Comparing one- and binary-class SVM-based software defect predictors

George Ciubotariu
Babeș-Bolyai University

WeADL 2023 Workshop

The workshop is organized under the umbrella of WeaMyL (https://weamyl.met.no/), project funded by the EEA and Norway Grants under the number RO-NO-2019-0133. Contract: No 26/2020.

Working together for a green, competitive and inclusive Europe

Contents

Problem statement

What is Software Defect Prediction (SDP)?

Short talk: detection vs prediction

abstraction level : line of code / function / class / file / directory / etc

pre-processing : raw code is passed through NLP tools

input = vectorial representations or other aggregated features

algorithm = Machine Learning classifier

output = defect (positive class) / non-defect (negative class)

Raw Text Pre-Processing

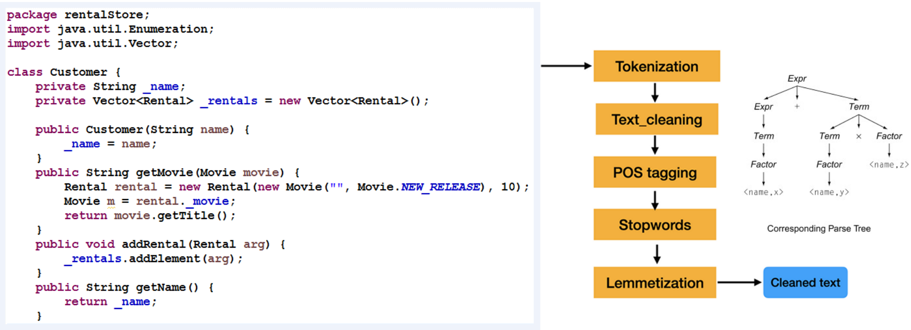

Figure: Code pre-processing pipeline

Source Code Feature Extractors

Features:

- based on static code
- based on the warnings produced by the PMD analysis tool [1]
- extracted from the Abstract Syntax Tree (AST) representation
- based on code churn [2] [3] [4]

Relevance

- increasing the dev team size does not always help
- automation is always desirable for reducing costs
- incredibly large and buggy legacy software
- fast-paced industry environments
- AI can perform regular checks for ensuring software reliability
- SDP plug-ins for IDEs could be developed to boost productivity

Software Development Difficulty

```
// Dear programmer:
// When I wrote this code, only god and
// I knew how it worked.
// Now, only god knows it!
//
// Therefore, if you are trying to optimize
// this routine and it fails (most surely),
// please increase this counter as a
// warning for the next person:
//
// total_hours_wasted_here = 254
//

```

DataRace.java   Trace.aj

```java
package com.race.ex;

public class DataRace extends Thread {

    private static volatile int count = 0; // shared memory

    public void run() {
        int x = count;
        count = x + 1;
    }

    public static void main(String args[]) {
        Thread t1 = new DataRace();
        Thread t2 = new DataRace();
        t1.start();
        t2.start();
    }
}

```

Figure: Code samples

Difficulty in Industry

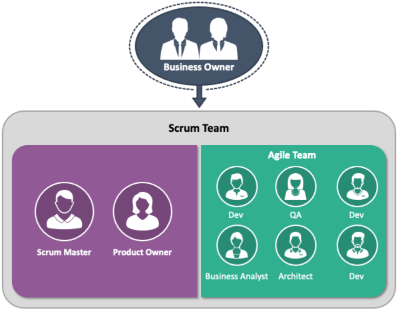

Figure: Agile team structure

Types of Defects

What buggy code may look like:

- compilation errors (should not pass QA)
- copy-pasted problems
- concurrency bugs
- unhandled connectivity errors (DB / microservices)
- platform limitations or hardware issues
- client requirement misunderstanding
- coding standards or other style conventions
- etc.

Overall Difficulty

- severe class imbalance
- difficult to pinpoint defects
- a lot of context is needed to understand the problem
- code correctness may vary along the product's lifetime
- many types of defects that have different degrees of severity
- regular supervised learning ML models underperform

Solutions from Literature

Several traditional solutions could be:

- statistical rebalancing or oversampling techniques
- artificial data generation or augmentation
- merging of multiple data sets

However, insufficient for improving Supervised Learning ML models.
Thus, our focus switches onto Unsupervised Learning formulations.

One-Class Classification for SDP

- OCC may be more suitable for SDP
- Supervised vs Unsupervised ML
- SVM vs OCSVM
- OCSVM_{+} vs OCSVM_{-}

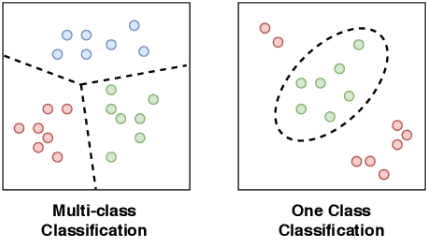

Figure: Types of classification models

Research Questions and Original Contributions

- **RQ1**: *How does the performance of OCSVM trained only on defective data compare to that of the same model solely trained on non-defective entities?*
- **RQ2**: *Does the OCSVM models bring an improvement in SDP compared to the classical binary SVM and other baseline methods?*

Calcite Dataset

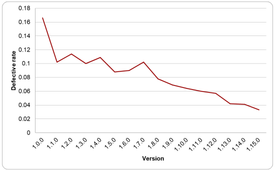

Figure: Defective rates for all 16 Calcite [5] versions.

Figure: Imbalanced data used in E1 & E2 for training the OCC models.

Methodology

- Classes used in SDP:
  - software faults (denoted by "$+$" and referred to as the *positive* class)
  - non-defective software entities (denoted by "$-$", the *negative* class)
- Our framework based on scikit-learn:
  - SVC and two OCSVM models from `scikit-learn`
    - **(1)** OCSVM_{+} - trained only on *positive* instances
    - **(2)** OCSVM_{-} - trained only on *negative* instances
- Experimental scenarios:

E1 progressive independent prediction experiments: the models are trained on version k (i.e., data set $\mathcal{D}_k$) and then tested on version $k+1$ (i.e., data set $\mathcal{D}_{k+1}$), $\forall k, 0 \leq k \leq n-1$.

E2 historical system evolution track, for assessing the real-life defect prediction capabilities: the models are trained on the instances from versions 0..k (i.e., $\bigcup\limits_{i=0}^{k} \mathcal{D}_i$) and then tested on version $k+1$ (i.e., data set $\mathcal{D}_{k+1}$), $\forall k, 0 \leq k \leq n-1$.

Performance Metrics

- Probability of detection (POD)
- False alarm ratio (FAR)
- Critical success index (CSI)
- Area under the ROC curve (AUC)
- F-score for the positive class (F1)

Results E1

Version for training (k)	Version for testing ($k+1$)	Model	TP	FP	TN	FN	*POD* (↑)	*FAR* (↓)	*CSI* (↑)	*AUC* (↑)	*F1* (↑)
1.0.0	**1.1.0**	**OCSVM$_+$**	70	441	549	43	**0.619**	0.863	**0.126**	**0.587**	**0.224**
		OCSVM$_-$	58	362	628	55	0.513	**0.862**	0.122	0.574	0.218
1.1.0	**1.2.0**	**OCSVM$_+$**	78	459	523	48	**0.619**	**0.855**	**0.133**	**0.576**	**0.235**
		OCSVM$_-$	61	372	610	65	0.484	0.859	0.122	0.553	0.218
1.2.0	**1.3.0**	**OCSVM$_+$**	80	560	443	32	**0.714**	0.875	0.119	0.578	0.213
		OCSVM$_-$	71	464	539	41	0.634	**0.867**	**0.123**	**0.586**	**0.219**
1.3.0	**1.4.0**	**OCSVM$_+$**	97	596	408	26	**0.789**	0.860	0.135	**0.598**	0.238
		OCSVM$_-$	82	479	525	41	0.667	**0.854**	**0.136**	0.595	**0.240**
1.4.0	**1.5.0**	**OCSVM$_+$**	72	508	565	31	**0.699**	**0.876**	**0.118**	**0.613**	**0.211**
		OCSVM$_-$	60	423	650	43	0.583	**0.876**	0.114	0.594	0.205
1.5.0	**1.6.0**	**OCSVM$_+$**	78	561	525	29	**0.729**	**0.878**	**0.117**	**0.606**	**0.209**
		OCSVM$_-$	61	441	645	46	0.570	**0.878**	0.111	0.582	0.200
1.6.0	**1.7.0**	**OCSVM$_+$**	85	554	570	43	**0.664**	**0.867**	**0.125**	**0.586**	**0.222**
		OCSVM$_-$	78	525	599	50	0.609	0.871	0.119	0.571	0.213
1.7.0	**1.8.0**	**OCSVM$_+$**	47	357	843	54	0.465	**0.884**	**0.103**	**0.584**	**0.186**
		OCSVM$_-$	60	596	604	41	**0.594**	0.909	0.086	0.549	0.159
1.8.0	**1.9.0**	**OCSVM$_+$**	45	452	768	45	0.500	**0.909**	**0.083**	0.565	**0.153**
		OCSVM$_-$	66	731	489	24	**0.733**	0.917	0.080	**0.567**	0.149
1.9.0	**1.10.0**	**OCSVM$_+$**	43	544	682	41	0.512	0.927	0.068	0.534	0.128
		OCSVM$_-$	54	611	615	30	**0.643**	**0.919**	**0.078**	**0.572**	**0.144**
1.10.0	**1.11.0**	**OCSVM$_+$**	43	443	808	37	0.538	**0.912**	**0.082**	0.592	**0.152**
		OCSVM$_-$	66	725	526	14	**0.825**	0.917	**0.082**	**0.623**	**0.152**
1.11.0	**1.12.0**	**OCSVM$_+$**	55	715	619	26	0.679	0.929	0.069	0.572	0.129
		OCSVM$_-$	62	746	588	19	**0.765**	**0.923**	**0.075**	**0.603**	**0.139**
1.12.0	**1.13.0**	**OCSVM$_+$**	37	646	576	16	0.698	0.946	0.053	0.585	0.101
		OCSVM$_-$	39	632	590	14	**0.736**	**0.942**	**0.057**	**0.609**	**0.108**
1.13.0	**1.14.0**	**OCSVM$_+$**	40	643	612	13	0.755	**0.941**	**0.057**	0.621	**0.109**
		OCSVM$_-$	43	712	543	10	**0.811**	0.943	0.056	**0.622**	0.106
1.14.0	**1.15.0**	**OCSVM$_+$**	33	656	651	12	**0.733**	0.952	0.047	0.616	0.090
		OCSVM$_-$	32	611	696	13	0.711	**0.950**	**0.049**	**0.622**	**0.093**

Table: Experimental results of the first experiment (**E1**). For each testing case (trained model on a version k and tested on version $k+1$) and OCSVM models, the confusion matrix is provided together with the performance metrics values.

Results E2

Versions for training ($0..k$)	Version for testing ($k+1$)	Model	TP	FP	TN	FN	*POD* (↑)	*FAR* (↓)	*CSI* (↑)	*AUC* (↑)	*F*1 (↑)
1.0.0..1.0.0	**1.1.0**	**OCSVM**$_{+}$	70	441	549	43	**0.619**	0.863	**0.126**	**0.587**	**0.224**
		OCSVM$_{-}$	58	362	628	55	0.513	**0.862**	0.122	0.574	0.218
1.0.0..1.1.0	**1.2.0**	**OCSVM**$_{+}$	77	455	527	49	**0.611**	**0.855**	**0.133**	**0.574**	**0.234**
		OCSVM$_{-}$	60	354	628	66	0.476	**0.855**	0.125	0.558	0.222
1.0.0..1.2.0	**1.3.0**	**OCSVM**$_{+}$	67	479	524	45	**0.598**	0.877	0.113	0.560	0.204
		OCSVM$_{-}$	64	417	586	48	0.571	**0.867**	**0.121**	**0.578**	**0.216**
1.0.0..1.3.0	**1.4.0**	**OCSVM**$_{+}$	73	473	531	50	0.593	0.866	0.122	0.561	0.218
		OCSVM$_{-}$	75	425	579	48	**0.610**	**0.850**	**0.137**	**0.593**	**0.241**
1.0.0..1.4.0	**1.5.0**	**OCSVM**$_{+}$	73	519	554	30	**0.709**	**0.877**	**0.117**	**0.613**	**0.210**
		OCSVM$_{-}$	64	499	574	39	0.621	0.886	0.106	0.578	0.192
1.0.0..1.5.0	**1.6.0**	**OCSVM**$_{+}$	81	577	509	26	**0.757**	**0.877**	**0.118**	**0.613**	**0.212**
		OCSVM$_{-}$	67	517	569	40	0.626	0.885	0.107	0.575	0.194
1.0.0..1.6.0	**1.7.0**	**OCSVM**$_{+}$	85	558	566	43	**0.664**	**0.868**	**0.124**	**0.584**	**0.220**
		OCSVM$_{-}$	78	519	605	50	0.609	0.869	0.121	0.574	0.215
1.0.0..1.7.0	**1.8.0**	**OCSVM**$_{+}$	50	382	818	51	0.495	**0.884**	**0.104**	**0.588**	**0.188**
		OCSVM$_{-}$	63	635	656	38	**0.624**	0.910	0.086	0.547	0.158
1.0.0..1.8.0	**1.9.0**	**OCSVM**$_{+}$	42	406	814	48	0.467	**0.906**	**0.085**	**0.567**	**0.156**
		OCSVM$_{-}$	53	546	656	37	**0.589**	0.912	0.083	**0.567**	0.154
1.0.0..1.9.0	**1.10.0**	**OCSVM**$_{+}$	39	426	800	45	0.464	**0.916**	0.076	0.558	0.142
		OCSVM$_{-}$	52	566	660	32	**0.619**	**0.916**	**0.080**	**0.579**	**0.148**
1.0.0..1.10.0	**1.11.0**	**OCSVM**$_{+}$	37	376	875	43	0.463	**0.910**	**0.081**	**0.581**	**0.150**
		OCSVM$_{-}$	53	610	641	34	**0.609**	0.920	0.076	0.561	0.141
1.0.0..1.11.0	**1.12.0**	**OCSVM**$_{+}$	37	438	896	44	**0.457**	0.922	0.071	0.564	0.133
		OCSVM$_{-}$	37	310	1024	44	**0.457**	**0.893**	**0.095**	**0.612**	**0.173**
1.0.0..1.12.0	**1.13.0**	**OCSVM**$_{+}$	43	734	488	10	**0.811**	0.945	0.055	0.605	0.104
		OCSVM$_{-}$	39	603	619	14	0.736	**0.939**	**0.059**	**0.621**	**0.112**
1.0.0..1.13.0	**1.14.0**	**OCSVM**$_{+}$	36	612	643	17	**0.679**	**0.944**	**0.054**	**0.596**	**0.103**
		OCSVM$_{-}$	32	546	709	21	0.604	0.945	0.053	0.584	0.101
1.0.0..1.14.0	**1.15.0**	**OCSVM**$_{+}$	29	623	684	16	0.644	0.956	0.043	0.584	0.083
		OCSVM$_{-}$	32	611	696	13	**0.711**	**0.950**	**0.049**	**0.622**	**0.093**

Table: Experimental results of the second experiment (**E2**). For each testing case (trained model on versions from 0 to k and tested on version $k+1$) and OCSVM models.

Discussion - RQ1

Experiment	Win	Lose	Tie
E1	**37**	34	4
E2	**41**	30	4
TOTAL	**78**	64	8

Table: Comparison between $OCSVM_+$ and $OCSVM_-$ models considering the performed experiments: E1 and E2.

Discussion - RQ2

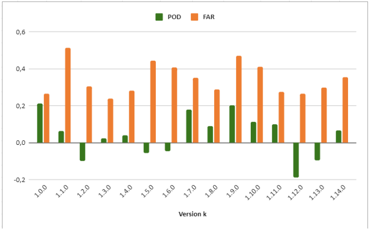

Figure: Improvement in *POD* and *FAR* achieved by the binary SVC model compared to the OCSVM_{+} model, for all testing scenarios in E2.

Discussion - RQ2

Table: Confusion matrices for the random guessing and ZeroR classifiers on a defect data set with n instances and a defective rate r.

Baseline	TP	TN	FP	FN
RG	$n \cdot r^2$	$n \cdot (1-r)^2$	$Fn \cdot r \cdot (1-r)$	$n \cdot r \cdot (1-r)$
ZeroR	0	$n \cdot (1-r)$	0	$n \cdot r$

Table: Average improvement achieved by OCSVM_+ over the RG and ZeroR baselines for each of the performance metrics used for evaluation.

Improvement OCSVM_+ vs.	*POD*	*Spec*	*FAR*	*CSI*	*AUC*	*F1*
RG	**53%**	-36%	**3%**	**5%**	**8%**	**10%**
ZeroR	**60%**	**56%**	**10%**	**9%**	**58%**	**17%**

Conclusions:

- for finding error-prone source code, we may need to either ensure that the labels are appropriate and the bug descriptions are more informative
- we believe that we could focus more on defective instances during training, since defects are more concise, and don't change their characteristics during the development stages of the software, while non-defects are more volatile, subjective, and interpretable

Future Enhancements:

- verify the findings of the current study in a cross-version SDP scenario on another Apache software systems, by training the OCC model on the software defects from all versions of a software system
- extend the AUC-based evaluation of the results by considering a recent work [6] that describes a deep ROC analysis

Thank you!

Questions?

Bibliography I

- GitHub, "PMD - An extensible cross-language static code analyzer," 2023.
  https://pmd.github.io/.
- R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *ICSE '08*, (New York, NY, USA), p. 181–190, ACM, 2008.
- A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 78–88, 2009.
- M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, p. 531–577, aug 2012.

Bibliography II

E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources," in *Proceedings of SIGMOD '18*, (New York, NY, USA), p. 221–230, ACM, 2018.

A. M. Carrington, D. G. Manuel, and et al., "Deep ROC Analysis and AUC as Balanced Average Accuracy, for Improved Classifier Selection, Audit and Explanation," *IEEE Trans. Pattern Anal. Mach.*, vol. 45, no. 1, pp. 329–341, 2023.