MLSmellHound

A Context-Aware Code Analysis Tool

Kannan, Jai; Barnett, Scott; Cruz, Luís; Simmons, Anj; Agarwal, Akash

DOI
10.1109/ICSE-NIER55298.2022.9793510

Publication date
2022

Document Version
Final published version

Published in
Proceedings - 2022 ACM/IEEE 44th International Conference on Software Engineering

Citation (APA)
Kannan, J., Barnett, S., Cruz, L., Simmons, A., & Agarwal, A. (2022). MLSmellHound: A Context-Aware Code Analysis Tool. In *Proceedings - 2022 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER 2022* (pp. 66-70). (Proceedings - International Conference on Software Engineering). IEEE. https://doi.org/10.1109/ICSE-NIER55298.2022.9793510

2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) | 978-1-6654-9596-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSE-NIER55298.2022.9793510

MLSmellHound: A Context-Aware Code Analysis Tool

Jai Kannan
Scott Barnett
jai.kannan@deakin.edu.au
scott.barnett@deakin.edu.au
Applied Artificial Intelligence Inst.
Deakin University
Geelong, Victoria, Australia

Luís Cruz
l.cruz@tudelft.nl
Delft University of Technology
Delft, Netherlands

Anj Simmons
Akash Agarwal
a.simmons@deakin.edu.au
a.agarwal@deakin.edu.au
Applied Artificial Intelligence Inst.
Deakin University
Geelong, Victoria, Australia

ABSTRACT

Meeting the rise of industry demand to incorporate machine learning (ML) components into software systems requires interdisciplinary teams contributing to a shared code base. To maintain consistency, reduce defects and ensure maintainability, developers use code analysis tools to aid them in identifying defects and maintaining standards. With the inclusion of machine learning, tools must account for the cultural differences within the teams which manifests as multiple programming languages, and conflicting definitions and objectives. Existing tools fail to identify these cultural differences and are geared towards software engineering which reduces their adoption in ML projects. In our approach we attempt to resolve this problem by exploring the use of context which includes i) purpose of the source code, ii) technical domain, iii) problem domain, iv) team norms, v) operational environment, and vi) development lifecycle stage to provide contextualised error reporting for code analysis. To demonstrate our approach, we adapt Pylint as an example and apply a set of contextual transformations to the linting results based on the domain of individual project files under analysis. This allows for contextualised and meaningful error reporting for the end user.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Machine learning, code smells, context-aware

ACM Reference Format:
Jai Kannan, Scott Barnett, Luís Cruz, Anj Simmons, and Akash Agarwal. 2022. MLSmellHound: A Context-Aware Code Analysis Tool. In *New Ideas and Emerging Results (ICSE-NIER'22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510455.3512773

1 INTRODUCTION

Code analysis tools reduce maintenance costs of a system through the early detection and prevention of defects [9, 10, 15, 19, 20, 22]. These tools also support the identification and refactoring of code smells [8, 18] and reducing ML specific technical debt. By integrating code analysis tools into CI/CD pipelines modifications to code can be tested to improve i) security [2], ii) readability [23], and iii) consistency [34].

Machine learning (ML) applications stand to benefit from code analysis by i) identifying ML specific technical debt [25], ii) catching defects in models and data and iii) mitigating ML unique failure modes [3, 16]. These unique failure modes contribute to eroded trust and reputation damage as demonstrated by Amazon's sexist recruiting tool [24] or failure of the Thistle F.C. ball tracking system [27]. ML systems also have increased maintenance costs and effort to manage infrastructure for data and model management [25]. The additional infrastructure and dependence on data increases area where defects can occur. Code analysis tools are a promising approach for improving the robustness of ML applications.

However, code analysis tools suffer from high false positive rates when analysing ML applications [30]. The impact of high false positive rates is poor adoption and wasted time. False positives are perceived as a challenge [28, 29]. Calibration is required to balance false negatives (missed defects) and false positives (no defect present). Development of ML applications utilises multiple languages and mixed teams of data scientists (specialists in development and application of ML algorithms) working alongside software engineers to integrate the algorithms with the surrounding infrastructure. As such, defects in ML applications require a combination of data science and software engineering expertise to resolve [12]. Thus, code analysis tools operating on ML applications must account for i) multiple programming languages, ii) conflicting definition and prioritisation of defects in interdisciplinary teams, iii) ML specific failure modes, and iv) provide contextualised error reporting to provide explanations in the context of ML to reduce false positives.

Existing code analysis tools for ML have focused on data linters [13], and tensor shapes [17]. Code analysis tools focus on analysing Python, a popular language for ML [26] or identifying general code smells [11]. However, existing approaches are not sufficient to provide actionable explanations for mixed development teams and ignore the technical domain of ML [4]. For example, during exploratory data analysis, code quality is sacrificed for speed of experimentation and discovery of insights; here the activity influences the definition of code quality. This paper proposes inclusion of context from the technical domain, software artefact and environment to improve the precision of code analysis tools through contextualised feedback.

ICSE-NIER'22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9224-2/22/05…$15.00
https://doi.org/10.1145/3510455.3512773

In this paper we present a research agenda for MLSmellHound, a context-aware tool that improves the precision of code analysis tools. The goal of our research is to improve the maintainability and robustness of ML applications through context-aware code analysis tools. We hypothesize that context can improve the precision of code analysis tools and propose the following research questions:

- **RQ1:** What context can be mined from ML projects?
- **RQ2:** How can context transformations be applied to prioritize ML smells?
- **RQ3:** Which software metrics can be used with context to predict ML smells?

Our novel approach is inspired by ideas from model driven engineering that uses i) a context metamodel, ii) a set of context transformations, and iii) context checkers to improve code analysis for ML.

Motivated by the work in Simmons et al. [26], we provide a demonstrator of MLSmellHound on Python coding conventions. We apply our approach to an existing code analysis tool, Pylint. Our tool uses the purpose of the source code to selectively apply and customise linting errors. Contributions arising from this paper include: **1)** an approach for leveraging **context** to reduce false positive rates in code analysis tools, and **2)** a demonstrator of **MLSmellHound** for leveraging context in coding conventions for ML applications.

2 MOTIVATION

To motivate our research we use coding conventions as an example.

Consider Dave, a data scientist, working with a software engineer, Tammy. Dave and Tammy are tasked with creating a fraud prediction model. Dave finds a research paper providing a solution that best fits the description of the problem. Dave implements the solution in Python and the widely available ML frameworks [7]. When Dave issues a pull request, the pull request is reviewed by Tammy. During the review Tammy identifies that the code violates their organisation's coding standards. Figure 1 shows an example of Dave's code.

```
true_prob = numpy.array([0.1, 0.2, 0.3, 0.4])
#true_prob /= true_prob.sum()
D = true_prob.size
legend = [str(x) for x in true_prob]

p = 1/5
pii = (1 + (D - 1) * p) / D # P(Y=i|X=i)
pij = (1 - p) / D # P(Y=j|X=i)
P = pij * numpy.ones((D, D)) + (pii - pij) * numpy.eye(D)

def gibbs_sampling(N, alpha):
    true_count = numpy.array(N * true_prob, dtype=int)
    true_count[-1] += N - true_count.sum()
```

Figure 1: Code snippet from an open-source project[1] violating Python's naming conventions

The organisation follows PEP8 coding standards, that states that variable names such as 'P' should be lowercase. When asked why Dave neglected to use the organisation's code analysis tool, Dave responds that he prefers keeping the variable names similar to the mathematical notation of the algorithm in the paper he is implementing, for example, capital letters to represent matrices, and indicates that the configuration used by the organisation ignores data science norms resulting in high false positives from his perspective. Dave and Tammy agree there is a need to adjust the organisation's standards to restore team cohesion; however, deliberating a fix diminishes productivity as i) existing linter configurations are calibrated, ii) rule sets defined and agreed upon, and iii) new scripts are implemented as existing tools cannot identify the context of ML code. To help Dave and Tammy we propose a context-aware tool that selectively applies convention rules depending on the code context.

[1] https://github.com/shuyo/iir.git

3 VISION

Our goal is to improve the maintenance aspects and robustness of ML applications by using context-aware code analysis tools. We borrow ideas from model driven engineering (MDE) [6] to compose a new category of tools for modelling context. These tools are designed to provide better tool support for interdisciplinary teams. We hypothesise that context can provide better explanations of the code artefacts and code objectives. By combining the context with ML specific code smells we aim to prevent a class of ML defects and improve the precision of code analysis tools. To realise our vision we have modelled the problem with concepts from MDE as shown in Figure 2. We use the principle of meta-modelling [5] to create the context meta-models and context transformations for the tool. Below, we pinpoint the core components of the proposed model.

3.1 Context Metamodel

RQ1. What context can be mined from ML projects?

The context metamodel represents properties that comprise the context for an ML application. We plan to investigate multiple facets of context including i) purpose of the source code, ii) technical domain [4], iii) problem domain, iv) team norms, v) operational environment, and vi) development lifecycle stage. The context metamodel will define the relationships between these concepts. The source code is given as input to the system. The system makes use of a **Parser** (c.f Figure 2) to abstract over different source code languages and repositories, and map it to contextual features; for example, extracting file level imports of known ML frameworks for that language to help determine if the incoming code is categorised as ML or non-ML code. A context metamodel is applied to annotate the source code, producing a **Project Specific Context** which provides a high level representation for informing the context-aware linter.

3.2 Context Transformations

RQ2. How can context transformations be applied to prioritise ML smells?

The context transformations enable MLSmellHound to modify the linter based on the context of a project. These transformations include:

- Subtraction - context disallows certain rules from applying on a file.

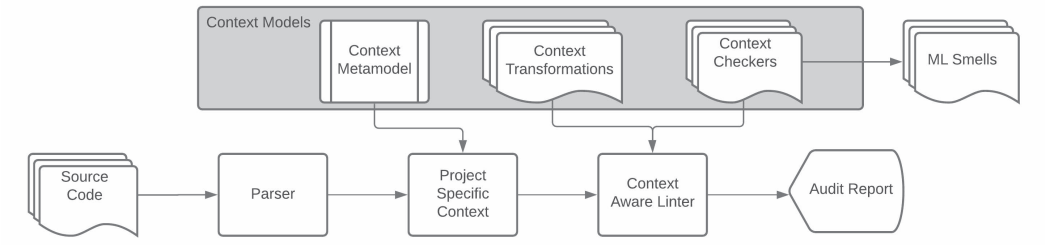

Figure 2: A conceptual overview of the novel context-aware code analysis tool, MLSmellHound.

- Addition - context selectively applies new rules to better inform the user of design problems.
- Reprioritisation - context used to re-rank linting results to draw attention to high priority items and demote items that are unlikely to be relevant in the given context.
- Remessage - operational context produces a different message to an end-user to improve usability.

Context transformations can be used with existing tools extending our approach to all code analysis tools.

3.3 Context Checkers

RQ3. Which software metrics can be used with the context to predict ML smells?

The context checkers are the rules that are used to operate on the software artefacts informed by the project specific context. These checkers are informed by ML smells that are known to cause issues and include metrics for identifying each smell. We hypothesise that the combination of context transformations and a context model will improve the precision of code analysis tools. The **Context-Aware Linter** (cf. Figure 2) is a code analyser performing two functions: i) checking for ML code smells and anti-patterns by applying the relevant set of **Context Checkers** to each file in the source code; ii) generating the context-transformed linting messages. Using these two functions the **Audit Report** is generated for the user to evaluate the code.

4 WHY IS IT NEW?

ML smells are an emerging area of research. Recent work has investigated i) the prevalence of code smells in ML applications [30], ii) deep learning [14] code smells, iii) anti-patterns [21], and linters for datasets [13]. However, a gap exists for a code analysis tool that automatically identifies ML-specific smells. Unlike existing tools, our approach also considers the technical domain [4] to provide context-aware recommendations for the code base. New tools are needed to identify suitable metrics that locate ML smells and support the developer in prioritising impact to redirect engineering effort.

Code analysis tools ignore cultural differences. Current code analysis tools are geared to support software engineering standards, however not all of these standards may be applicable for ML software, resulting in an increase in false positives and reduced adoption [29, 30]. Code analysis tools need to consider the cultural differences existing between disciplines in cross functional teams to improve support for shared development. Code quality in ML software may score poorly against traditional code quality metrics due to experimentation, mathematical vocabulary and the nature of code artefacts. To support these developers, we utilise a set of transformations based on the context of the program to selectively apply linting rules. Our tool prioritises outputs for developers, to support data scientists to build standards-compliant code without hindering their productivity.

Novel use of context for improving the precision of code analysis tools. Context has been used in software engineering to improve software quality [1, 33], security [31], and to improve development time [32]. In our approach, we leverage MDE to treat code artefacts as models, to extract metadata to improve code analysis for interdisciplinary teams, and to selectively apply a set of transformations to linting outputs with the goal of reducing false positive rates. Contextual features extracted include **the purpose of the code** (which we plan to detect by analysing the operations implemented in the code). For example, we detect whether a code file pertains to **ML** functionality, or is intended as a **non-ML** module that provides the surrounding infrastructure. To the best of our knowledge, this is the first work to use context to improve code analysis for ML projects.

5 MLSMELLHOUND DEMONSTRATOR

To demonstrate the approach, we developed an initial prototype of MLSmellHound that provides context-aware linting[2]. Our tool is based on Pylint, a popular linter for Python projects that is highly configurable but that does not consider context. *The demonstrator shows how existing code analysis tools can be augmented with context using our approach.* For the purposes of demonstration, we use a simplified representation of the purpose of the source code as the context (whether a source file is an *ML* or *non-ML* module), although we will explore finer granularities (e.g. data wrangling, configuration, etc.) and additional dimensions of context (e.g. stage of the team data science process) in the future. To determine this simplified context, we traverse through each of the Python files in a given directory line by line. If we detect the presence of an ML library import, we flag that file as an *ML* module. A sample of the output is presented in Figure 3.

The following context transformations are implemented in the tool: **Subtraction/Addition:** Depending on the context, we configure Pylint to selectively enable or disable different rules, as well as use adjusted thresholds and patterns. For example, ML code

[2]Download MLSmellHound from: https://github.com/a2i2/ml-smell-hound

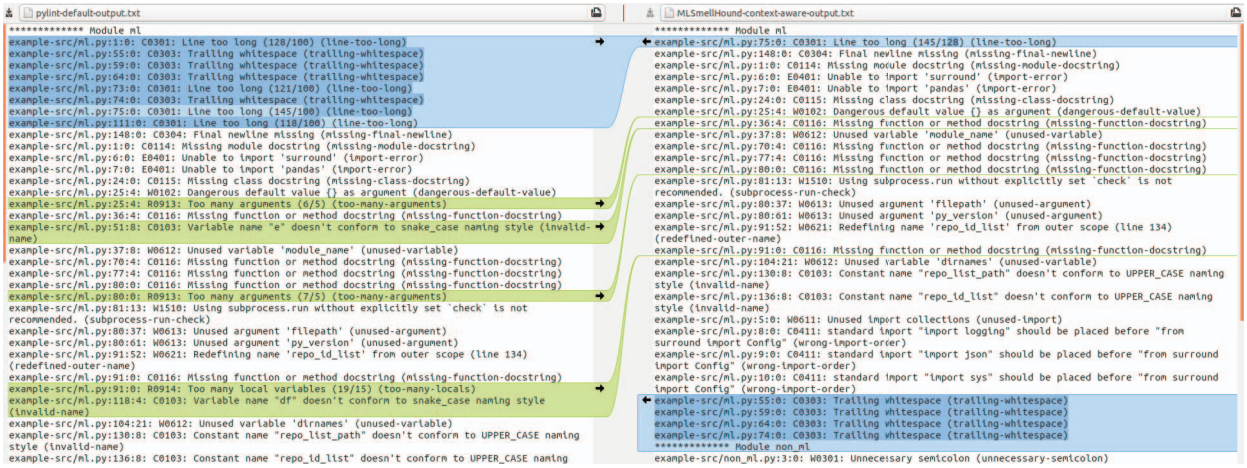

Figure 3: Screenshot showing difference between Pylint default output (left), and transformed ouput generated by MLSmellHound (right). Pylint warns about "e" and "df", whereas MLSmellHound accepts these in the context of ML files. The output also shows that trailing-whitespace warnings have been re-ranked to a lower priority for ML files.

often uses short variable names inspired by mathematical conventions [26], thus we use an altered regular-expression for validating conformance to variable naming conventions when linting ML files.

Remessage: A common smell detected in ML source code is an excessive number of parameters, which has been speculated to be a result of algorithms with many hyperparameters [26]. To ensure the message is understandable to data scientists we provide a simple mechanism for overwriting these messages by mapping Pylint symbols to custom messages tailored to the context. **Reprioritisation:** Our tool transforms the linting output by moving message categories that are not relevant for a particular context to the bottom of the generated output. E.g., ML code influenced by Tensorflow often uses non-standard indentation (two spaces instead of four), thus indentation warnings are deprioritised for ML files.

6 FUTURE PLANS

To realise the vision for MLSmellHound outlined in section 3 we have broken our approach down into four phases: i) qualitative evaluation of context for code analysis, ii) catalogue of ML smells and fixes, iii) metrics and context for locating ML smells, and iv) end-user evaluation of MLSmellHound.

Qualitative evaluation of context in code analysis. For further validation, we plan to extend Simmons et al. [26] by interviewing practitioners to better understand why coding conventions change based on context. The interviews will also be designed to understand the relevant context when building ML applications.

Catalogue of ML smells and fixes. We plan to conduct a mapping study to mine a catalogue of ML smells and anti-patterns with appropriate solutions. Our goal is to establish if ML anti-patterns map to existing software engineering code smells and analyse their impact on the occurrence of ML failures to create a set of ML-specific smells which map to defects that development teams need to consider while developing ML applications. To improve the completeness of the catalogue we will include both academic and gray literature (i.e. blogs, industry papers, and tutorials). From the catalogue we plan to identify context required for identifying an ML smell and will evaluate our findings with practitioners.

Metrics and context for locating ML smells. Based on the findings from the previous stage a set of smells will be selected. We will investigate how and where the context can automatically be mined for these smells in a software engineering project. These smells will be integrated into MLSmellHound to assess if context can be used to predict other ML smells.

End-user evaluation of MLSmellHound. Our plan is to run MLSmellHound against open-source repositories and ask participants to assess the i) accuracy of the reported smells, ii) usability of MLSmellHound, and iii) benefit for reducing failure in ML applications. We will also run a questionnaire to evaluate the relevance of context used in the predictions as a form of interpretation of the ML-smell prediction.

7 RISK

Differences in development exist between data scientists and software engineers in software quality and expected outcomes, for example software engineers may value software maintainability while data scientists may value the performance of the model in production. To account for this, we plan to identify these differences and include an approximation of the mental models of each discipline to ensure ML smells and warnings are explained and ranked in a manner that is compatible with the reasoning of that discipline.

Another risk for our research is the assumption that dimensions of context influence the relevance of code smells and the appropriate preventative action can be identified and mined from available artefacts. We plan to test this assumption through a controlled study to evaluate the relevance of context with practitioners compared to a control group using a tool that does not take into account context.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning NATURAL coding conventions. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 281–293. https://doi.org/10.1145/2635868.2635883 arXiv:1402.4182

[2] Nikolaos Bafatakis, Niels Boecker, Wenjie Boon, Martin Cabello Salazar, Jens Krinke, Gazi Oznacar, and Robert White. 2019. Python coding style compliance on stack overflow. In *IEEE International Working Conference on Mining Software Repositories*. IEEE Computer Society, 210–214. https://doi.org/10.1109/MSR.2019.00042

[3] Lucas Baier, Vincent Kellner, Niklas Kühl, and Gerhard Satzger. 2021. Switching Scheme: A Novel Approach for Handling Incremental Concept Drift in Real-World Data Sets. *Proceedings of the Annual Hawaii International Conference on System Sciences* (2021), 990–999. https://doi.org/10.24251/HICSS.2021.120 arXiv:2011.02738

[4] Scott Barnett. 2018. Extracting technical domain knowledge to improve software architecture. (2018).

[5] Benoit Baudry, Clementine Nebut, and Yves Le Traon. 2007. Model-Driven Engineering for Requirements Analysis. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. Institute of Electrical and Electronics Engineers (IEEE, 459–459. https://doi.org/10.1109/edoc.2007.15

[6] Jean Bézivin. 2006. Model driven engineering: An emerging technical space. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4143 LNCS (2006), 36–64. https://doi.org/10.1007/11877028_2

[7] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The open-closed principle of modern machine learning frameworks. In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. ACM Press, 353–363. https://doi.org/10.1145/3196398.3196445

[8] Aloisio S. Cairo, Glauco de F. Carneiro, and Miguel P. Monteiro. 2018. The Impact of Code Smells on Software Bugs: A Systematic Literature Review. *Information* 9, 11 (2018). https://doi.org/10.3390/info9110273

[9] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. 2016. Detecting code smells in python programs. In *Proceedings - 2016 International Conference on Software Analysis, Testing and Evolution, SATE 2016*. Institute of Electrical and Electronics Engineers Inc., 18–23. https://doi.org/10.1109/SATE.2016.10

[10] Ke Dai and Philippe Kruchten. 2017. Detecting technical debt through issue trackers. *CEUR Workshop Proceedings* 2017, QuASoQ (2017), 59–65.

[11] Hristina Gulabovska and Zoltán Porkoláb. 2019. Survey on static analysis tools of python programs. *CEUR Workshop Proceedings* 2508, September (2019), 22–25.

[12] Mark Haakman, Luís Cruz, Hennie Huijgens, and Arie van Deursen. 2021. AI lifecycle models need to be revised. an exploratory study in fintech. *Empirical Software Engineering* (2021).

[13] Nick Hynes, D Sculley, and Michael Terry. 2017. The Data Linter: Lightweight, Automated Sanity Checking for ML Data Sets. In *NeurIPS*.

[14] Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. 2020. The Scent of Deep Learning Code: An Empirical Study. *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020* (2020), 420–430. https://doi.org/10.1145/3379597.3387479

[15] Arvinder Kaur and Ruchikaa Nayyar. 2020. A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. *Procedia Computer Science* 171 (jan 2020), 2023–2029. https://doi.org/10.1016/J.PROCS.2020.04.217

[16] Ram Shankar Siva Kumar, David O Brien, Kendra Albert, Salomé Viljöen, and Jeffrey Snover. 2019. Failure Modes in Machine Learning Systems. (nov 2019). arXiv:1911.11034 https://arxiv.org/abs/1911.11034v1

[17] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[18] Luigi Lavazza, Sandro Morasca, and Davide Tosi. 2021. Comparing Static Analysis and Code Smells as Defect Predictors: An Empirical Study. *IFIP Advances in Information and Communication Technology* AICT-624 (may 2021), 1–15. https://doi.org/10.1007/978-3-030-75251-4_1

[19] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. 2020. SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings. *Journal of Systems and Software* 168 (oct 2020), 110671. https://doi.org/10.1016/J.JSS.2020.110671

[20] Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi. 2015. MARFCAT: Fast code analysis for defects and vulnerabilities. *2015 IEEE 1st International Workshop on Software Analytics, SWAN 2015 - Proceedings* (mar 2015), 35–38. https://doi.org/10.1109/SWAN.2015.7070488

[21] Nikhil Muralidhar, Sathappah Muthiah, Patrick Butler, Manish Jain, Yu Yu, Katy Burne, Weipeng Li, David Jones, Prakash Arunachalam, Hays 'Skip' McCormick, and Naren Ramakrishnan. 2021. Using AntiPatterns to avoid MLOps Mistakes. April (2021). arXiv:2107.00079 http://arxiv.org/abs/2107.00079

[22] Rajshakhar Paul. 2021. Improving the effectiveness of peer code review in identifying security defects. (aug 2021), 1645–1649. https://doi.org/10.1145/3468264.3473107

[23] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2011. A simpler model of software readability. In *Proceedings - International Conference on Software Engineering*. 73–82. https://doi.org/10.1145/1985441.1985454

[24] Reuters. 2018. *Amazon scraps secret AI recruiting tool that showed bias against women.* https://www.reuters.com/article/us-amazon-com-jobs-automation-insight-idUSKCN1MK08G

[25] D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean François Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*. 2503–2511.

[26] Andrew J. Simmons, Scott Barnett, Jessica Rivera-Villicana, Akshat Bajaj, and Rajesh Vasa. 2020. A large-scale comparative analysis of Coding Standard conformance in Open-Source Data Science projects. In *International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, New York, NY, USA, 1–11. https://doi.org/10.1145/3382494.3410680 arXiv:2007.08978

[27] The Verge. 2020. *AI camera operator repeatedly confuses bald head for soccer ball during live stream.* https://www.theverge.com/tldr/2020/11/3/21547392/ai-camera-operator-football-bald-head-soccer-mistakes

[28] Kristin Fjola Tomasdottir, Mauricio Aniche, and Arie Van Deursen. 2017. Why and how JavaScript developers use linters. In *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. Institute of Electrical and Electronics Engineers Inc., 578–589. https://doi.org/10.1109/ASE.2017.8115668

[29] Kristin Fjola Tomasdottir, Mauricio Aniche, and Arie Van Deursen. 2020. The Adoption of JavaScript Linters in Practice: A Case Study on ESLint. *IEEE Transactions on Software Engineering* 46, 8 (aug 2020), 863–891. https://doi.org/10.1109/TSE.2018.2871058

[30] Bart Van Oort, Luis Cruz, Mauricio Aniche, and Arie Van Deursen. 2021. The prevalence of code smells in machine learning projects. *Proceedings - 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI, WAIN 2021* (2021), 35–42. https://doi.org/10.1109/WAIN52551.2021.00011 arXiv:2103.04146

[31] Shao-Fang Wen and Basel Katt. 2019. Toward a Context-Based Approach for Software Security Learning. *Journal of Applied Security Research* 14 (2019), 1–20. https://doi.org/10.1080/19361610.2019.1585704

[32] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marcus, Paul Petersen, Jesmin Jahan Tithi, Tim Mattson, Tim Kraska, Pradeep Dubey, Vivek Sarkar, and Justin Gottschlich. 2020. Context-Aware Parse Trees. (2020). arXiv:2003.11118 http://arxiv.org/abs/2003.11118

[33] Xiaohan Yu, Quzhe Huang, Zheng Wang, Yansong Feng, and Dongyan Zhao. 2020. Towards Context-Aware Code Comment Generation. (2020), 3938–3947. https://doi.org/10.18653/v1/2020.findings-emnlp.350

[34] Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen, and Baowen Xu. 2019. How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects. *Empirical Software Engineering* 24, 6 (dec 2019), 3871–3903. https://doi.org/10.1007/S10664-019-09720-X