HOW HIGH WILL IT BE?

USING MACHINE LEARNING MODELS TO PREDICT BRANCH COVERAGE IN AUTOMATED TESTING

G. GRANO, T. TITOV, S. PANICHELLA, H. GALL

MALTESQUE@SANER 2018, 20, CAMPOBASSO (ITALY)

✉ GRANO@IFI.UZH.CH 🐦 GIOGRANO90

U.S. ECONOMY IMPACT [1]

$59.5 BILLION

$22.2 BILLION

AVOIDABLE

0.6% GDP

1 HTTP://WWW.ABEACHA.COM/NIST_PRESS_RELEASE_BUGS_COST.HTML

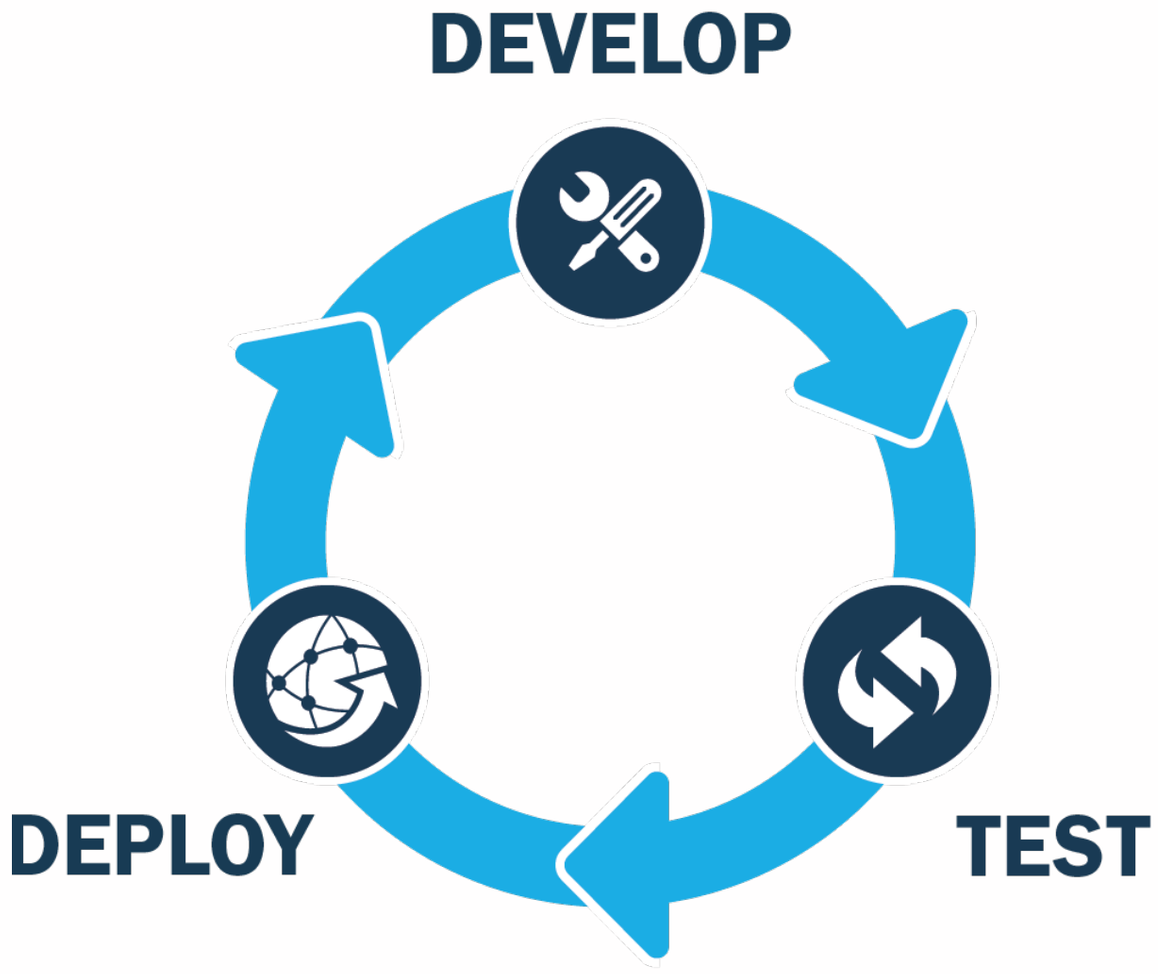
2 - GIOVANNI GRANO @ S.E.A.L.

CONTINUOUS INTEGRATION

THE WAY WE DEVELOP AND RELEASE CODE IS RAPIDLY CHANGING

IN AN IDEAL WORLD, AT EVERY SINGLE COMMIT THE ENTIRE TEST SUITE SHOULD BE EXECUTED ...

... BUT COMPANIES LIKE GOOGLE COMMIT ABOUT 16.000 CHANGES PER DAY!



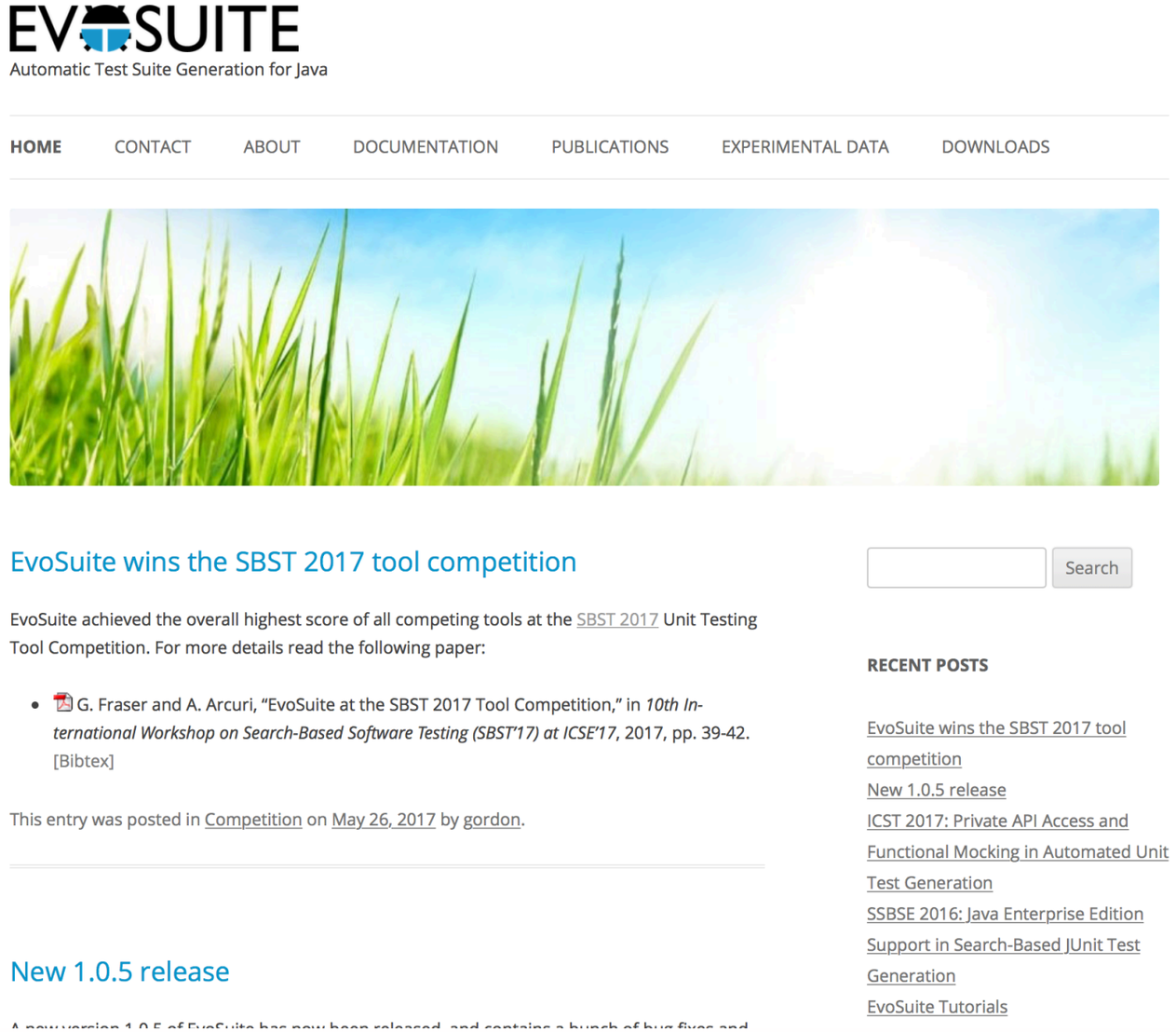
TEST DATA GENERATION

PRETTY ACTIVE RESEARCH AREA IN THE LAST YEARS

MATURE TOOLS ABLE TO GENERATE TEST SUITES WITH HIGH COVERAGE:

- EVOSUITE
- RANDOOP
- ...
- AND MANY MORE!

HOW DO THEY FIT IN SUCH A CI/CD ENVIRONMENT?



CONTINUOUS TESTING GENERATION [2]

CONTINUOUS INTEGRATION ENHANCED WITH AUTOMATED TEST GENERATION

IT RAISES MANY QUESTIONS:

- TESTING ORDER
- HOW MUCH TIME TO SPEND PER CLASS

Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation

José Campos[1] Andrea Arcuri[2] Gordon Fraser[1] Rui Abreu[3]

[1]Department of Computer Science, University of Sheffield, UK
[2]Certus Software V&V Center Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway
[3]Faculty of Engineering, University of Porto Porto, Portugal

ABSTRACT

In object oriented software development, automated unit test generation tools typically target one class at a time. A class, however, is usually part of a software project consisting of more than one class, and these are subject to changes over time. This context of a class offers significant potential to improve test generation for individual classes. In this paper, we introduce *Continuous Test Generation* (CTG), which includes automated unit test generation during continuous integration (i.e., infrastructure that regularly builds and tests software projects). CTG offers several benefits: First, it answers the question of how much time to spend on each class in a project. Second, it helps to decide in which order to test them. Finally, it answers the question of which classes should be subjected to test generation in the first place. We have implemented CTG using the EvoSuite unit test generation tool, and performed experiments using eight of the most popular open source projects available on GitHub, ten randomly selected projects from the SF100 corpus, and five industrial projects. Our experiments demonstrate improvements of up to +58% for branch coverage and up to +69% for thrown undeclared exceptions, while reducing the time spent on test generation by up to +83%.

Categories and Subject Descriptors. D.2.5 [Software Engineering]: Testing and Debugging – *Testing Tools*;

General Terms. Algorithms, Experimentation, Reliability

Keywords. Unit testing, automated test generation, continuous testing, continuous integration

1. INTRODUCTION

Research in software testing has resulted in advanced unit test generation tools such as EvoSuite [7] or Pex [33]. Even though these tools make it feasible for developers to apply automated test generation on an individual class during development, testing an entire project consisting of many classes in an interactive development environment is still problematic: Systematic unit test generation is usually too computationally *expensive* to be used by developers on entire projects. Thus, most unit test generation tools are based on the scenario that each class in a project is considered a unit and tested independently.

In practice, unit test generation may not always be performed on an individual basis. For instance, in industry there are often requirements on the minimum level of code coverage that needs to be achieved in a software project, meaning that test generation may need to be applied to *all* classes. As the software project evolves, involving code changes in multiple sites, test generation may be repeated to maintain and improve the degree of unit testing. Yet another scenario is that an automated test case generation tool might be applied to all classes when introduced for the first time in a legacy project. If the tool does not work convincingly well in such a case, then likely the tool will not be adopted.

By considering a software project and its evolution as a whole, rather than each class independently, there is the potential to use the context information for improving unit test generation:

- When generating test cases for a set of classes, it would be sub-optimal to use the same amount of computational resources for all of them, especially when there are at the same time both trivial classes (e.g., only having get and set methods) and complex classes full of non-linear predicates.
- Test suites generated for one class could be used to help the test data generation for other classes, for example using different types of seeding strategies [9].
- Finally, test suites generated for one revision of a project can be helpful in producing new test cases for a new revision.

An attractive way to exploit this potential lies in using *continuous integration* [6]: In continuous integration, a software project is hosted on a controlled version repository (e.g., SVN or Git) and, at each commit, the project is built and the regression test suites are run to verify that the new added code does not break anything in the application. Continuous integration is typically run on powerful servers, and can often resort to build farms or cloud-based infrastructure to speed up the build process for large projects. This opens doors for automated test generation tools, in order to enhance the typically manually generated regression test suites with automatically generated test cases, and it allows the test generation tools to exploit the advantages offered when testing a project as a whole.

In this paper, we introduce *Continuous Test Generation (CTG)*, which enhances continuous integration with automated test generation. This integration raises many questions on how to test the classes in a software project: For instance, in which order should they be tested, how much time to spend on each class, and which information can be carried over from the tests of one class to another? To provide first answers to some of these questions, we have implemented CTG as an extension to the EvoSuite test generation tool and performed experiments on a range of different software projects. In detail, the contributions of this paper are as follows:

ASE'14, September 15-19, 2014, Västerås, Sweden

TEST SUITE AUGMENTATION^3

AUTOMATIC GENERATION CONSIDERING CODE CHANGES AND THEIR EFFECT ON THE PREVIOUS CODEBASE

HARDY DOABLE TO THE EXPENSIVE AMOUNT OF TIME NEEDED TO GENERATE TESTS

Directed Test Suite Augmentation: Techniques and Tradeoffs

Zhihong Xu†, Yunho Kim*, Moonzoo Kim*, Gregg Rothermel†, Myra B. Cohen†

†Department of Computer Science and Engineering
University of Nebraska - Lincoln
{zxu,grother,myra}@cse.unl.edu

*Computer Science Department
Korea Advanced Institute of Science and Technology
kimyunho@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

ABSTRACT

Test suite augmentation techniques are used in regression testing to identify code elements affected by changes and to generate test cases to cover those elements. Our preliminary work suggests that several factors influence the cost and effectiveness of test suite augmentation techniques. These include the order in which affected elements are considered while generating test cases, the manner in which existing regression test cases and newly generated test cases are used, and the algorithm used to generate test cases. In this work, we present the results of an empirical study examining these factors, considering two test case generation algorithms (concolic and genetic). The results of our experiment show that the primary factor affecting augmentation is the test case generation algorithm utilized; this affects both cost and effectiveness. The manner in which existing and newly generated test cases are utilized also has a substantial effect on efficiency but a lesser effect on effectiveness. The order in which affected elements are considered turns out to have relatively few effects when using concolic test case generation, but more substantial effects when using genetic test case generation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

1. INTRODUCTION

Software engineers use regression testing to validate software as it evolves. To do this cost-effectively, they often begin by running existing test cases. Existing test cases, however, may not be adequate to validate the code or system behaviors that are present in a new version of a system. *Test suite augmentation techniques* (e.g., [2, 32, 42]) address this problem, by identifying where new test cases are needed and then creating them.

Despite the need for test suite augmentation, most research on regression testing has focused on using existing test cases. There has been research on approaches for *identifying affected elements* (code components potentially affected by changes) [2, 5, 20, 30, 32], but these approaches leave the task of generating new test cases to engineers. There has been research on automatically generating test cases given pre-supplied coverage goals (e.g., [13, 29, 34, 37]), but this research has not attempted to integrate the test case generation task with reuse of existing test cases for augmentation.

In principle, any test case generation technique could be used to generate test cases for a modified program. We believe, however, that test case generation techniques that leverage existing test cases hold the greatest promise where test suite augmentation is concerned. This is because existing test cases provide a rich source of data on potential inputs and code reachability, and existing test cases are naturally available as a starting point in the regression testing context. Further, recent research on test case generation has resulted in techniques that rely on dynamic test execution, and such techniques can naturally leverage existing test cases.

In prior work [42] we developed a *directed test suite augmentation technique*. The technique begins by using a regression test selection algorithm [31] to identify code affected by changes and existing test cases relevant to testing that code. The technique then uses the identified test cases to seed a concolic test case generation approach [34] to create test cases that execute the affected code. A case study shows that the approach improves both the efficiency of the technique and its ability to cover affected elements. Further work [41] examined a similar approach to augmentation using a genetic algorithm for test case generation.

While these initial results are encouraging, our attempts to create augmentation techniques suggest that several factors can potentially influence the cost and effectiveness of those techniques. Three factors in particular appear to be: (1) the order in which affected elements are considered while generating test cases, (2) the manner in which existing and newly generated test cases are used, and (3) the algorithm used to generate test cases.

To create effective test suite augmentation techniques we need to understand the influence of the foregoing factors. Based on such an understanding, we should be better able to create augmentation techniques that leverage test cases in a cost-effective manner. We have therefore designed and conducted a controlled experiment investigating these factors in the context of test suite augmentation. Our experiment considers concolic and genetic test case generation algorithms, two different orderings of affected elements, and two different manners of using existing test cases. We consider each relevant combination on these on four object programs, measuring the effectiveness of the approaches in terms of code coverage, and their costs in terms of the time required to perform augmentation.

The results of our experiment show that among the factors that we consider, the primary factor affecting augmentation is the algorithm utilized to generate test cases; this affects both augmentation cost and effectiveness. The manner in which existing and newly generated test cases are utilized also has a substantial effect on ef-

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

COVERAGE PREDICTION

KNOWING A PRIORI THE COVERAGE ACHIEVED BY TEST DATA GENERATION TOOLS

- MAXIMIZE THE COVERAGE FOR THE ENTIRE SYSTEM GIVEN AN AMOUNT OF TIME
- BUDGET ALLOCATION FOR CRITICAL COMPONENTS

RESEARCH QUESTIONS

- RQ1: WHICH TYPE OF FEATURES CAN WE LEVERAGE TO TRAIN MACHINE LEARNING MODELS TO PREDICT THE BRANCH COVERAGE ACHIEVED BY TEST DATA GENERATION TOOLS?
- RQ2: TO WHAT EXTEND CAN WE PREDICT THE COVERAGE ACHIEVED BY TEST DATA GENERATION TOOLS?

PROJECT SELECTION

OPEN SOURCE PROJECTS FROM DEFECT4J

- APACHE CASSANDRA
- APACHE IVY
- GOOGLE GUAVA
- GOOGLE DAGGER

	GUAVA	CASSANDRA	DAGGER	IVY
LOC	78.525	220.573	848	50.430
JAVA FILES	538	1.474	43	464

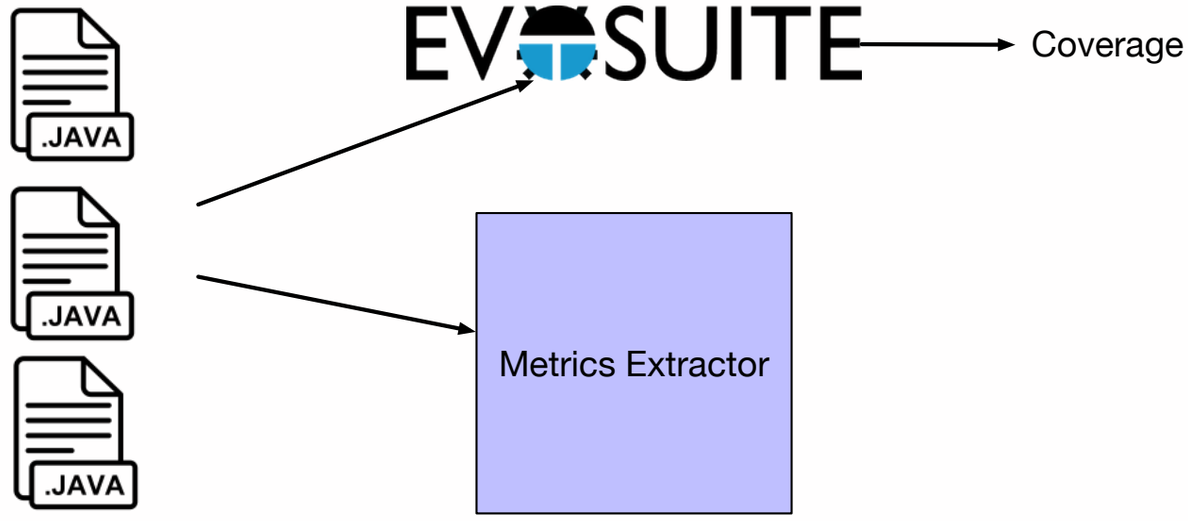
TRAINING SET

- RUN EVOSUITE
- LABELED DATA (WITH COVERAGE)
- COMPUTED METRICS = INPUT VARIABLES

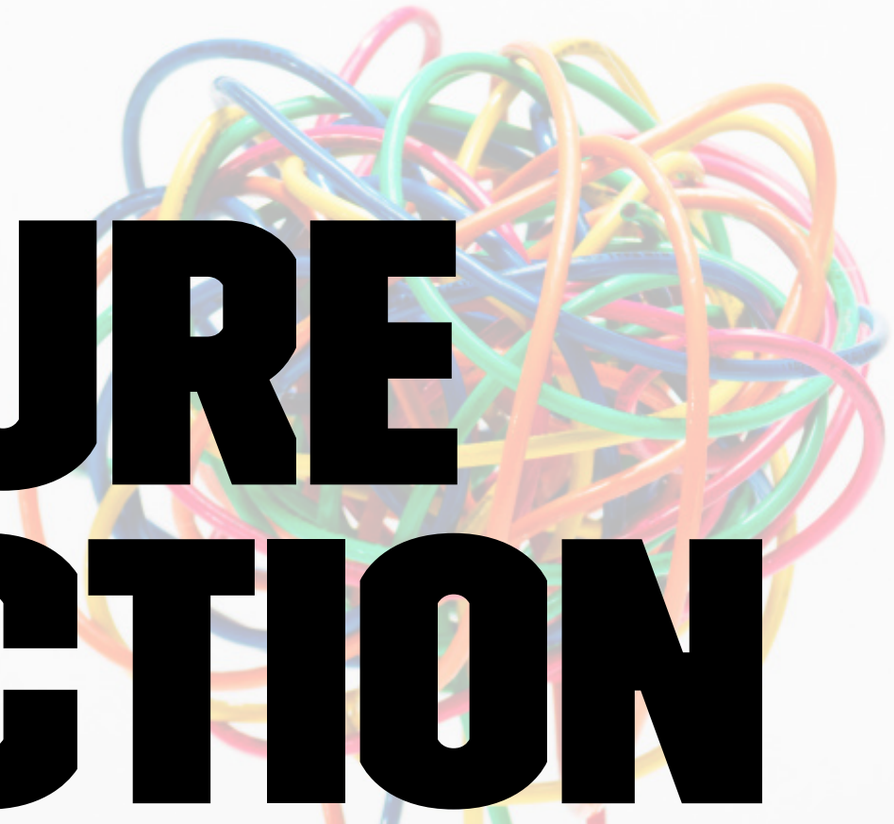
$$f : X \longrightarrow Y$$

THREATS:

- WE DID IT ONCE



FEATURE SELECTION



GOAL: CAPTURE COMPLEXITY

PACKAGE LEVEL FEATURES

COMPUTED WITH JDEPEND [4]

NAME	DESCRIPTION
CA	INDICATOR OF THE PACKAGE'S RESPONSIBILITY
CE	INDICATOR OF THE PACKAGE'S INDEPENDENCE
A	ABSTRACT CLASSES / TOTAL NUMBER OF CLASSES
I	INDICATOR OF THE PACKAGE'S RESILIENCE TO CHANGE
...	...

4 HTTPS://GITHUB.COM/CLARKWARE/JDEPEND

CK AND OO FEATURE

CK TOOL PROVIDED BY ANICHE [5]

NAME	DESCRIPTION
CBO	COUPLING BETWEEN OBJECTS
DIT	DEPTH OF INHERITANCE TREE
NOC	NUMBER OF CHILDREN
NOSF	NUMBER OF STATIC FIELD
...	...

[5] HTTPS://GITHUB.COM/MAURICIOANICHE/CK

JAVA RESERVED KEYWORDS

TO CAPTURE ADDITIONAL COMPLEXITY

PREVIOUSLY USED IN INFORMATION RETRIEVAL AS A FEATURE [6]

52 JAVA RESERVED KEYWORDS

[6] SANDERSON ET AL **THE HISTORY OF INFORMATION RETRIEVAL RESEARCH**

GRID SELECTION

BEST HYPER-PARAMETERS
3-CROSS FOLD VALIDATION

FEATURE TRANSFORMATION

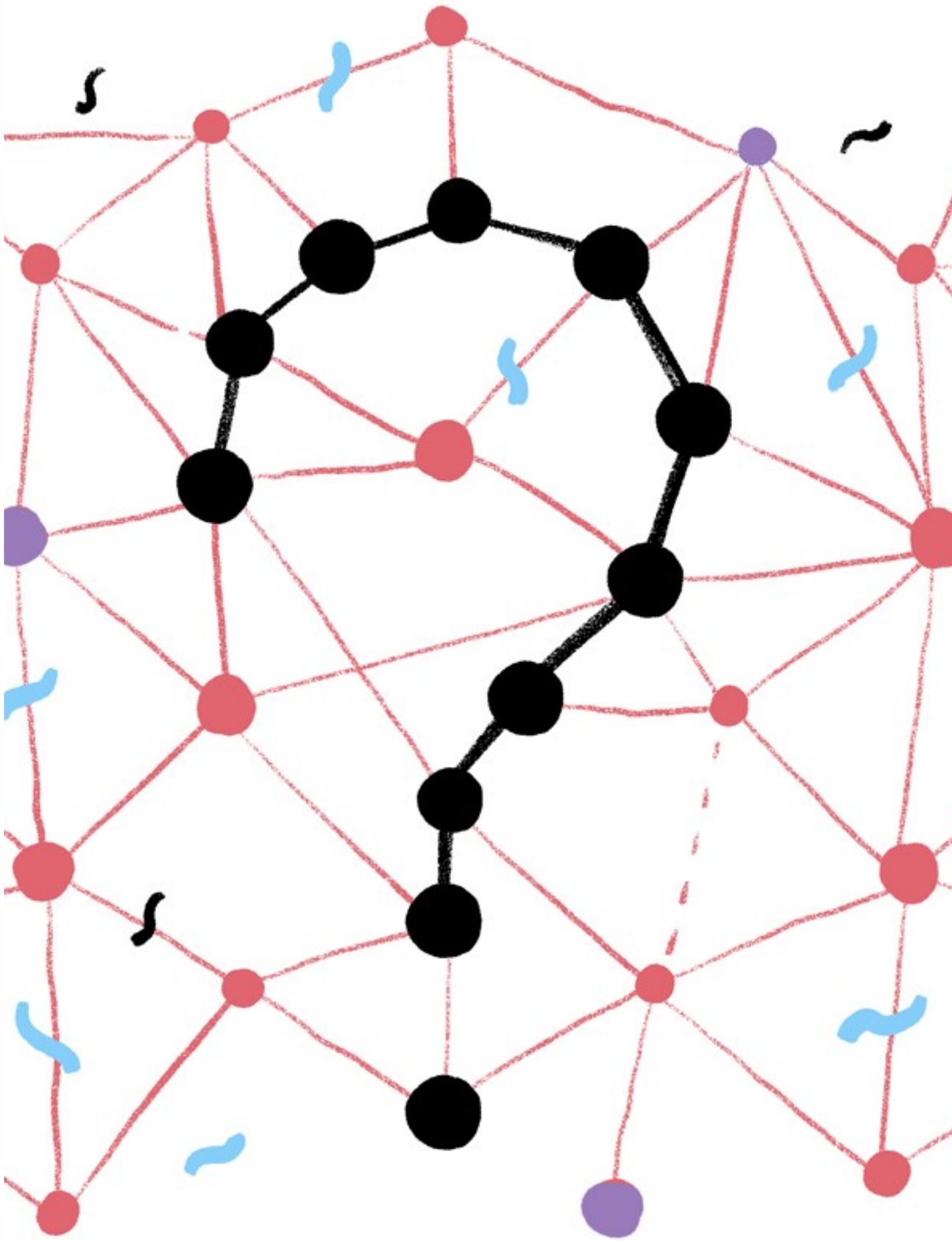
Z-SCORE $z = \frac{(X - \mu)}{\sigma}$

ALGORITHMS

HUBER REGRESSION

SUPPORT VECTOR REGRESSION

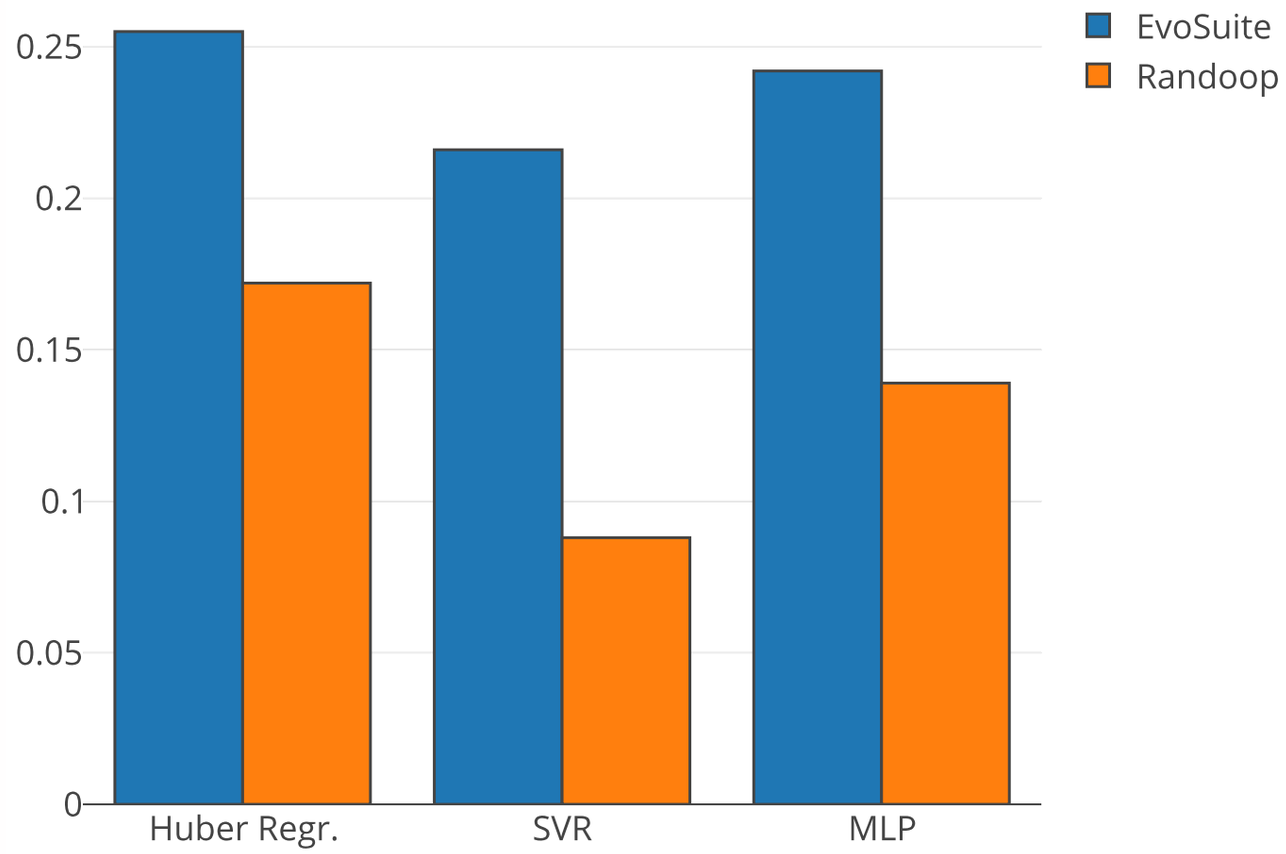
MULTI-LAYER PERCEPTION



RESULTS RQ1

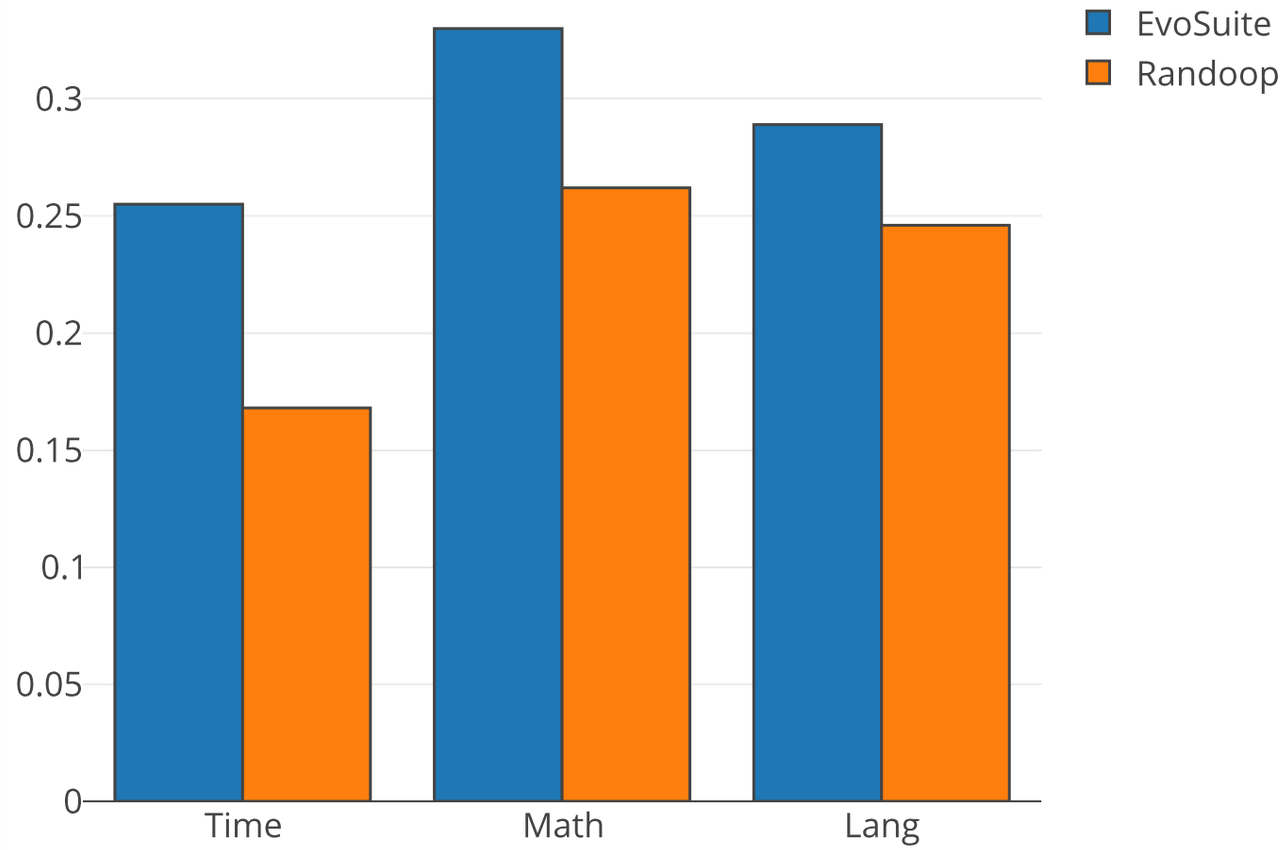
$$MAE = \frac{\sum_{i+1}^{n} |y_i - x_i|}{n}$$

	HUBER	SVR	MLP	TOOL'S AVERAGE
EVOSUITE	0.255	0.216	0.242	0.238
RANDOOP	0.172	0.088	0.139	0.131
AVERAGE	0.213	0.152	0.191	



RESULTS RQ2

	TIME	MATH	LANG	TOOL'S AVER.
EVOSUITE	0.255	0.330	0.289	0.291
RANDOOP	0.168	0.262	0.246	0.225
AVER.	0.211	0.296	0.267	



CONCLUSION

KNOWING A PRIORI THE COVERAGE ACHIEVED BY TEST DATA GENERATION TOOLS MIGHT EASE IMPORTANT DECISIONS:

- WE TOOK THE FIRST STEPS, INVESTIGATING WELL KNOWN FEATURES
- WELL KNOW FEATURES (FROM GUT FEELING) GIVE REASONABLE RESULTS

FUTURE WORK

IMPROVEMENTS

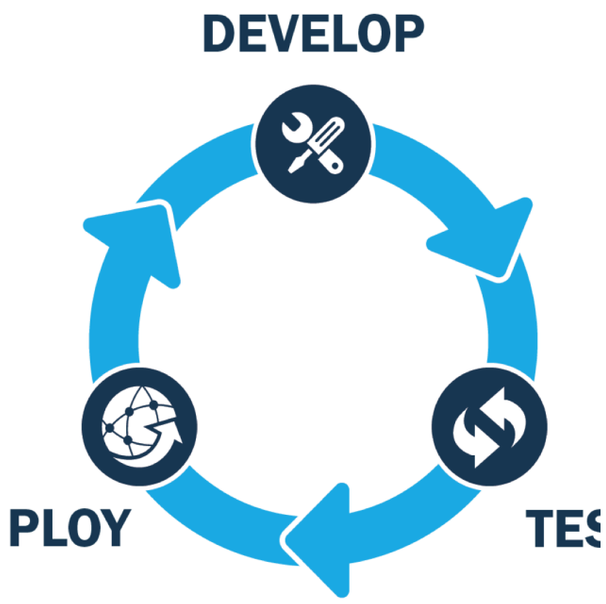
- LARGER DATASET
- BRANCH-LEVEL FEATURES
- FEATURE ANALYSIS
- DIFFERENCES BETWEEN EMPLOYED TOOLS

CONTINUOUS INTEGRATION

THE WAY WE DEVELOP AND RELEASE CODE IS RAPIDLY CHANGING

IN AN IDEAL WORLD, AT EVERY SINGLE COMMIT THE ENTIRE TEST SUITE SHOULD BE EXECUTED ...

... BUT COMPANIES LIKE GOOGLE COMMIT ABOUT 16.000 CHANGES PER DAY!

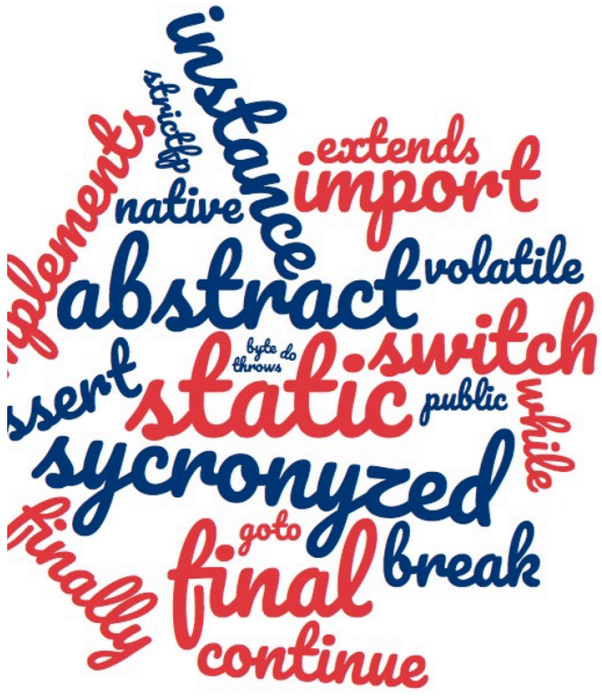


JAVA RESERVED KEYWORDS

TO CAPTURE ADDITIONAL COMPLEXITY

PREVIOUSLY USED IN INFORMATION RETRIEVAL AS A FEATURE [6]

52 JAVA RESERVED KEYWORDS



[6] SANDERSON ET AL. THE HISTORY OF INFORMATION RETRIEVAL RESEARCH

PROJECT SELECTION

OPEN SOURCE PROJECTS FROM DEFECT4J

- APACHE CASSANDRA
- APACHE IVY
- GOOGLE GUAVA
- GOOGLE DAGGER

	GUAVA	CASSANDRA	DAGGER	IVY
LOC	78,525	220,573	848	50,430
JAVA FILES	538	1,474	43	464

RESULTS RQ2

	TIME	MATH	LANG	TOOL'S AVER.
EVOSUITE	0.255	0.330	0.289	0.291
RANDOOP	0.168	0.262	0.246	0.225
AVER.	0.211	0.296	0.267