Using Deep Learning to Identify Technical Debt

Levereging Self Admitted Technical Debt

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Major in Artificial Intelligence

presented by
Simone Giacomelli

under the supervision of
Prof. Dr. Gabriele Bavota
co-supervised by
Dr. Csaba Nagy

September 2020

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Simone Giacomelli
Lugano, 1 September 2020

Dedication

Abstract

Composed of four parts:
a: context
b: problem
c: proposed solution
d: results

max 400 words. I will write it once the intro has been approved

Acknowledgements

Ack

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Technical Debt (TD) is a metaphor between the financial concept and software development. A TD is contracted when a workaround or shortcut is taken during code implementation. Choosing an easy and quick solution over a slower and more correct one gives the benefit to save time and deliver the artifact faster.
The downsides of incurring in TD are many, one of the most intuitive is that further work on the affected parts will be more expensive and time consuming than on clean and healthy code. There are also indirect effects to contemplate: the software could misbehave in the domain it is operating, causing costs and damages as the effect of unexpected output or wrong behavior [**tom2013exploration**].
TD do not only appear in software projects but can also be found in many layers of a technology stack: for example, delaying an hardware upgrade or a maintenance can give an immediate benefit of less downtime or financial savings, but an increased cost for future unexpected downtime or failures [**allman2012managing**].
Developers or managers can choose to incur in TD because of strict deadline, limited resources available or just plain laziness [**hinsen2015technical**, **allman2012managing**]. Cunningham, who coined the technical debt metaphor, writes in "The WyCash portfolio management system" [**cunningham1992wycash**]: "A little debt speeds development so long as it is paid back promptly with a rewrite". Cunningham implies that one could benefit from a small amount of TD but its should be paid back as soon as possible.
TD can arise from intentional and unintentional decision, e.g. an inexperienced person could contract it without being aware of it [**hinsen2015technical**]; In both cases it's often done for saving the limited available resources and shortening time-to-market [**tom2012consolidated**]. For example, startups are highly pressured to quickly test products and ideas in order to save capitals and be faster than competitors. Besker at al. studied how startups incur in TD, which are the factors, challenges and benefits of intentional acquisition of TD; two of the regulating factors found by the authors are the experience of the developers and the software knowledge of the founders [**besker2018embracing**].

In contrast to the beneficial viewpoint of TD, Ron Jeffries argues that the metaphor could be "perhaps too gentle", because it highlights the wise aspect of the choice of contracting a debt; the problem is that people also takes debt unwisely. Technical debt benefits a software project as long as it is handled before the bigger long term cost is realized [**guo2016exploring**].
******
TD has attributes that help us to create a model In order to better analyze and create a tractable model of TD, Alves et al. identifies three variables [**martini2018technical**]:

Identified variables (3) Tracking and managing is important Few proposed tracking and managing examples

there is a cost paying the debt and there are interest costs when efforts are wasted coping with non optimal code.

Why detecting it is important
Tom et al. in their systematic literature review studied, among others, the benefits and drawbacks of incurring in TD [**tom2012consolidated**]; the part we are now interested in is the drawbacks:
Increasing costs over time, such as the amount of effort required to deliver a certain amount of functionality
Work estimation becomes difficult
Developer productivity is negatively impacted
Becomes increasingly difficult to repay as decisions are affected by existing debt
Increased risk involved in modifications to the system
Change becomes prohibitively expensive to the point of bankruptcy, and a complete rewrite and new platform may become necessary
Decreased quality in the end product

Ci sono team che risolvo as-you-go, altri hanno strategie per rilevare, identificare e fixare. It has been studied that TD [Investigating the Impact of Design Debt on Software Quality] E' importante rilevare e monitorare i TD perche' questi aumentano il costo di un progetto software, costo dovuto dall'interesse che si deve pagare in relazione al TD.

1.1 Objectives and Results

Once the context is clear, I will explain the goal of the thesis and summarize the achieved results The goal of the thesis is to exploit SATD to train a model that can acquire the capacity to distinguish between TD-free code and code affected by TD. Using the comments in open source projects I will identify class methods noted as SATD. Through the vcs commit history I will identify when this comments disappear; the assumption here is that when the comment is removed from the code the SATD has been fixed. Many cases are excluded to minimize the probability of keeping a false positive, i.e. the SATD is not fixed but the comment is removed. For example, the simplest case excluded is when the code is exactly the same and the only change is the SATD comment removal. Another reason of exclusion is when the code is changed too much; in such case it would not be prudent to keep the sample in the dataset.

The achieved results tell us that it is difficult to predict with high accuracy on methods with big bodies. As the train dataset is limited to shorter and shorter method size, the accuracy grows.

1.2 Structure of the Thesis

A simple bullet list saying "Chapter 2 presents the state of the art, bla... Chapter 3 ..."

- Chapter 2 presents the state of the art
- Chapter 3 explains how to leverage SATD to train a Deep Learning model to detect TD
- Chapter 4 Empirical Study design
- Chapter 5 Results discussion
- Chapter 6 Threats to validity
- Chapter 7 Conclusion

Chapter 2

State of the Art

The following chapters describe existing approaches to detect different types of technical debt

2.1 Automatic Identification of Software Bugs

Classification of bug detection tools.
The following table needs to be customized

	Static	**Dynamic**	**Model Checking**
Programming-rule based tools	PREfix [2] RacerX [4]	Purify [12] Valgrind [25]	VeriSoft[9] JPFinder[13]
Statistic-rule based tools	CP-Miner [18] D. Engler's [5]	DIDUCE [10] AccMon [32] Liblit's [19]	CMC[21]
Annotation-based	ESC/Java [8]		

PREfix is a source code analyzer that detects a broad range of errors in C and C++ code [**bush2000static**]. Its goal is to detect many runtime issues on real world programs, without dynamic analysis and instrumentation; only the source code text is used. PREfix can detect defects efficiently through a model that abstracts functions and their interactions; the analyzer traces execution paths handling multiple language features: pointers, arrays, structs, bit field operations, control flows statements and so on. The method used is based on the simulation of individual functions; it uses a virtual machine that simulates the actions of each operator and function call. With the detailed tracking it can report defects information to the user so to easily characterize the detected error. The tool can be applied both to a complete program source or only a subset. This

bottom up approach is particularly useful when the source code is not fully available (e.g. in the case of a thirty part library).

RacerX deals with complex multithread systems. It detects race conditions and deadlock using static analysis [**engler2003racerx**]. The tool can infer from the source code the object lock that is assigned to a particular code block. It detects code that is multithreaded and the code that apply dangerous shared access. RaceX uses annotations only to mark the code that deals with lock acquisition; this requirement keeps the burden on the user to the minimum to increase ease of adoption. The authors of RaceX report their experiences on the biggest problem about race detection: in large codebase there are massive amounts of unprotected variable access; the key point is to report only those that can actually cause problems. There is emphasis on two aspects. The first is to minimize the impact of reporting false positive in order to avoid the users to discard the use of the tool; to achieve this, RaceX employs specific techniques to lower the impact of analysis mistakes. The second is the speed of the tool: the authors keep the time of execution for the analysis under deep scrutiny; they claim that for a codebase of 1.8 LOC the time required is between 2-14 minutes. The tool has been found capable of finding sever problems in huge projects like Linux, FreeBSD and a large closed source commercial software.

Purify is a dynamic analysis tool for software testing and quality assurance [**hastings1992fast**]. It instruments the object code files generated from compilers (the software dates back to 1992 and the supported platform is Sun Microsystem's SPARC); the target object files include also third-party libraries. Purify detects multiple errors: memory leaks, access error, reading uninitialized memory. The injected instructions check every read and write memory operations; the slow down of the target is under three times in respect to the non-instrumented execution time. The authors emphasizes: the ease of use in the classic makefile development toolchain and the low overhead that allows the developer to detect bugs early on in the development cycle.

Valgrind [25] todo: this is copy&paste of the website description and wikipedia Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

[wikipedia] Valgrind is in essence a virtual machine using just-in-time (JIT) compilation techniques, including dynamic recompilation. Nothing from the original program ever gets run directly on the host processor. Instead, Valgrind first translates the program into a temporary, simpler form called Intermediate Representation (IR), which is a processor-neutral, SSA-based form. After the conversion, a tool (see below) is free to

do whatever transformations it would like on the IR, before Valgrind translates the IR back into machine code and lets the host processor run it. Valgrind recompiles binary code to run on host and target (or simulated) CPUs of the same architecture. It also includes a GDB stub to allow debugging of the target program as it runs in Valgrind, with "monitor commands" that allow you to query the Valgrind tool for various sorts of information.

A considerable amount of performance is lost in these transformations (and usually, the code the tool inserts); usually, code run with Valgrind and the "none" tool (which does nothing to the IR) runs at 1/4 to 1/5 of the speed of the normal program.

CP-Miner [18] todo: this is copy&paste of the paper abstract Copy-pasted code is very common in large software be- cause programmers prefer reusing code via copy-paste in order to reduce programming effort. Recent studies show that copy-paste is prone to introducing bugs and a sig- nificant portion of operating system bugs concentrate in copy-pasted code. Unfortunately, it is challenging to ef- ficiently identify copy-pasted code in large software. Ex- isting copy-paste detection tools are either not scalable to large software, or cannot handle small modifications in copy-pasted code. Furthermore, few tools are available to detect copy-paste related bugs. In this paper we propose a tool, CP-Miner, that uses data mining techniques to efficiently identify copy-pasted code in large software including operating systems, and detects copy-paste related bugs. Specifically, it takes less than 20 minutes for CP-Miner to identify 190,000 copy- pasted segments in Linux and 150,000 in FreeBSD. More- over, CP-Miner has detected 28 copy-paste related bugs in the latest version of Linux and 23 in FreeBSD. In addition, we analyze some interesting characteristics of copy-paste in Linux and FreeBSD, including the distribution of copy- pasted code across different length, granularity, modules, degrees of modification, and various software versions.

D. Engler's [5] todo: this is copy&paste of the paper abstract A major obstacle to finding program errors in a real system is knowing what correctness rules the system must obey. These rules are often undocumented or specified in an ad hoc manner. This paper demonstrates techniques that automatically extract such checking information from the source code itself, rather than the programmer, thereby avoiding the need for a priori knowledge of system rules.The cornerstone of our approach is inferring programmer "beliefs" that we then cross-check for contradictions. Beliefs are facts implied by code: a dereference of a pointer, p, implies a belief that p is non-null, a call to "unlock(1)" implies that 1 was locked, etc. For beliefs we know the programmer must hold, such as the pointer dereference above, we immediately flag contradictions as errors. For beliefs that the programmer may hold, we can assume these beliefs hold and use a statistical analysis to rank the resulting errors from most to least likely. For example, a call to "spin_lock" followed once by a call to "spin_unlock" implies that the programmer may have paired these calls by coincidence. If the pairing happens 999 out

of 1000 times, though, then it is probably a valid belief and the sole deviation a probable error. The key feature of this approach is that it requires no a priori knowledge of truth: if two beliefs contradict, we know that one is an error without knowing what the correct belief is.Conceptually, our checkers extract beliefs by tailoring rule "templates" to a system — for example, finding all functions that fit the rule template "a must be paired with b." We have developed six checkers that follow this conceptual framework. They find hundreds of bugs in real systems such as Linux and OpenBSD. From our experience, they give a dramatic reduction in the manual effort needed to check a large system. Compared to our previous work [9], these template checkers find ten to one hundred times more rule instances and derive properties we found impractical to specify manually.

DIDUCE [10] todo: this is copy&paste of the paper abstract DIDUCE is a practical and effective tool that aids programmers in detecting complex program errors and identifying their root causes. By instrumenting a program and observing its behavior as it runs, DIDUCE dynamically formulates hypotheses of invariants obeyed by the program. DIDUCE hypothesizes the strictest invariants at the beginning, and gradually relaxes the hypothesis as violations are detected to allow for new behavior. The violations reported help users to catch software bugs as soon as they occur. They also give programmers new visibility into the behavior of the programs such as identifying rare corner cases in the program logic or even locating hidden errors that corrupt the program's results.We implemented the DIDUCE system for Java programs and applied it to four programs of significant size and complexity. DIDUCE succeeded in identifying the root causes of programming errors in each of the programs quickly and automatically. In particular, DIDUCE is effective in isolating a timing-dependent bug in a released JSSE (Java Secure Socket Extension) library, which would have taken an experienced programmer days to find. Our experience suggests that detecting and checking program invariants dynamically is a simple and effective methodology for debugging many different kinds of program errors across a wide variety of application domains.

AccMon [32] todo: this is copy&paste of the paper abstract

This paper makes two contributions to architectural support for software debugging. First, it proposes a novel statistics-based, on- the-fly bug detection method called PC-based invariant detection. The idea is based on the observation that, in most programs, a given memory location is typically accessed by only a few instructions. Therefore, by capturing the invariant of the set of PCs that nor- mally access a given variable, we can detect accesses by outlier instructions, which are often caused by memory corruption, buffer overflow, stack smashing or other memory-related bugs. Since this method is statistics-based, it can detect bugs that do not violate any programming rules and that, therefore, are likely to be missed by many existing tools. The second contribution is a novel architec- tural extension called the Check Look-aside Buffer (CLB).

The CLB uses a Bloom filter to reduce monitoring overheads in the recently- proposed iWatcher architectural framework for software debugging. The CLB significantly reduces the overhead of PC-based invariant debugging. We demonstrate a PC-based invariant detection tool called Ac- cMon that leverages architectural, run-time system and compiler support. Our experimental results with seven buggy applications and a total of ten bugs, show that AccMon can detect all ten bugs with few false alarms (0 for five applications and 2-8 for two appli- cations) and with low overhead (0.24-2.88 times). Several existing tools evaluated, including Purify, CCured and value-based invariant detection tools, fail to detect some of the bugs. In addition, Purify's overhead is one order of magnitude higher than AccMon's. Finally, we show that the CLB is very effective at reducing overhead.

Liblit's [19] todo: this is copy&paste of the paper abstract We propose a low-overhead sampling infrastructure for gath- ering information from the executions experienced by a pro- gram's user community. Several example applications illus- trate ways to use sampled instrumentation to isolate bugs. Assertion-dense code can be transformed to share the cost of assertions among many users. Lacking assertions, broad guesses can be made about predicates that predict program errors and a process of elimination used to whittle these down to the true bug. Finally, even for non-deterministic bugs such as memory corruption, statistical modeling based on logistic regression allows us to identify program behaviors that are strongly correlated with failure and are therefore likely places to look for the error.

ESC/Java [8] todo: this is copy&paste of the paper abstract Software development and maintenance are costly endeavors. The cost can be reduced if more software defects are detected earlier in the development cycle. This paper introduces the Extended Static Checker for Java (ESC/Java), an experimental compile-time program checker that finds common programming errors. The checker is powered by verification-condition generation and automatic theoremproving techniques. It provides programmers with a simple annotation language with which programmer design decisions can be expressed formally. ESC/Java examines the annotated software and warns of inconsistencies between the design decisions recorded in the annotations and the actual code, and also warns of potential runtime errors in the code. This paper gives an overview of the checker architecture and annotation language and describes our experience applying the checker to tens of thousands of lines of Java programs.

VeriSoft [9] todo: this is copy&paste of the paper abstract VeriSoft is a tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary code written in full-fledged programming languages such as C or C++. It can automatically detect coordination problems between concurrent processes. Specifically, VeriSoft searches the state space of the system for deadlocks,

livelocks, divergences, and violations of user-specified assertions. An interactive graphical simulator/debugger is also available for following the execution of all the processes of the concurrent system.
https://link.springer.com/chapter/10.1007/3-540-63166-6_52

JPFinder [13] todo: this is copy&paste of the paper abstract Jpf, is a prototype translator from Java to Promela, the modeling language of the Spin model checker [4]. Jpf is a product of a major effort by the Automated Software Engineering group at NASA Ames to make model checking technology part of the software process. Experience has shown that severe bugs can be found in final code using this technique [1], and that automated translation from a programming language to a modeling language like Promela can help reducing the effort required. Jpf allows a programmer to annotate his Java program with assertions and verify them using the Spin model checker. In addition, deadlocks can be identi- fied. An assertion is written as a call to an assert method defined in a predefined Java class, the Verify class. The argument to the method is a boolean Java expression over the state variables. The Verify class contains additional tem- poral logic methods which allow to state temporal logic properties about static variables. Hence Java itself is used as the specification language. An application of Jpf is described elsewhere in the proceedings [3]. A respectable subset of Java is covered by Jpf, including dynamic object cre- ation, object references as first class citizens, inheritance, exceptions, interrupts, and perhaps most importantly: thread operations. Among major concepts not translated are: packages, method overloading and overriding, method recursion, strings, and floating point numbers. Finally, the class library is not translated

CMC [21] todo: this is copy&paste of the paper abstract Many system errors do not emerge unless some intricate sequence of events occurs. In practice, this means that most systems have errors that only trigger after days or weeks of execution. Model checking [4] is an effective way to find such subtle errors. It takes a simplified description of the code and exhaustively tests it on all inputs, using techniques to explore vast state spaces efficiently. Unfortunately, while model checking systems code would be wonderful, it is almost never done in practice: building models is just too hard. It can take significantly more time to write a model than it did to write the code. Furthermore, by checking an abstraction of the code rather than the code itself, it is easy to miss errors. The paper's first contribution is a new model checker, CMC, which checks C and C++ implementations directly, eliminating the need for a separate abstract description of the system behavior. This has two major advantages: it reduces the effort to use model checking, and it reduces missed errors as well as time-wasting false error reports resulting from inconsistencies between the abstract description and the actual implementation. In addition, changes in the implementation can be checked immediately without updating a high-level description.

The paper's second contribution is demonstrating that CMC works well on real code by applying it to three implementations of the Ad-hoc On-demand Distance Vector (AODV) networking protocol [7]. We found 34 distinct errors (roughly one bug per 328 lines of code), including a bug in the AODV specification itself. Given our experience building systems, it appears that the approach will work well in other contexts, and especially well for other networking protocols.

2.2 Code Smells and Antipatterns

copy&paste Several techniques have been proposed in the literature to detect code smell instances affecting code components, and all of these take their cue from the suggestions provided by four well-known books: [29], [16], [86], [66].

2.2.1 Four inspirational books description

copy&paste
Webster [86] - todo The first one, by Webster [**webster1995pitfalls**] defines common pitfalls in Object Oriented Development, going from the project management down to the implementation.

Riel [66] - todo Riel [66] describes more than 60 guidelines to rate the integrity of a software design

Fowler [29] - todo The third one, by Fowler [29], describes 22 code smells describing for each of them the refactoring actions to take.

Brown et al. [16] - todo Brown et al. [16] define 40 code antipatterns of different nature (i.e., architectural, managerial, and in source code), together with heuristics to detect them.

2.2.2 Last decade proposed approaches

From these starting points, in the last decade several approaches have been proposed to detect design flaws in source code - todo fix copy&paste

Travassos et al. [travassos1999detecting] Travassos et al. created a set of techniques for manually identifying defects in order to improve software quality. These practices help individuals to read object oriented code and assess, through a predefined taxonomy, if some problem is present. The authors conducted an empirical study

on these techniques and report their feasibility.

van Emden and Moonen [83] todo, fix copy&paste van Emden and Moonen [83] presented jCOSMO, a code smell browser that visualizes the detected smells in the source code. In particular, they focus their attention on two Java programming smells, known as instanceof and typecast. The first occurs when there are too many instanceof operators in the same block of code that make the source code difficult to read and understand. The typecast smell appears instead when an object is explicitly converted from one class type into another, possibly performing illegal casting which results in a runtime error.

Simon et al. [] todo, fix copy&paste Simon et al. [72] provided a metric-based visualization tool able to discover design defects repre- senting refactoring opportunities. For example, a Blob is detected if different sets of cohesive attributes and methods are present inside a class. In other words, a Blob is identified when there is the possibility to ap- ply Extract Class refactoring.

Marinescu [50] todo, fix copy&paste Marinescu [50] proposed a metric-based mechanism to capture deviations from good design principles and heuristics, called "detection strategies". Such strategies are based on the identifica- tion of symptoms characterizing a particular smell and metrics for measuring such symptoms. Then, thresholds on these metrics are defined in order to define the rules

Lanza and Marinescu [44] todo, fix copy&paste Lanza and Marinescu [44] showed how to exploit quality metrics to identify "disharmony patterns" in code by defining a set of thresholds based on the measurement of the exploited metrics in real software systems. Their de- tection strategies are formulated in four steps. In the first step, the symptoms characterizing a smell are defined. In the second step, a proper set of metrics measuring these symptoms is identified. Having this information, the next step is to define thresholds to classify the class as affected (or not) by the defined symptoms. Finally, AND/OR operators are used to correlate the symptoms, leading to the final rules for detecting the smells.

Munro [53] todo, fix copy&paste Munro [53] presented a metric-based detection tech- nique able to identify instances of two smells, i.e., Lazy Class and Temporary Field, in the source code. A set of thresholds is applied to some structural metrics able to capture those smells. In the case of Lazy Class, the metrics used for the identification are Number of Meth- ods (NOM), LOC, Weighted Methods per Class (WMC), and Coupling Between Objects (CBO).

Moha et al. [51] todo, fix copy&paste Moha et al. [51] introduced DECOR, a tech-

nique for specifying and de- tecting code and design smells. DECOR uses a Domain-Specific Language (DSL) for specifying smells using high-level abstractions. Four design smells are identified by DECOR, namely Blob, Swiss Army Knife, Functional Decomposition, and Spaghetti Code.

Tsantalis and Chatzigeorgiou [79] todo, fix copy&paste Tsantalis and Chatzigeorgiou [79] presented JDeodor- ant, a tool able to detect instances of Feature Envy smells with the aim of suggesting move method refactoring op- portunities. For each method of the system, JDeodorant forms a set of candidate target classes where a method should be moved. This set is obtained by examining the entities (i.e., attributes and methods) that a method accesses from the other classes. In its current version JDeodorant8 is also able to detect other three code smells (i.e., State Checking, Long Method, and God Classes), as well as opportunities for refactoring code clones.

Ligu et al. [47] todo, fix copy&paste Ligu et al. [47] introduced the identification of Refused Bequest code smell using a combination of static source code analysis and dynamic unit test execution. Their approach aims at discovering classes that really want to support the interface of the superclass [29]. In order to understand what are the methods really invoked on subclass instances, they intentionally override these methods introducing an error in the new implementation (e.g., division by zero). If there are classes in the system invoking the method, then a failure will occur. Otherwise, the method is never invoked and an instance of Refused Bequest is found.

2.2.3 Code smell detection formulated as an optimization problem:

Kessentini et al. [36] todo, fix copy&paste Kessentini et al. [36] as they presented a technique to detect design de- fects by following the assumption that what significantly diverges from good design practices is likely to represent a design problem. The advantage of their approach is that it does not look for specific code smells (as most ap- proaches) but for design problems in general. Also, in the reported evaluation, the approach was able to achieve a 95% precision in identifying design defects [36].

Kessentini et al. [37] todo, fix copy&paste Kessentini et al. [37] also presented a cooperative parallel search- based approach for identifying code smells instances with an accuracy higher than 85%.

Boussaa et al. [13] todo, fix copy&paste Boussaa et al. [13] proposed the use of competitive coevolutionary search to code-smell detection problem. In their approach two populations evolve simultaneously: the first generates detection rules with the aim of detecting the highest possible proportion of code smells, whereas the second

population generates smells that are currently not detected by the rules of the other population.

Sahin et al. [68] todo, fix copy&paste Sahin et al. [68] proposed an approach able to generate code smell detection rules using a bi-level optimization problem, in which the first level of optimization task creates a set of detection rules that maximizes the coverage of code smell examples and artificial code smells generated by the second level. The lower level is instead responsible to maximize the number of code smells artificially generated. The empirical evaluation shows that this approach achieves an average of more than 85% in terms of precision and recall.

2.2.4 Non binary classification (smell/clean)

The approaches described above classify classes strictly as being clean or anti-patterns, while an accurate analysis for the borderline classes is missing [40]

Khomh et al. [40] todo, fix copy&paste Khomh et al. [40] proposed an approach based on Bayesian belief networks providing a likelihood that a code component is affected by a smell, instead of a boolean value as done by the previous techniques.

Oliveto et al. [58] todo, fix copy&paste This is also one of the main characteristics of the approach based on the quality metrics and B-splines proposed by Oliveto et al. [58] for identifying instances of Blobs in source code

2.2.5 Usage of historical data for code smells

Ratiu et al. [65] todo, fix copy&paste Ratiu et al. [65] proposed to use the historical information of the suspected flawed structure to increase the accuracy of the automatic problem detection.

Palomba et al. [60] todo, fix copy&paste Palomba et al. [60] provided evidence that historical data can be successfully exploited to identify not only smells that are intrinsically characterized by their evolution across the program history – such as Divergent Change, Parallel Inheritance, and Shotgun Surgery – but also smells such as Blob and Feature Envy [60].

2.3 TD and machine learning

–this section will reference the paper from CSABA (techdebt conference 2020 related paper)–
(c&p) Several detection techniques and tools have been proposed in the literature. Recently, the adoption of machine learning techniques to detect bad smells became a trend [20]

Khom et al. [27] and [28] Khomh et al. proposed a Bayesian approach which initially converts existing detection rules to a probabilistic model to perform the predictions [27]. Khom et al. [28] extend [27] by the introduction of Bayesian Belief Networks, improving the accuracy of the detection.

Maiga et al. Maiga et al. proposed an SVM-based approach that uses the feedback information provided by practitioners [35, 36]

Amorim et al. [3] Amorim et al. [3] presented an experience report on the effectiveness of Decision Trees for detecting bad smells. They choose these classifiers due to their interpretability [3]. Thus, most of the proposed works focus on only one classifier. They were also trained in a dataset composed of few systems and, consequently, the results may be positive towards their aproach due to overfitting.

Fontana et al. [21] Fontana et al. evaluated different machine learning algorithms on a set of different systems [21]. Their work was later extended and refined, providing a larger comparison of classifiers [20]. The notorious impact of this work was the incredible performance re- ported. Even naive algorithms were able of achieving great results using a small training dataset. This draws attention to possible drawbacks and limitations of their work, which was later reported by Di Nucci et al. [15]

Di Nucci et al. [15] Di Nucci et al. [15]. They replicated the study and verified that the reported performance was highly biased by the dataset and the procedures adopted, such as unrealistic balanced dataset, in which one third of the instances were smelly.

Daniel Cruz at al. –the paper itself- Detecting Bad Smells with Machine Learning Algorithms: an Empirical Study. Bad smells are symptoms of bad design choices implemented on the source code. They are one of the key indicators of technical debts, specifically, design debt. To manage this kind of debt, it is important to be aware of bad smells and refactor them whenever possible. Therefore, several bad smell detection tools and techniques have been proposed over the years. These tools and techniques present different strategies to perform detections. More recently, machine learning al-

gorithms have also been proposed to support bad smell detection. However, we lack empirical evidence on the accuracy and efficiency of these machine learning based techniques. In this paper, we present an evaluation of seven different machine learning algorithms on the task of detecting four types of bad smells. We also provide an analysis of the impact of software metrics for bad smell detection using a unified approach for interpreting the models' decisions. We found that with the right optimization, machine learning algorithms can achieve good performance (F1 score) for two bad smells: God Class (0.86) and Refused Parent Bequest (0.67). We also uncovered which metrics play fundamental roles for detecting each bad smell.

2.4 Self-Admitted Technical Debt (WAS 2.3)

Potdar and Shihab [30] Potdar and Shihab [30] pioneered the study of SATD by mining five software systems to investigate (i) the amount of SATD they contain, (ii) the factors promoting the intro- duction of the SATD, and (iii) how likely is the SATD to be removed after its introduction.

Storey et al. [35] Storey et al. [35] explored the use of task annotations (e.g., TODO, FIXME) in code comments as a mechanism to manage developers tasks. Their findings highlight several activities that are supported by such annotations (e.g., TODOs are sometimes used to communicate by asking questions to other developers).

Guo et al. [18] Guo et al. [18] tracked the lifecycle of a single delayed maintenance task (i.e., a technical debt instance) to study the effect of this decision on the project outcomes. Their data confirm the harmfulness of technical debt, showing that the decision to delay the task resulted in tripled costs for its implementation.

Klinger et al. [21] Klinger et al. [21] interviewed four technical architects at IBM to understand how decisions to acquire technical debt are made within an enterprise. They found that technical debt is often acquired due to non-technical stakeholders (e.g., due to imposed requirement to meet a specific deadline sacrificing quality). Also, they highlight the lack of effec- tive communication between technical and non-technical stakeholders involved in the technical debt management

Kruchten et al. [23] Kruchten et al. [23] built on top of the technical debt definition provided by Cunningham [14] to clearly define what constitute technical debt and provide some theoretical foundations. They presented the "technical debt landscape", classifying the technical debt as visible (e.g., new features to add) or invisible (e.g., high code complexity) and highlighting the debt types mainly causing issues to the evolv-

ability of software (e.g., new features to add) or to its maintain- ability (e.g., defects).

Lim et al. [25] Lim et al. [25] interviewed 35 practitioners to investigate their perspective on technical debt. They found that most of participants were familiar with the notion of technical debt—"We live with it every day" [25]—and they do not look at technical debt as a poor programming practice, but more as an intentional decision to trade off competing concerns during development [25]. Also, practitioners acknowledged the difficulty in measuring the impact of technical debt on software projects. Similarly, Kruchten et al. [24] reported their understanding of the technical debt in industry as the result of a four year interaction with practitioners.
Kruchten et al. [24] Kruchten et al. [24] reported their understanding of the technical debt in industry as the result of a four year interaction with practitioners.

Zazworka et al. [41] Zazworka et al. [41] compared how technical debt is detected manually by developers and automatically by de- tection tools. The achieved results show very little overlap between the technical debt instances manually and auto- matically detected.

Spinola et al. [34] Spinola et al. [34] collected a set of 14 statements about technical debt from the literature (e.g., "The root cause of most technical debt is pressure from the customer" [32]) and asked 37 practitioners to express their level of agreement for each statement. The statement achieving the highest agreement was "If technical debt is not managed effectively, maintenance costs will increase at a rate that will eventually outrun the value it delivers to customers".

Alves et al. [2] Alves et al. [2] proposed an ontology of terms on technical debt. Part of this ontology is the classification of technical debt types derived by studying the definitions existing in the literature.

Maldonado and Shihab [15] Maldonado and Shihab [15] also used the classification by Alves et al. to investigate the types of SATD more diffused in open source projects. They identified 33K comments in five software systems reporting SATD. These comments have been manually read by one of the authors who found as the vast majority of them (∼60%) reported design debt.

Wehaibi et al. [37] Wehaibi et al. [37] analysed the relation between SATD and software quality in five open source systems. Their main findings show that: (i) the defect-proneness of files containing SATD instances increases after their introduction; and (ii) developers experience difficulties in performing SATD-related changes

2.5 Summing Up (WAS 2.4)

Here I explain why what I did is different

Chapter 3

Using Deep Learning to Detect Technical Debt

Here I describe the approach.

I'm going to start from a short overview, then I go into details

3.1 Mining SATD Instances and their Fixes

3.2 The Deep Learning Model

3.3 Hyperparameter Tuning

Chapter 4

Empirical Study Design

I'm going to start with a short description about the goal/research questions

4.1 Context Selection

I'll explain the dataset used for the evaluation

4.2 Data Collection and Analysis

How I compute the results

4.3 Replication Package

A link to a repo with all data and code

Chapter 5

Results Discussion

. . . results discussion . . .

5.1 Quantitative Results

Report precision/recall for different confidence thresholds

5.2 Qualitative Results

Discuss interesting cases in which your approach succeeds/fails

Chapter 6

Threats to Validity

. . . Discussion on the limitations of my study . . .

Chapter 7

Conclusion

. . . conclusion . . .