{ and } CYCLO > 9 \tag{1}$$

The detection rule declares a method as code smell, when its size is large and it is complex. Obtained results for J48 are identical with Arcelli Fontana et al. [1].
As for JRip, the computed rule is as follows:

$$LOC \geq 91 \text{ and } CYCLO \geq 10 \tag{2}$$

It differs from the Arcelli Fontana's results by a small margin – we can observe increment by 10 for LOC metric and by 2 for $CYCLO$ metric.

2.7.2 Data Class

For *Data Class* code smell, detection rule derived from the J48 classifier is yet again identical with the results obtained by Arcelli Fontana. The following Boolean logic expression describes the code smell in terms of J48 decision tree:

$$NOAM > 2 \text{ and } WMCNAMM \leq 21 \text{ and } NIM \leq 30 \tag{3}$$

JRip classifier produces the following expression:

$$\begin{aligned} &(WOC \leq 0.352941) \text{ and } (NOAM \geq 3) \text{ and } (RFC \leq 39) \\ &\quad \text{or } (AMW \leq 1.181818) \text{ and } (NOM \geq 8) \text{ and } (NOAM \geq 4) \\ &\quad \text{or } (NOM \leq 27) \text{ and } (NOAM \geq 4) \end{aligned} \tag{4}$$

The first operand of logical disjunction is nearly the same as Arcelli Fontana's, but unfortunately the rest of logical expression diverges. Obtained results introduce NOM metric and omit $CFNAMM$ and $NOPVA$ metrics.

2.7.3 God Class

Results obtained for *God Class* code smell are convergent for both J48 and JRip. The detection rule of J48 for *God Class* is:

$$WMCNAMM > 47 \tag{5}$$

and for JRip:

$$WMCNAMM \geq 48 \tag{6}$$

Both rules are the same. They are coincident with Arcelli Fontana's results for this code smell.

2.7.4 Feature Envy

Detection rules for *Feature Envy* code smell appear to be the most divergent from the results of Arcelli Fontana et al. [1]. J48 Pruned produces the following decision tree:

$$ATFD > 4 \text{ and } NMO > 8 \text{ and } FDP > 5 \tag{7}$$

This expression introduces FDP and NMO as the metrics that constitute whether a sample source code is influenced by *Feature Envy* code smell. FDP metric describe number of foreign data providers and NMO shows the number of overridden methods in a class. Number of overridden methods provides that class uses data in differences ways. So it could use data from other class than from its own as well. However, Arcelli Fontana's J48 decision tree rule had made use of LAA and NOA metrics.
Same results has been observed for the detection rule extracted from JRip:

$$(ATFD \geq 9) \text{ or } (ATFD \geq 3 \text{ and } LAA \leq 0.) \text{ and } LOC \geq 20) \\ \text{or } (FDP \geq 3 \text{ and } LAA \leq 0.578947) \tag{8}$$

First operand of the disjunction is identical with Arcelli Fontana's, however the rest of operands differ.
Probable reason of differences in extracted rules is that we used grip search after `RWeka` computed. Additionally, some rules was extracted by `caret` functions. Arcelli Fontana et al. [1] used only RWeka.

3 Empirical study definition

Arcelli Fontana et al. [1] carried out research on obsolete dataset. We investigated that only fifty are still supported (as of 2019-06-10).

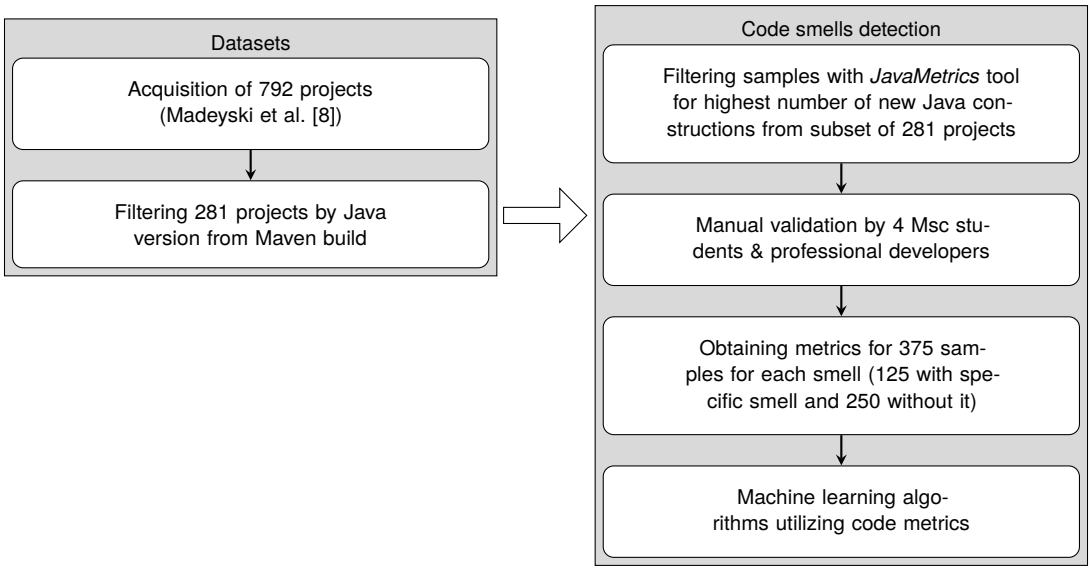


Fig. 4: Steps taken during our research

3.1 Objective

The aim is to take similar steps to Arcelli Fontana et al. [1] but for more recent and still active projects involving new major Java constructs. We want to find out whether proposed algorithms cope with different dataset and how they perform with new language features. Are they still as effective?

3.2 Extending code smell detection with Designite

In our research we decided to compare results from a new code smell detection tool (advisor) on the same datasets as Arcelli Fontana et al. [1], consisting of 74 system from Qualitas Corpus (Tempero et al. [13]). *Long Method* code smell was exclusively explored, since it is the only one that complies with the newly introduced advisor – Designite for Java Sharma [12].
Our results have been made available online[5] for each of the datasets used by Arcelli Fontana et al. [1].
PMD advisor has detected the highest number of smells (see Figure 5) and Designite detections coincide the most with Arcelli Fontana's PMD (Figure 6). One of the factors affecting such result is that PMD and Designite detected the highest number of smells overall. On the other hand, there are notable differences among PMD and Designite comparison and comparisons between following pairs: iPlasma and Designite, Marinescu and Designite. AND/OR ratio for Designite and PMD suggests the closest match between this pair of advisors. However, due to large number of detections for those, the ratio may not be the best accuracy indicator.

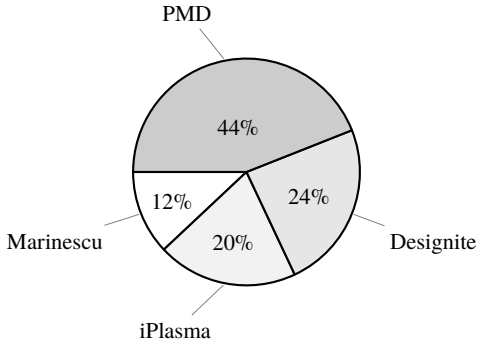


Fig. 5: Ratio of detected smells – PMD, Marinescu, iPlasma, Designite

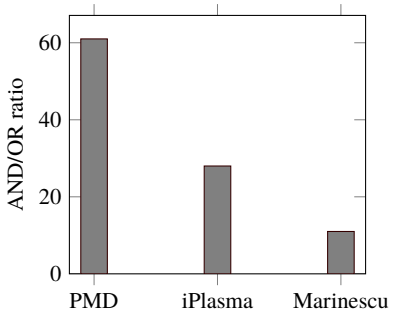


Fig. 6: AND/OR ratio of detected smells – Designite compared with other advisors

3.3 Introducing datasets containing projects with newer Java constructions

Despite the fact that the Qualitas Corpus dataset (Tempero et al. [13]) once had been a proper dataset containing Java projects, nowadays it shows several weaknesses such as:

- it is not updated (the current release is 20130901),
- due to lack of updates it contains projects with very old source codes (Table 1),
- old source codes mean that new Java constructions cannot be observed nor examined.

For a solution to these problems, we decided to introduce a new dataset containing 792 projects Madeyski et al. [8]. The main aim of creating the dataset is to study the impact of newer Java constructions on code smells detection. We concluded, that the current breakthrough Java version is the Java 8. Java 8 has been chosen

[5] http://madeyski.e-informatyka.pl/download/GrodzickaEtAl19DataSet.zip

as the determinant of the freshness of the projects, due to the multitude of new constructions it introduces and its long-term support (LTS) status.

3.3.1 Information about the new dataset

All of the projects in the base dataset Madeyski et al. [8] are open source and available on GitHub. Not all of them are plain Java projects and, additionally, not necessarily use at least Java 8. For this reason the base dataset had to go through filtering.
The R package `reproducer` [8] contains basic information to obtain the projects in their specific versions from GitHub and had been used while collecting them. Further description can be found in the Appendix [4]. The dataset contains actively developed projects (see Table 7, Figure 7). A substantial difference can be observed comparing the datasets' summary with the Qualitas Corpus datasets' summary (see Table1).

Table 7: Summary of last modification dates of 792 projects from the new dataset

	Date
Min.	2002-02-28 13:58:11
1st Qu.	2016-06-18 19:45:16
Median	2018-03-06 08:29:50
Mean	2017-03-21 00:46:19
3rd Qu.	2018-12-04 15:49:56
Max.	2019-04-16 22:55:47

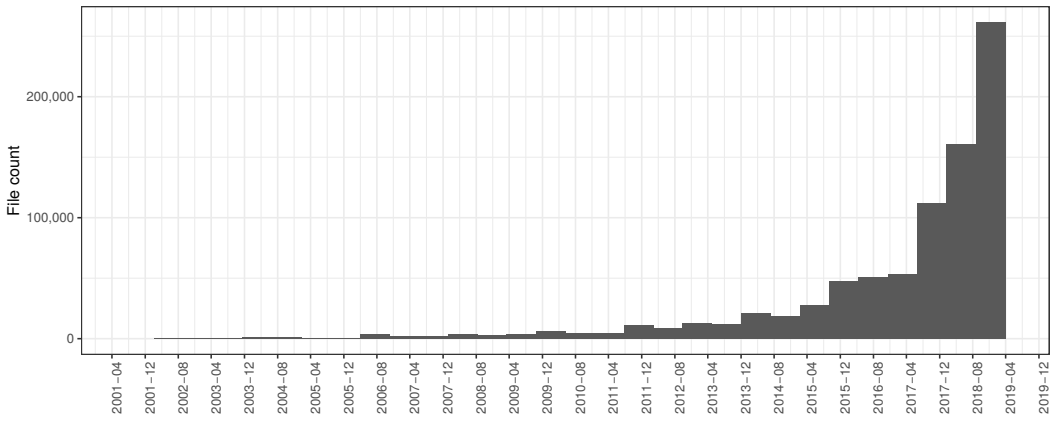


Fig. 7: Overview of last modification dates of 792 projects from the new dataset

3.3.2 Filtering Java Maven projects from the base dataset

In order to ensure that projects comprise Java, we decided to filter them upon Maven build tool. Using Maven build information allowed Java version retrieval aside. Although the base dataset contains constantly updated projects, we were particularly interested in ones with the Java version equal to or higher than the Java 8 version.

3.3.3 Checking Java version in the Maven build tool

The first step to obtain new constructions is filtering projects' build tools by Java version used in builds. Figure 8, Table 8 show distribution of build tools among projects.
There are 439 plain Maven projects, others combine different or multiple build tools. Projects marked as *other or lack of* indicate use of other build scripts and tools than *pom.xml* for Maven, *build.gradle* for Gradle or *build.xml* for Ant.
Additionally, *build.properties* and *.travis.yml* files have been searched for Java versions.

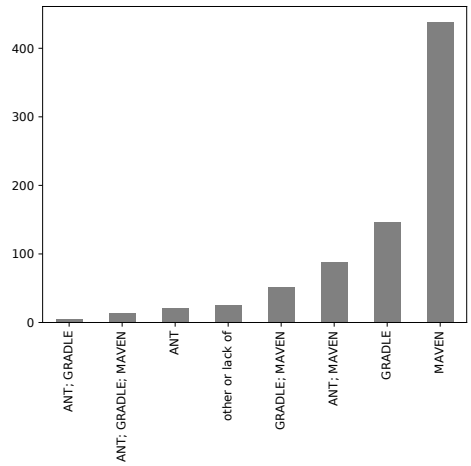


Fig. 8: Projects' build tools

Table 8: Projects' build tools

Build tool(s)	Number of projects
Maven	439
Gradle	147
Ant & Maven	88
Gradle & Maven	52
Ant	21
Ant & Gradle & Maven	14
Ant & Gradle	5
other or lack of	26

The Maven build tool, as the most frequently used, has been selected to retrieve information about the Java version from Maven-based projects. Maven configuration file (`pom.xml`) has been searched for Java version used during the projects' builds. Extracting the version from Maven build script was not always straightforward because of multiple ways of encoding it. Moreover, non-Maven projects or these with unconventionally specified build were omitted and Java version for them was marked as *unknown* (Figure 9, Table 9).

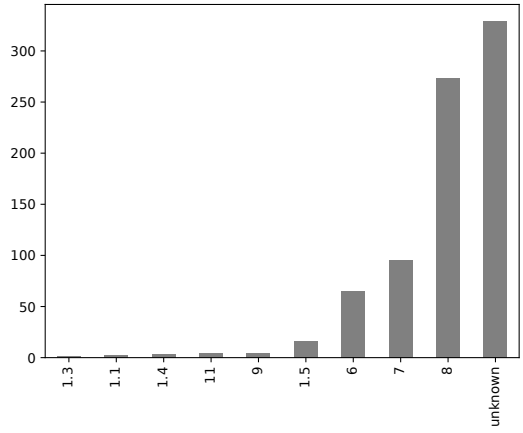


Fig. 9: Projects' Java versions

Table 9: Projects' Java versions

Java version	Number of projects
8	273
7	95
6	65
1.5	16
11	4
9	4
1.4	3
1.1	2
1.3	1
unknown	329

After filtering the 792 projects for at least Java 8 version, the following results were obtained (Table 12, Figure 12).

As for summary of original datasets' subset, following information can be observed (Table 10, Figure 10).

Table 10: Summary of last modification dates of 281 projects from the new dataset

	Date
Min.	2005-02-03 22:37:19
1st Qu.	2017-08-28 17:23:51
Median	2018-08-02 11:18:04
Mean	2017-10-28 07:30:51
3rd Qu.	2019-01-03 05:27:38
Max.	2019-04-16 18:01:42

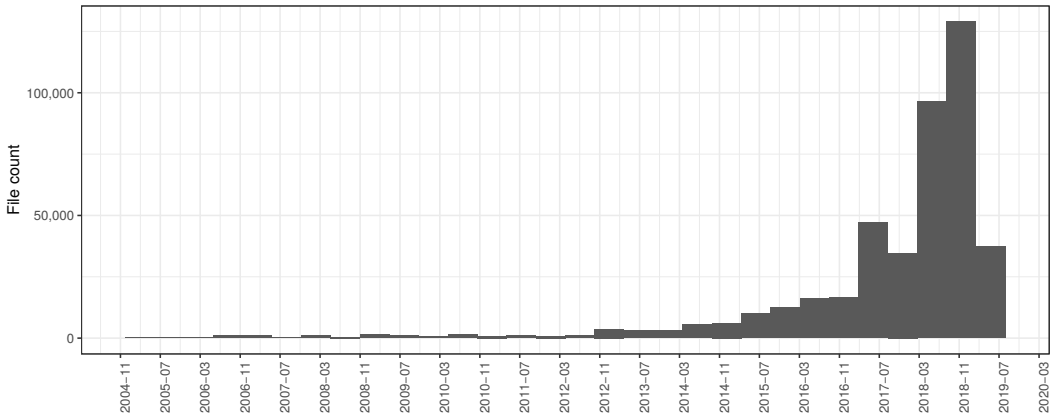


Fig. 10: Overview of last modification dates of 281 projects from the new dataset

Appendix [4] contains details required to obtain the filtered projects in their specific versions from GitHub.

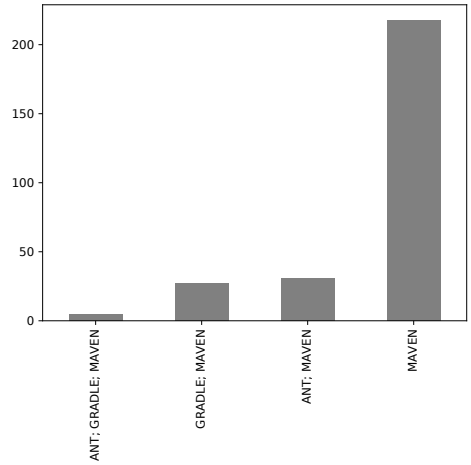


Fig. 11: Filtered projects' build tools

Table 11: Filtered projects' build tools

Build tool(s)	**Number of projects**
Maven	218
Ant & Maven	31
Gradle & Maven	27
Ant & Gradle & Maven	5

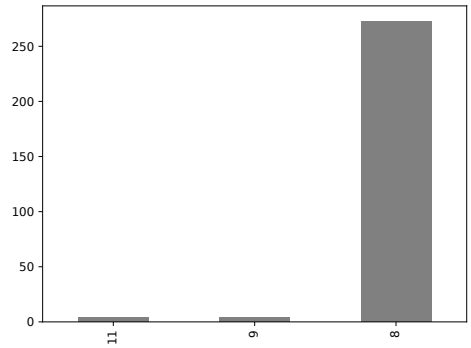


Fig. 12: Filtered projects' Java versions

Table 12: Filtered projects' Java versions

Java version	Number of projects
8	273
11	4
9	4

3.4 Filtering and retrieving information about the actual use of new Java constructions using the JavaParser

JavaMetrics is our custom solution that employs JavaParser[6] to automate metric derivation. It exposes the following metrics that can be run against the provided project:

- Access to Foreign Data ($ATFD$)
- Cyclomatic Complexity ($CYCLO$)
- Foreign Data Providers (FDP)
- Locality of Attribute Access (LAA)
- **Lambda Density (**LD**)**
- Lines of Code (LOC)
- **Method Reference Density (**MRD**)**
- Number of Inherited Methods (NIM)
- Number of Accessor Methods ($NOAM$)
- **Number of Lambdas (**NOL**)**
- Number of Methods (NOM)
- Number of Mutator Methods ($NOMM$)
- **Number of Method Reference (**$NOMR$**)**
- Number of Public Attributes ($NOPA$)
- Number of Private Attributes ($NOPV$)
- Weighted Method Count (WMC)
- Weighted Method Count of Not Accessor or Mutator Methods ($WMCNAMM$)
- Weight of Class (WOC)

Custom metrics are written in **bold**. Both LD and MRD refer to constructions (lambda and method reference) density in class, for instance $LD = NOL/LOC * 100$ (result in percent). NOL and $NOMR$ are implemented for methods separately to use them for *Long Method* detection.
JavaMetrics utilizes visiting the Abstract Syntax Tree (AST) nodes in deriving the metrics. JavaParser allows to traverse the AST with callback invocation only for certain types of nodes. Thanks to robust JavaParser API, it is possible to detect Java language constructs and process them accordingly.
The tool is flexible enough to implement other custom metrics without changing the existing code structure (open-closed principle). A CSV file is generated as the result of the *JavaMetrics* run. The output consists of parsed unit details such as package, class, method signature and additional information whether methods and fields are final or static. Moreover it provides the selected metrics values as the succeeding columns in the output.
Unfortunately, even though the metrics derivation works as expected, the ergonomics of the JavaParser may pose a challenge to completely automate metric derivation process. Missing part involves the resolution of

[6] https://javaparser.org, access: 2019-06-10

classes. For example the NIM metric expects information about the superclass that is most likely in the separate file. JavaParser tool has incorporated a second tool called JavaSymbolSolver as a part of the library. It allows to resolve classes by means of the `import` clauses and referenced files. Both the directory containing the packages under the resolution and external dependencies in the form of JAR files shall be provided.
In order for *JavaMetrics* to work seamlessly, we would have to parse the provided directory to find the directories enclosing the packages and parse the build automation tool files (such as *pom.xml* for Maven) to identify the external dependencies. Those external dependencies would have to be downloaded and appropriately registered within the *JavaMetrics*. This issue is the matter of the future directions of work.

3.5 Manual tagging

Our tool named *JavaMetrics* was used for initial filtering of samples from dataset (Madeyski et al. [8]) with the highest number of lambdas and method references. *JavaMetrics* let us conduct in-depth analysis of the selected samples by bringing metrics.
Code smell recognition was performed mostly by us, 4th year Software Engineering students and junior developers with approximately one year of professional experience. We were validating selected samples till reaching 125 *Long Methods*. Our judgement was partly based on metrics from *JavaMetrics* that gave us overview for each sample.
Tagged samples have been replenished with some tagged by professional developers (Table 13) mentioned in Acknowledgement 6. Due to similarities in their specification, we have treated *Blob* as *God Class* for the purpose of our research.

Table 13: Tagged samples from professional developers

Code smell	Minor or higher severity	Nonsevere
Long Method	0	3
Data Class	5	26
Blob	15	16

Eventually 125 samples were selected for *Long Method*, *Data Class* and *God Class* along with complementary 250 samples without those smells. All of professionally tagged samples were used. Such datasets (with resulting metrics) were used as an input for the machine learning algorithms.

4 Results

Results of manual tagging consist of 375 samples – 125 samples with specific code smell and 250 without it. Each 375 samples along with their metrics were used as an input to build code smell prediction rules employing machine learning algorithms.

4.1 Classifier performance

Table 14: `caret` results for *Long Method*

#	Classifier	Accuracy	Std dev.	F measure	Std dev.	AUROC	Std dev.
1	J48	96.1%	NA	97.08%	NA	0.955	NA
2	rf	96.81%	NA	97.63%	NA	0.9963	NA
3	naive_bayes	94.68%	NA	96%	NA	0.9778	NA
4	JRip	96.81%	NA	97.61%	NA	0.9632	NA

Table 15: `caret` results for *God Class*

#	Classifier	Accuracy	Std dev.	F measure	Std dev.	AUROC	Std dev.
1	J48	89.01%	NA	91.46%	NA	0.8947	NA
2	rf	86.52%	NA	89.73%	NA	0.9462	NA
3	naive_bayes	83.69%	NA	88.32%	NA	0.9135	NA
4	JRip	86.88%	NA	89.52%	NA	0.903	NA

Table 16: `caret` results for *Data Class*

#	Classifier	Accuracy	Std dev.	F measure	Std dev.	AUROC	Std dev.
1	J48	91.13%	NA	93.54%	NA	0.9036	NA
2	rf	93.62%	NA	95.24%	NA	0.976	NA
3	naive_bayes	89.72%	NA	92.14%	NA	0.9436	NA
4	JRip	92.2%	NA	94.18%	NA	0.9177	NA

4.2 Extracted rules

The J48 and JRip algorithms provide human readable detection rules as the part of the output. We present them to show significant metrics used for certain code smells detection.

4.2.1 *Long Method*

For *Long Method*, J48 generated a decision tree that can be expressed as:

$$LOC\_M > 51 \tag{9}$$

JRip classifier computed the following rule:

$$LOC\_M \geq 52 \tag{10}$$

4.2.2 *God Class*

J48 generated a decision tree that can be expressed as following set of rules:

$$\begin{aligned}
&(WMCNAMM \leq 49 \text{ and } WMCNAMM > 8 \text{ and } LD \leq 2.028986 \text{ and } NOM \leq 14)\\
&\text{or } (WMCNAMM \leq 49 \text{ and } WMCNAMM > 32 \text{ and } LD \leq 2.028986 \text{ and } NOM > 14 \text{ and } MRD > 0.470958)\\
&\text{or } (WMCNAMM > 49 \text{ and } NOL_C > 13)\\
&\text{or } (WMCNAMM > 49 \text{ and } NOL_C \leq 5)\\
&\text{or } (WMCNAMM > 49 \text{ and } NOL_C \leq 6 \text{ and } NOL_C > 5 \text{ and } NOMM \leq 2 \text{ and } LD > 1.595745)\\
&\text{or } (WMCNAMM > 49 \text{ and } NOL_C \leq 13 \text{ and } NOL_C > 6 \text{ and } NOMM \leq 0)\\
&\text{or } (WMCNAMM > 49 \text{ and } NOL_C \leq 13 \text{ and } NOL_C > 6 \text{ and } NOMM \leq 2 \text{ and } NOMM > 0 \text{ and } LD \leq 1.647059)\\
&\text{or } (WMCNAMM > 49 \text{ and } NOL_C > 13)\\
&\text{or } (WMCNAMM > 49 \text{ and } NOL_C \leq 13 \text{ and } NOL_C > 5 \text{ and } NOMM > 2) \quad (11)
\end{aligned}$$

As for JRip, the detection rule is as follows:

$$(WMCNAMM \geq 51) \text{ or } (NOL\_C \leq 5 \text{ and } WMCNAMM \geq 32) \quad (12)$$

4.2.3 *Data Class*

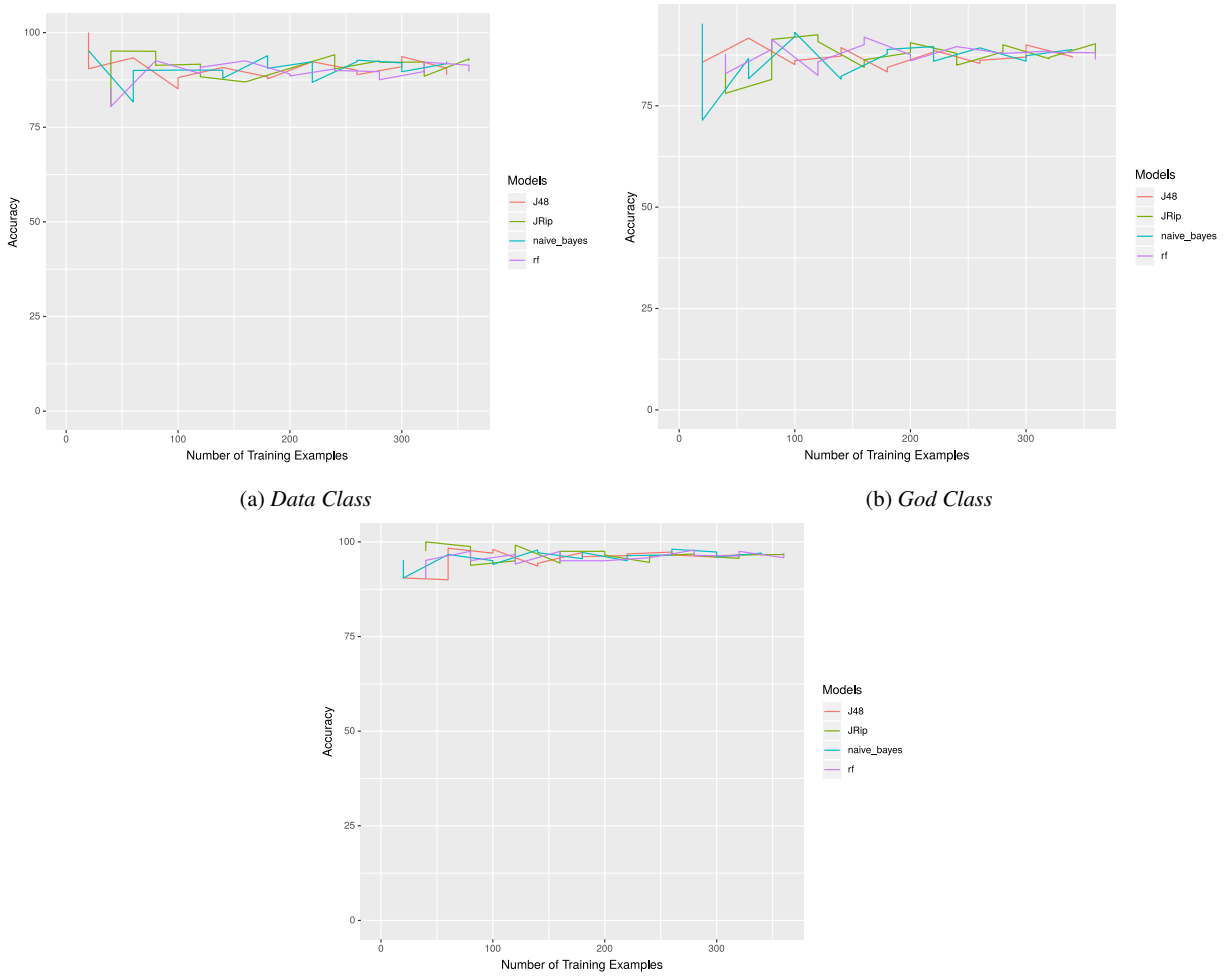
The detection rule derived from J48 can be expressed as:

$$\begin{aligned}
&(NOL_C \leq 6 \text{ and } WOC \leq 0.769231)\\
&\text{or } (NOL_C \leq 6 \text{ and } WOC > 0.769231 \text{ and } NOPA \leq 3 \text{ and } NOMM > 1)\\
&\text{or } (NOL_C \leq 6 \text{ and } WOC > 0.769231 \text{ and } NOPA > 3) \quad (13)
\end{aligned}$$

JRip classifer produces the following expression:

$$(NOL\_C \leq 6 \text{ and } WOC \leq 0.76) \text{ or } (LD \leq 2.083333 \text{ and } NOPA \geq 4) \quad (14)$$

4.3 Learning curves



(a) *Data Class*

(b) *God Class*

(c) *Long Method*

Fig. 13: Learning curves for code smells

5 Discussion

During manual tagging we have found that *Long Method* code smell occurs approximately two times less frequently than *God Class*. This might be due to new constructions contributing to shorten overall code length. J48 for *God Class* (Equation 11) shows that high lambda count and high $WMCNAMM$ metric values with $CYCLO$ as the embedded metric may indicate a *God Class*. Lambdas concise syntax contributes to the class complexity, since more operations can be expressed with less code.
Data Classes usually incorporate no logic and this is noticeable in their equation which discards higher number of lambdas. JRip detection rule (Equation 14) introduces NOL_C metric. It might be due to data classes having little to no business logic embedded in their code. Therefore low values of NOL_C metric might be used to recognize the smelly samples.
Learning curves on figures 13a, 13b and 13c depicts the performance of the classifier depending on the number of training samples. It was generated starting with 20 samples with a constant increment of 20 samples up to total

of 375 samples. All of the learning curves have non-monotonic, oscillatory trend with a slight stabilization noticeable as the number of samples increases.
An obvious weakness of our research is that we are not experienced developers and yet had to perform manual tagging for the majority of samples. Tagging is a crucial part of research that determines extracted rules. We admit that our tagging process was guided by result metrics from *JavaMetrics* tool, hence the outcome might not be quite accurate.
Another uncertainty relates to extracting Java version from build tools. There are multiple build tools in different versions for diverse Java releases specification methods that makes them hard to obtain. Moreover, repositories often consist of many projects, which build tools may also vary.
Despite the results, one should be concerned as using datasets from actively developed projects may carry potential risk in validity of the study and increase difficulty of code smell prediction.
In recent years we can observe growing awareness of the importance of writing clean code. Thus more recent dataset may bring consistently less code smells (or at least not such distinct ones) due to its better design.
Advances in building applications in Java language may contribute to a difficulty in detection of certain code smells. Using various libraries and frameworks can affect the detection of code smells as well. As an example, Lombok[7] can completely exclude presumable *Data Classes* from the detected instances due to a slight decrease in WOC metric. Therefore our approach may not be suitable for every project and shall not be taken as a generic way to predict code smells.

6 Conclusions and future directions

Using lambda constructs contributes naturally to more concise syntax which may obfuscate certain code smells like *Long Method*. Moreover, using Java Streams does not affect the increase of $CYCLO$ metric, which may further distort the predictions for code smells indicated by such metric.
In addition to extending the tool with more metrics concerning emerging Java language constructs and creating the external dependency parsing approach, we have identified the possible future directions of research and development:

- **Tagging.** Employing professional developers for manual tagging.
- **Algorithms.** Involving more ML algorithms in creating predictive models.
- **Metrics.** Extend *JavaMetrics* tool to correctly parse external dependencies.
- **Constructions.** Considering wider range of modern Java features than lambda expressions and method reference.

Acknowledgements This work has been conducted as a part of research and development project POIR.01.01.01-00-0792/16 supported by the National Centre for Research and Development (NCBiR). We would like to thank Tomasz Lewowski, Tomasz Korzeniowski, Marek Skrajnowski and the entire team from *code quest sp. z o.o.* for tagging code smells and for all of the comments and feedback from the real-world software engineering environment.

References

1. Arcelli Fontana, F., Mäntylä, M.V., Zanoni, M., Marino, A.: Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering **21**(3), 1143–1191 (2016)
2. Fontana, F.A., Mariani, E., Mornioli, A., Sormani, R., Tonello, A.: An experience report on using code smells detection tools. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 450–457 (2011). DOI 10.1109/ICSTW.2011.12
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
4. Grodzicka, H., Ziobrowski, A., Łakomiak, Z., Kawa, M., Madeyski, L.: Appendix to the paper "Code smell prediction employing machine learning meets emerging Java language constructs" (2019). URL `http://madeyski.e-informatyka.pl/download/GrodzickaEtAl19.pdf`

[7] https://projectlombok.org, access: 2019-06-12

5. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explor. Newsl. **11**(1), 10–18 (2009). DOI 10.1145/1656274.1656278. URL http://doi.acm.org/10.1145/1656274.1656278
6. Hornik, K., Buchta, C., Zeileis, A.: Open-source machine learning: R meets Weka. Computational Statistics **24**(2), 225–232 (2009). DOI 10.1007/s00180-008-0119-7
7. Hsu, C.W., Chang, C.C., Lin, C.J.: A practical guide to support vector classification. Tech. rep., Department of Computer Science, National Taiwan University (2003). URL http://www.csie.ntu.edu.tw/~cjlin/papers.html
8. Madeyski, L., Kitchenham, B.: reproducer: Reproduce Statistical Analyses and Meta-Analyses (2019). URL http://madeyski.e-informatyka.pl/reproducible-research/. R package version 0.3.0 (http://CRAN.R-project.org/package=reproducer)
9. Palomba, F.: Textual analysis for code smell detection. IEEE International Conference on Software Engineering **37**(16), 769–771 (2015)
10. Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Poshyvanyk, D., Lucia, A.D.: Mining version histories for detecting code smells. IEEE Transactions on Software Engineering **41**(5), 462–489 (2015)
11. Palomba, F., Nucci, D.D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D., De Lucia, A.: Landfill: An open dataset of code smells with public evaluation. In: Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, pp. 482–485. IEEE Press, Piscataway, NJ, USA (2015)
12. Sharma, T.: Designite: A customizable tool for smell mining in c# repositories. SATToSE **41** (2017)
13. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: Qualitas corpus: A curated collection of java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference (APSEC2010), pp. 336–345 (2010). DOI http://dx.doi.org/10.1109/APSEC.2010.46
14. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques, 2nd edn. Morgan Kaufmann, San Francisco (2005)

Appendix: Reproduction classifier comparison

Following tables present comparison of our reproduction results to Arcelli Fontana et al. [1].

Table 17: RWeka results for *Data Class* (grey) compared with Arcelli Fontana's (white)

#	Classifier	Accuracy	Std dev.	F measure	Std dev.	AUROC	Std dev.
1	B-J48 Pruned	98.78%	0.00	99.05%	0.00	0.9910	0.0050
2	B-J48 Pruned	99.02%	1.51	99.26%	1.15	0.9985	0.0064
3	B-J48 Unpruned	97.55%	0.01	98.10%	0.01	0.9856	0.0073
4	B-J48 Unpruned	98.67%	1.79	98.99%	1.86	0.9984	0.0064
5	B-J48 Reduced Error Pruning	98.09%	0.01	98.52%	0.01	0.9930	0.0045
6	B-J48 Reduced Error Pruning	98.07%	2.47	98.55%	1.36	0.9951	0.0103
7	B-JRip	98.55%	0.00	98.87%	0.00	0.9950	0.0037
8	B-JRip	98.83%	1.60	99.12%	1.21	0.9959	0.0111
9	B-Random Forest	97.55%	0.00	98.12%	0.00	0.9988	4e-04
10	B-Random Forest	98.57%	1.68	98.94%	2.08	0.9993	0.0017
11	B-Naive Bayes	86.17%	0.01	88.34%	0.01	0.9221	0.0117
12	B-Naive Bayes	97.33%	2.29	98.02%	2.01	0.9955	0.0091
13	B-SMO RBF Kernel	94.21%	0.01	95.48%	0.00	0.9848	0.0033
14	B-SMO RBF Kernel	97.14%	2.61	97.86%	2.11	0.9782	0.0286
15	B-SMO Poly Kernel	94.34%	0.00	95.59%	0.00	0.9387	0.0036
16	B-SMO Poly Kernel	96.90%	2.25	97.65%	2.35	0.9862	0.0173
17	J48 Pruned	98.47%	0.01	98.81%	0.00	0.9834	0.0076
18	J48 Pruned	98.55%	1.84	98.91%	1.39	0.9864	0.0219
19	J48 Unpruned	98.52%	0.00	98.85%	0.00	0.9816	0.0043
20	J48 Unpruned	98.38%	1.87	98.79%	1.90	0.9873	0.0208
21	J48 Reduced Error Pruning	96.86%	0.01	97.55%	0.01	0.9795	0.0081
22	J48 Reduced Error Pruning	97.98%	2.46	98.46%	1.40	0.9839	0.0211
23	JRip	97.5%	0.01	98.05%	0.01	0.9782	0.0089
24	JRip	98.17%	2.18	98.62%	2.07	0.9809	0.0241
25	Random Forest	97.55%	0.00	98.12%	0.00	0.9989	5e-04
26	Random Forest	98.95%	1.51	99.29%	2.05	0.9996	0.0014
27	Naive Bayes	79.69%	0.01	81.99%	0.01	0.9475	0.0063
28	Naive Bayes	96.12%	2.95	97.04%	1.95	0.9938	0.0099
29	SMO RBF Kernel	95.43%	0.00	96.54%	0.00	0.9394	0.0039
30	SMO RBF Kernel	97.05%	2.38	97.78%	2.11	0.9686	0.0286
31	SMO Poly Kernel	94.23%	0.00	95.51%	0.00	0.9379	0.0031
32	SMO Poly Kernel	96.60%	2.76	97.41%	2.13	0.9912	0.0138

Table 18: RWeka results for *Feature Envy* (grey) compared with Arcelli Fontana's (white)

#	Classifier	Accuracy	Std dev.	F measure	Std dev.	AUROC
1	B-J48 Pruned	96.01%	0.01	97.01%	0.00	0.9880
2	B-J48 Pruned	96.62%	2.78	97.41%	2.16	0.9900
3	B-J48 Unpruned	93.79%	0.01	95.39%	0.01	0.9684
4	B-J48 Unpruned	96.50%	2.96	97.37%	2.37	0.9899
5	B-J48 Reduced Error Pruning	95.28%	0.01	96.48%	0.01	0.9829
6	B-J48 Reduced Error Pruning	95.90%	3.11	96.90%	2.24	0.9866
7	B-JRip	96.19%	0.01	97.15%	0.00	0.9869
8	B-JRip	96.64%	2.84	97.44%	2.16	0.9891
9	B-Random Forest	92.14%	0.01	94.24%	0.01	0.9817
10	B-Random Forest	96.40%	2.70	97.29%	2.73	0.9886
11	B-Naive Bayes	90.44%	0.01	92.73%	0.01	0.9496
12	B-Naive Bayes	91.50%	4.20	93.56%	2.93	0.9527
13	B-SMO RBF Kernel	89.38%	0.01	92.17%	0.01	0.9467
14	B-SMO RBF Kernel	93.88%	3.20	95.40%	3.17	0.9369
15	B-SMO Poly Kernel	90.82%	0.01	93.34%	0.01	0.8816
16	B-SMO Poly Kernel	92.05%	3.50	94.06%	3.07	0.9541
17	J48 Pruned	95.31%	0.01	96.47%	0.00	0.9517
18	J48 Pruned	95.95%	2.77	96.91%	2.16	0.9647
19	J48 Unpruned	95.03%	0.01	96.26%	0.01	0.9523
20	J48 Unpruned	96.12%	2.71	97.04%	2.17	0.9661
21	J48 Reduced Error Pruning	95.10%	0.01	96.31%	0.01	0.9567
22	J48 Reduced Error Pruning	95.93%	2.80	96.89%	2.10	0.9646
23	JRip	94.95%	0.00	96.18%	0.00	0.9534
24	JRip	95.67%	3.13	96.69%	2.34	0.9584
25	Random Forest	92.04%	0.01	94.16%	0.01	0.9813
26	Random Forest	96.26%	2.86	97.19%	2.57	0.9902
27	Naive Bayes	86.26%	0.01	89.76%	0.00	0.9241
28	Naive Bayes	85.50%	6.09	89.17%	2.46	0.9194
29	SMO RBF Kernel	80.15%	0.00	87.07%	0.00	0.7008
30	SMO RBF Kernel	93.83%	3.39	95.36%	3.15	0.9309
31	SMO Poly Kernel	90.90%	0.00	93.41%	0.00	0.8817
32	SMO Poly Kernel	95.45%	3.61	96.58%	2.80	0.9484

Table 19: RWeka results for *God Class* (grey) compared with Arcelli Fontana's (white)

#	Classifier	Accuracy	Std dev.	F measure	Std dev.	AUROC
1	B-J48 Pruned	97.14%	0.00	97.86%	0.00	0.9834
2	B-J48 Pruned	97.02%	2.82	97.75%	2.14	0.9923
3	B-J48 Unpruned	96.68%	0.01	97.51%	0.01	0.9794
4	B-J48 Unpruned	97.02%	2.88	97.75%	2.00	0.9925
5	B-J48 Reduced Error Pruning	97.37%	0.00	98.03%	0.00	0.9885
6	B-J48 Reduced Error Pruning	97.26%	2.64	97.94%	2.18	0.9861
7	B-JRip	96.73%	0.00	97.55%	0.00	0.9879
8	B-JRip	96.90%	3.15	97.67%	2.39	0.9916
9	B-Random Forest	97.55%	0.00	98.17%	0.00	0.9951
10	B-Random Forest	96.95%	2.86	97.70%	2.67	0.9890
11	B-Naive Bayes	94.59%	0.00	95.91%	0.00	0.9757
12	B-Naive Bayes	97.54%	2.65	97.70%	2.72	0.9871
13	B-SMO RBF Kernel	94.92%	0.01	96.21%	0.01	0.9835
14	B-SMO RBF Kernel	94.62%	3.34	95.98%	2.89	0.9838
15	B-SMO Poly Kernel	95.20%	0.00	96.45%	0.00	0.9399
16	B-SMO Poly Kernel	94.33%	3.58	95.75%	2.48	0.9799
17	J48 Pruned	96.61%	0.00	97.46%	0.00	0.9600
18	J48 Pruned	97.31%	2.51	97.98%	1.89	0.9783
19	J48 Unpruned	96.58%	0.00	97.45%	0.00	0.9718
20	J48 Unpruned	97.31%	2.51	97.98%	1.93	0.9783
21	J48 Reduced Error Pruning	97.73%	0.00	98.29%	0.00	0.9761
22	J48 Reduced Error Pruning	97.29%	2.52	97.94%	1.89	0.9742
23	JRip	97.65%	0.00	98.24%	0.00	0.9694
24	JRip	97.12%	2.70	97.81%	2.48	0.9717
25	Random Forest	97.27%	0.01	97.96%	0.00	0.9953
26	Random Forest	97.33%	2.64	97.98%	2.24	0.9927
27	Naive Bayes	94.03%	0.01	95.41%	0.00	0.9820
28	Naive Bayes	97.55%	2.51	98.14%	2.20	0.9916
29	SMO RBF Kernel	89.36%	0.00	92.58%	0.00	0.8435
30	SMO RBF Kernel	95.43%	3.26	96.62%	2.40	0.9427
31	SMO Poly Kernel	95.38%	0.00	96.59%	0.00	0.9410
32	SMO Poly Kernel	95.71%	3.14	96.83%	2.40	0.9459