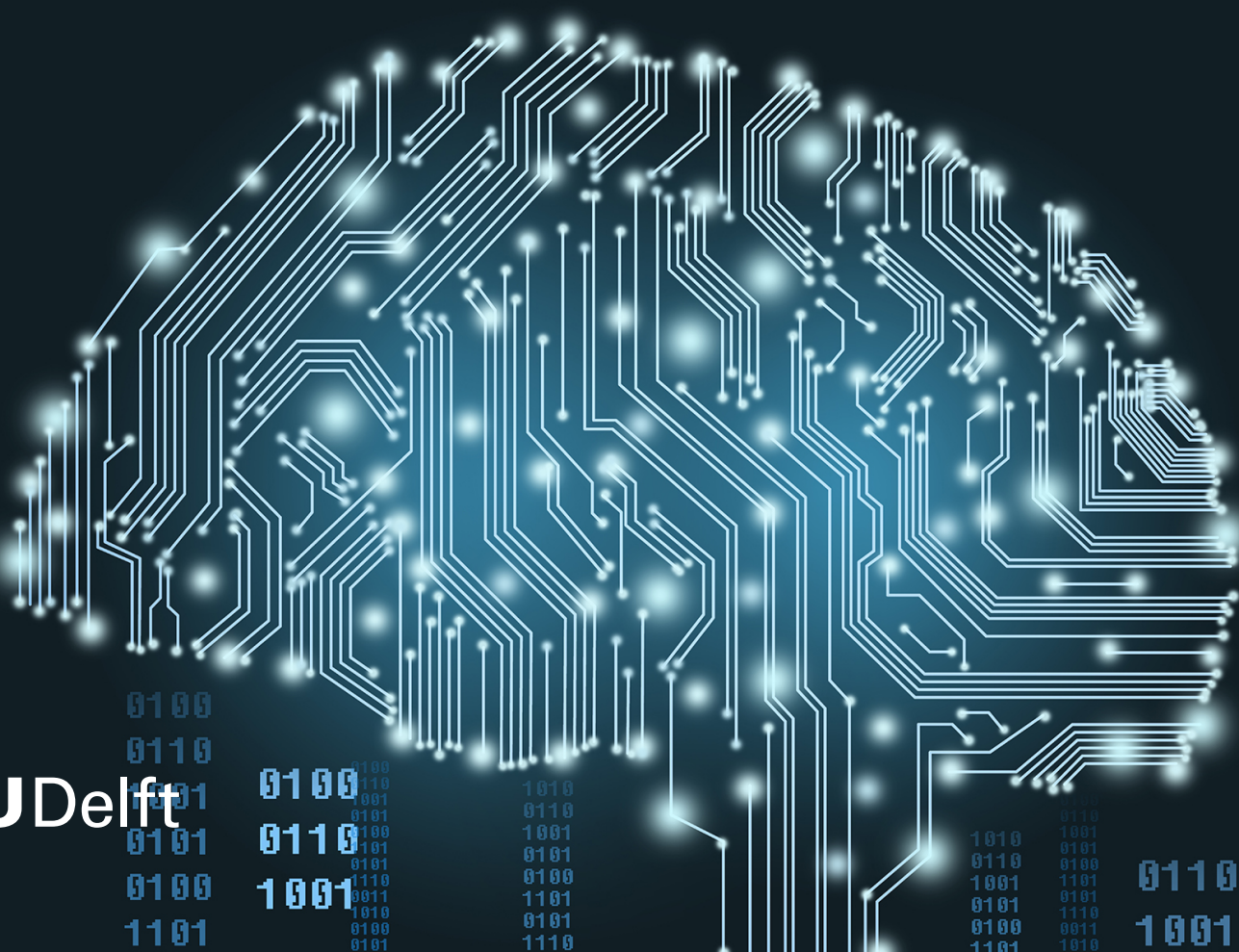


# Code smells & Software Quality in Machine Learning Projects

Bart van Oort

4343255

MSc Thesis — Computer Science — Software Technology  
18 October 2021



CODE SMELLS &  
SOFTWARE QUALITY  
IN MACHINE LEARNING PROJECTS

A thesis submitted to the Delft University of Technology in partial fulfillment  
of the requirements for the degree of

Master of Science

by

Bart van Oort

18 October 2021

© Bart van Oort. All rights reserved. Contact me on [bart@vanoort.is](mailto:bart@vanoort.is) for inquiries about copying or redistributing this work.

The work in this thesis was made in the:



Software Engineering Research Group  
Computer Science - Software Technology  
Faculty of Electrical Engineering, Mathematics & Computer Science  
Delft University of Technology



AI FOR FINTECH RESEARCH

AI for FinTech Research Lab  
Data, Tools & Technology  
ING Analytics  
ING Bank N.V.

Supervisors: Prof.dr. Arie van Deursen  
Dr. Luís Cruz  
Dr. Babak Loni  
Co-reader: Dr. Cynthia Liem MMus

# ABSTRACT

Artificial Intelligence (AI) and Machine Learning (ML) are pervasive in the current computer science landscape. Yet, there still exists a lack of Software Engineering (SE) experience and best practices in this field. One such best practice, static code analysis, can be used to find code smells, i.e., (potential) defects in the source code, refactoring opportunities, and violations of common coding standards. This research first set out to measure the prevalence of code smells in ML application projects. However, the results from this study additionally showed deficiencies in the dependency management of these projects, presenting a major threat to their maintainability and reproducibility. Static code analysis practices were also found to be lacking. These issues inspired the novel concept of *project smells* introduced in this research, which consider the ML project as a whole, including not just the code, but also the data, tools and technologies surrounding it and its development. To help ML practitioners in detecting and mitigating these project smells, as well as to help educate on SE principles, techniques and tools, I developed an open-source static analysis tool `mlLint` using input from experienced ML engineers at the global bank and data-driven organisation ING. This tool was then used to evaluate the concept of project smells and how they fit the industrial context of ING in a second study. This second study also investigated obstructions to implementing best practices recommended by `mlLint`, perceptions on static analysis tools and how ML practitioners perceive the difference in importance of `mlLint`'s linting rules (by extension, project smells) for proof-of-concept versus production-ready projects. The results indicate a need for *context-aware* static analysis tools, that fit the needs of the project at its current stage of development, while requiring minimal configuration effort from the user.

# PREFACE

This thesis was conducted at TU Delft's Software Engineering Research Group (SERG) under the AI for FinTech Research (AFR) Lab, which is a research collaboration between ING and TU Delft. ING is a global bank with a strong European base that offers retail and wholesale banking services to 38.5 million customers in over 40 countries [ING Bank N.V., 2021]. ING has extensive use-cases for increasing its business value with AI and ML, such as assessing credit risk, fighting economic crime by monitoring transactions and improving customer service. As part of a major shift in the organisation to adopt AI and ML, ING is defining standards for the different processes around the lifecycle of ML applications. Their AFR lab is part of their joint research effort on this topic and has weekly meetings where we update each other on our research and hold presentations about research on AI, SE or some combination thereof.

Throughout this thesis I have been in contact with ING. After submitting my first paper, in February 2021, I joined the ML Engineering chapter at ING (then led by Hadi Hashemi) and the GAP squad under Bart Buter's lead. Three days per week, I was present at their morning stand-ups and I have joined many of their ML Engineering chapter meetings (of which I also presented two). I also took part in two full ML Engineering chapter days to dive deeper into the world of ML engineering.

*Note: this thesis is atypical in that it consists primarily of two papers written over the course of this thesis. Please read [Section 1.2](#) to learn how this affects this report's structure.*

## ACKNOWLEDGEMENTS

Throughout my MSc thesis I have been supported by several great people in various ways, whom I would like to thank for their help and expertise. First and most of all, I thank Luís Cruz, my daily supervisor, for his excellent supervision and his both academic as well as moral support and guidance. I also thank Maurício Aniche for connecting me with this thesis, co-supervising me during the first half of my thesis and being very supportive overall. My thanks also go out to Prof. Arie van Deursen for overseeing my research and providing me with direction at several key crossroads throughout my thesis. Lastly, I thank Cynthia Liem for co-reading this thesis and taking part in my thesis committee.

On the ING side, I would particularly like to thank Elvan Kula for her social leadership in the AFR Lab and being my direct supervisor at ING. I also thank Hadi Hashemi for his supervision of me at ING and taking me under his wing in the ML Engineering chapter. Naturally, I also thank Babak Loni for his continued supervision and guidance when he succeeded Hadi as chapter lead and for taking part in my thesis committee.

My thanks also go out to all members of SERG, the AFR Lab, the ML Engineering Chapter and the GAP Squad, of the latter in particular Bart Buter, Amrit Purshotam and Ralf Wolter. Additionally, I want to thank the members of SE4ML, specifically Alex Serban and Joost Visser for their insights, expertise and periodic feedback during the development of `mlLint`.

Finally, none of my brain would have remained sane without the continued social and personal support from my dearest friends, family and relatives.

# CONTENTS

1	INTRODUCTION	1
1.1	Contributions & Publications	2
1.2	Report Structure	3
2	THE PREVALENCE OF CODE SMELLS	4
2.1	Academic publication & presentation	4
2.2	The Prevalence of Code Smells — Full Paper	6
3	MLLINT – DETECTING PROJECT SMELLS & IMPROVING SOFTWARE QUALITY	14
3.1	Implementation	14
3.2	Peer interactions	15
3.3	Evaluation efforts	15
3.4	Other information	16
3.5	“Project smells” — Full Paper	17
4	CONCLUSION & FUTURE WORK	28
4.1	Future Work	28
A	TABLE OF LINTING RULE PRIORITISATION SURVEY RESULTS	31
A.1	Proof-of-concept project	31
A.2	Production-ready project	33

# 1

## INTRODUCTION

This MSc thesis originally started out as research into code smells in ML applications; the early goal was to tread in the footsteps of a previous master thesis by Haakman [2020], who studied how state-of-the-art lifecycle models fit the current needs of the FinTech industry [Haakman et al., 2021] and pioneered `dslinter`<sup>1</sup>, a plugin for Pylint to help detect Data Science (DS) and Machine Learning (ML) specific code smells in Python ML applications. His results indicated that “*existing lifecycle models CRISP-DM and TDSP largely reflect the current development processes of machine learning applications, but there are crucial steps missing, including a feasibility study, documentation, model evaluation, and model monitoring*” [Haakman, 2020; Haakman et al., 2021]. On the topic of ML-specific code smells, he implemented, publicised and evaluated `dslinter` on ML application code from 1000 Kaggle-sourced Jupyter Notebooks, showing that such static code analysis can be helpful in detecting and preventing ML-specific code smells [Haakman, 2020]. Yet, he notes that “*more focus is needed on the entire lifecycle.*”

As a full-stack web developer, software engineer and DevOps-enthusiast with *some* experience in developing ML applications, though little in-depth knowledge of ML algorithms, I first needed to acquaint myself with the current state of ML application code. Thus, I started with an initially simple empirical study on the prevalence of non-ML-specific code smells in ML projects. To this end, I collected a dataset consisting of the source code repositories of 74 open-source Python ML applications from academic and industry-related sources alike, installed their dependencies<sup>2</sup> and ran the popular static analysis tool Pylint on them in its default configuration.

However, we quickly noticed that installing the dependencies of these ML projects was not trivial: 32 out of 74 projects required generating or manually editing the dependency specification file (`requirements.txt`) used in the project, in order for the dependencies to install without error. These are serious issues in code dependency management in Python ML projects that present a major threat to the reproducibility and maintainability of these projects. Additionally, we noticed that very few of the projects contained any configurations for static analysis tools such as Pylint, indicating a gap in their adoption with data scientists.

These dependency management and static analysis tool adoption issues, along with Haakman's note that more focus is needed on the entire lifecycle, inspired the realisation that a more holistic approach to code smells would be required in order to assess the software quality of ML applications. This coined the idea of *project smells* that consider the project as a whole, including not just the code, but also the data, tools and technologies surrounding it and its development. These project smells are thus concerned with (technological) deficits in the development process and management of ML projects, including poor dependency management and a lack of static analysis. Code smells are also a subcategory of project smells.

To help ML practitioners in detecting and mitigating these project smells, as well as to help educate them on SE principles, techniques and tools—additionally giving this thesis more of a practical impact

---

<sup>1</sup> <https://github.com/MarkHaakman/dslinter>

<sup>2</sup> Installing a project's dependencies is required for certain Pylint rules to work effectively

beside its scientific contributions—I developed a static analysis tool called `mllint` using input from experienced ML engineers at ING. This tool uses a command-line interface and is open-sourced under a GPLv3 license. It was presented at ING on multiple occasions and described as ‘*ambitious*’, but was overall received positively, gathering 54 stars on GitHub.

For the final chapter in this thesis, the goal was to evaluate `mllint` and its concept of project smells. In this research, we performed qualitative analysis on the reports that `mllint` produces on a small set of ML projects at ING and run a survey with ML practitioners (public and at ING) who have experience with `mllint`, to gauge their perceptions on the benefits and drawbacks of using static analysis tools such as `mllint` and the importance of each of the linting rules of `mllint` for proof-of-concept versus production-ready ML projects.

The results of this research were sufficiently promising that we decided to disseminate them through a paper submitted to a top-tier conference, for which we picked the Software Engineering In Practice (SEIP) track of ICSE, given our practice-oriented tool and case at ING. The results indicate a lack of standardised tooling for dealing with varying kinds of data dependencies, as well as a lack of a standardised, easy-to-use, consistently used, maintainable and reproducible method of managing code dependencies in the Python language ecosystem. It was also found that there is a mixed sentiment towards static analysis tools to detect code quality issues: while ML practitioners recognise their potential for code quality assurance in productionising ML projects, there is apprehension in adopting them during the development phase of the project, given their tendency to produce false positives and cumbersome, time-consuming configuration. This calls for more research on *context-aware* static analysis, such that it fits the needs of the project at its current stage of development, while requiring minimal configuration effort from the user.

## 1.1 CONTRIBUTIONS & PUBLICATIONS

The main contributions of this MSc thesis are as follows:

1. A replicable, empirical study into prevalence of code smells as detected by Pylint in ML projects, giving insights into the code smells most prevalent in ML applications, the shortcomings of Pylint for performing static analysis on ML code and challenges relating to dependency management in Python ML projects.
2. A dataset of 74 open-source ML application projects and an open-source tool to perform simultaneous static code analysis on all of these projects.<sup>3</sup>
3. Researched insights on and experiences with the novel concept of *project smells*: a more holistic approach to code smells that considers the project as a whole, including not just the code, but also the data, tools and technologies surrounding it.
4. An open-source static analysis tool `mllint`<sup>4,5</sup> to help detect and remedy such project smells in ML projects, as well as report on the overall software quality of the project. `mllint` aims to help ML practitioners in developing and maintaining production-grade ML and AI projects.

On top of that, the contributions made in this thesis resulted in the following scientific publications:

<sup>3</sup> <https://gitlab.com/bvobart/python-ml-analysis>

<sup>4</sup> <https://github.com/bvobart/mllint>

<sup>5</sup> <https://bvobart.github.io/mllint>



1. A research paper about the empirical study on the prevalence of code smells submitted to, published and presented at the 1st Workshop on AI Engineering (WAIN'21), associated with the International Conference on Software Engineering (ICSE) [van Oort et al., 2021].
2. A research paper submitted to the Software Engineering In Practice (SEIP) track of ICSE on project smells, how they fit in the FinTech context of ING, the benefits and drawbacks of using static analysis tools such as `mlLint` as perceived by ML practitioners, and the perceived importance of each of the linting rules of `mlLint` for proof-of-concept versus production-ready ML projects [van Oort et al., 2022].

## 1.2 REPORT STRUCTURE

This report consists primarily of the two papers produced over the course of this MSc thesis, with each paper containing its own respective introduction, related work, methodology, results, discussion, threats to validity and concluding sections. This introductory [Chapter 1](#) serves to shine a spotlight behind the scenes and explain the origin of this thesis, as well as the rationale behind the direction of my research. Additionally, it summarises the contributions and structure of this thesis.

Both papers in this thesis are presented exactly as they were published / submitted to their respective venue. Each is placed in their own chapter, preceded by an introductory text describing the context of the study, along with information about work done that is relevant to this thesis report, but not to the readers of the academic venues that the papers were submitted to. The sections, figures, tables, footnotes and references of each paper are constrained to their respective manuscripts and are thus not continuous throughout this thesis. This also goes for the page numbers at the bottom of the WAIN'21 paper: the true page numbers of this thesis report along with the current chapter name are shown in the top-right of the page, or on the bottom-right of the page if the page contains a chapter header.

As for the bibliography at the end of this thesis report, it merely contains academic references made in this introduction ([Chapter 1](#)), the conclusion ([Chapter 4](#)) or in the introductory texts of the paper chapters. Any references made in the papers themselves, are specified in the respective paper's bibliography section.

[Chapter 2](#) introduces the first paper on *The Prevalence of Code Smells in Machine Learning projects* as published at WAIN'21. Next, [Chapter 3](#) introduces the second paper on the topic of project smells and experiences in analysing the software quality of ML projects with `mlLint` as submitted to SEIP'22. Finally, [Chapter 4](#) concludes this thesis with recommendations for future work.

## 2 | THE PREVALENCE OF CODE SMELLS

To get better acquainted with the current state of ML application code and code smells, I started with an empirical study on the prevalence of known Python code smells in ML projects. To this end, our approach was simple and straightforward: collect a dataset of source code repositories of ML applications, install their dependencies and run the popular Python static analysis tool Pylint<sup>1</sup> on their source code to detect code smells. Then, aggregate the resulting list of code smells across all projects and analyse their prevalence.

Initially, our goal for our dataset of ML applications was to have a rich mixture of both open-source and closed-source industry projects, such that we could either combine or contrast them. Combining them would provide us with a more representative sample of the current, real-world state of ML and AI projects. Contrasting them would allow us to compare the prevalence of code smells in open-source versus industry projects. However, due to difficulties in getting on-boarded at ING along with their intricate web of compliance regulations to be navigated in gaining access to ML project source code, we were not able to collect a sample of ML projects from ING and thus had to abandon the combination or comparison with industry projects.

The results of our code smell analysis with Pylint, combined with manual inspection of the detected smells, show that code duplication is widespread. Furthermore, the PEP8 convention for identifier naming style may not always be applicable to ML code due to its resemblance with mathematical notations of the underlying ML algorithm. We also found that Pylint produces a high rate of false positives in trying to analyse correct usage of import statements in certain cases, which are specifically concerning given the tendency for ML to suffer from a high degree of glue code [Sculley et al., 2015]. More interestingly, however, we found serious issues with the specification of code dependencies in Python ML projects that present a major threat to the reproducibility and maintainability of these projects. Out of the 74 projects that we analysed, 32 required generating or manually editing the dependency specification file (`requirements.txt`) used in the project, in order for the dependencies to correctly install. Usage of the `pip freeze` command was observed to be a major culprit in these cases.

### 2.1 ACADEMIC PUBLICATION & PRESENTATION

From the start of this research, we aimed at encapsulating our findings in a paper for the 1st Workshop on AI Engineering (WAIN'21<sup>2</sup>). While the submission deadline was only two months after the start of my thesis, our findings around dependency management issues were a large motivator towards completing our research and submitting the paper on time. This work paid off in full, as our paper was accepted and published at WAIN'21! It is worth noting that WAIN'21 is a peer-reviewed conference

---

<sup>1</sup> <https://pylint.org/>

<sup>2</sup> <https://conf.researchr.org/home/wain-2021>

workshop co-located with ICSE, where only 12 out of 37 submitted full papers were accepted (32% acceptance rate).

Given that the paper was accepted, I also presented it at WAIN'21, for which I recorded and edited a video presentation of the paper<sup>3</sup>. This was broadcast at the online WAIN'21 conference workshop on the 31st of May 2021, followed by a live Q&A session. The full recording including the Q&A can be found on YouTube<sup>4</sup>. Two workshop participants were particularly interested in my paper, as they messaged me on the conference platform the day before my presentation to ask several questions about my research. One of them, who had just started his PhD around code quality in ML projects, lauded our research and asked for advice as he wanted to replicate our research on ML code from Jupyter Notebooks. Three other workshop participants joined the discussion room directly after the presentation and Q&A to ask some short questions.

Additionally, this paper was presented in two one-hour presentations in February at an ML Engineering chapter meeting in ING Global Analytics and in an AFR Lab meeting.

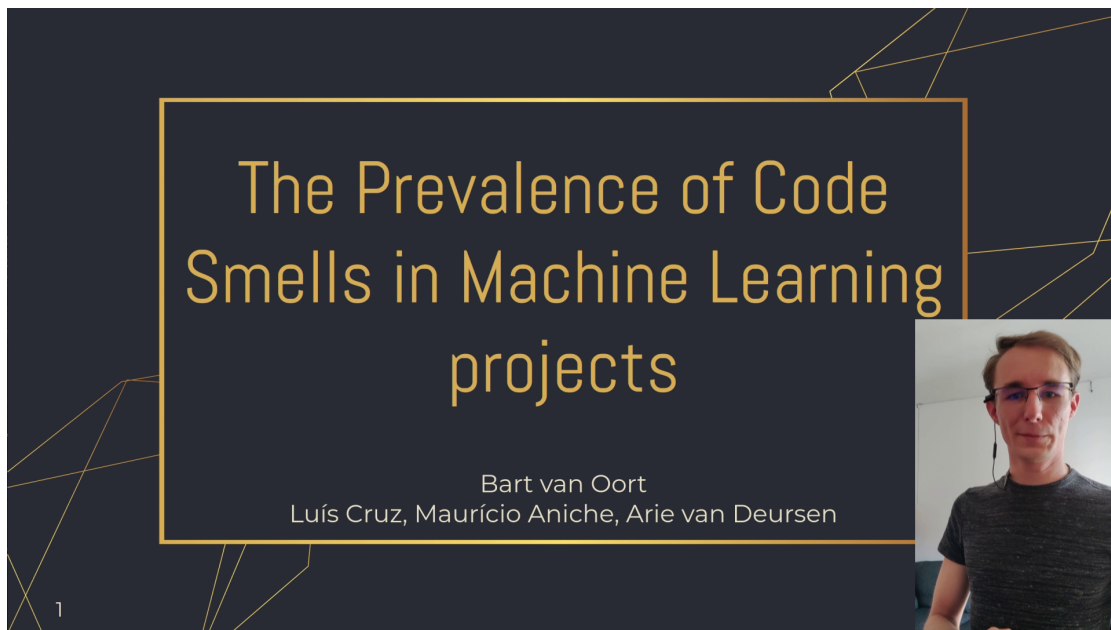


Figure 2.1: Snapshot from the video presentation of van Oort et al. [2021]

<sup>3</sup> <https://www.youtube.com/watch?v=TbgawiiYwJQ>

<sup>4</sup> <https://www.youtube.com/watch?v=fQf2Cy9jzfc>

# The Prevalence of Code Smells in Machine Learning projects

Bart van Oort<sup>1,2</sup>, Luís Cruz<sup>2</sup>, Maurício Aniche<sup>2</sup>, Arie van Deursen<sup>2</sup>

*Delft University of Technology*

<sup>1</sup> *AI for Fintech Research, ING*

<sup>2</sup> *Delft, Netherlands*

bart.van.oort@ing.com, {l.cruz, m.f.aniche, arie.vandeursen}@tudelft.nl

**Abstract**—Artificial Intelligence (AI) and Machine Learning (ML) are pervasive in the current computer science landscape. Yet, there still exists a lack of software engineering experience and best practices in this field. One such best practice, static code analysis, can be used to find code smells, i.e., (potential) defects in the source code, refactoring opportunities, and violations of common coding standards. Our research set out to discover the most prevalent code smells in ML projects. We gathered a dataset of 74 open-source ML projects, installed their dependencies and ran Pylint on them. This resulted in a top 20 of all detected code smells, per category. Manual analysis of these smells mainly showed that code duplication is widespread and that the PEP8 convention for identifier naming style may not always be applicable to ML code due to its resemblance with mathematical notation. More interestingly, however, we found several major obstructions to the maintainability and reproducibility of ML projects, primarily related to the dependency management of Python projects. We also found that Pylint cannot reliably check for correct usage of imported dependencies, including prominent ML libraries such as PyTorch.

**Index Terms**—Artificial Intelligence, Machine Learning, static code analysis, code smells, Python, dependency management.

## I. INTRODUCTION

Artificial Intelligence (AI) and Machine Learning (ML) are pervasive in the current landscape of computer science. Companies such as Facebook, Google, Nvidia and ING are making use of AI and ML for a plethora of tasks that are difficult (if not impossible) to describe using traditional Software Engineering (SE) [1, 2, 3, 4]. Examples include facial recognition & recomposition, natural language processing, real-time video transformation, detection of medical anomalies and intercepting fraudulent financial transactions.

Yet, as Sculley et al. [2] wrote in their 2015 paper on the hidden technical debt in ML systems at Google, “*only a small fraction of real-world ML systems is composed of the ML code. (...) The required surrounding infrastructure is vast and complex.*” This is also in part what leads Menzies [5] to predict that the future of software will be a rich and powerful mix of ideas from both SE and AI. Menzies also advocates for more SE experience in the field of AI and ML, stating that poor SE leads to poor AI while better SE leads to better AI [5]. The data scientists that write AI / ML code often come from non-SE backgrounds where SE best practices are unknown [6].

One such SE best practice is the practice of static code analysis to find (potential) defects in the source code, refactoring opportunities and violations of common coding standards,

which we amalgamate into ‘code smells’ for the rest of this paper. Research has shown that the attributes of quality most affected by code smells are maintainability, understandability and complexity, and that early detection of code smells reduces the cost of maintenance [7].

With a focus on the maintainability and reproducibility of ML projects, the goal of our research is therefore to apply static code analysis to applications of ML, in an attempt to uncover the frequency of code smells in these projects and list the most prevalent code smells. Thus, we formulate the following research question: *What are the most prevalent code smells in Machine Learning code?*

The main contributions of this paper are:

- An empirical study on the prevalence of code smells in 74 Python ML projects.
- A dataset of 74 ML projects and an open-source tool to perform simultaneous static code analysis on all of these projects.

## II. RELATED WORK

Several studies have investigated linting and static code analysis of non-ML projects [8, 9, 10, 11]. Tómasdóttir et al. [8] researched why JavaScript (JS) developers use linters and how they tend to configure them. They found that maintaining code consistency, preventing errors, saving discussion time and avoiding complex code were among the top reasons why JS developers use linters. They also found that JS developers commonly stick with already existing preset linting configurations. Vassallo et al. [12] found a similar result; among other results, they found that developers are often unwilling to configure automatic static analysis tools (ASATs) and emphasise “*the necessity to improve existing strategies for the selection and prioritisation of ASATs warnings that are shown to developers.*”

Within the Python ecosystem, Chen et al. [11] investigated the detection and prevalence of code smells in 106 Python projects with the most stars on GitHub. They found that long parameter lists and long methods were more prevalent than other code smells. Omari and Martinez [9] used Pylint to analyse the code quality of a dataset of large Python projects. Furthermore, Bafatakis et al. [10] used Pylint to investigate the Python coding style compliance of StackOverflow answers.

Within the Machine Learning ecosystem, we only found one paper by Simmons et al. [6] that performed static code

analysis on a large dataset of Data Science (DS) projects. They also analysed non-DS projects with the goal of comparing the code quality and coding standard conformance of (open-source) DS projects versus non-DS projects, using Pylint in its default configuration as a metric. They sourced their DS projects from Biswas et al. [13], who in 2019 published a dataset of 1558 “*mature Github projects that develop Python software for Data Science tasks.*”. Aside from applications of ML, it also includes ML libraries and tools.

Our study differs from [6] in that we do not compare against non-DS projects and in that we do not solely focus on the adherence to coding standards as [6] does. Our primary focus lies more on investigating obstructions to the maintainability and reproducibility of ML projects, which includes coding standards violations, but also entails recognising refactoring opportunities and other code smells [7]. Moreover, we solely focus on applications of ML, and leave ML libraries and tools out of scope. We argue that the underlying nature of ML libraries and tools is very different from ML applications, and thus different results are expected when studied separately.

Furthermore, Simmons et al. [6] simplified the installation of the projects’ dependencies by using `findimports`<sup>1</sup> to resolve all imports used in the projects, instead of relying on what projects’ authors defined in their repositories, noting that “*it was impractical to reliably determine and install dependencies for the projects analysed.*” However, if there is an inherent difficulty in resolving these dependencies within Python projects, then that is in itself an obstruction to the reproducibility and maintainability of these projects. Hence, we investigate this in our study.

### III. METHODOLOGY

For this paper, we performed an empirical study on the prevalence of code smells in ML code. We collected a dataset of 74 ML projects and implemented a tool to set these projects up with their dependencies in order to replicate their execution environment. It then runs Pylint with its default configuration on all projects in the dataset, collecting and counting the detected code smells. The tool and dataset are both open-source and can be found on GitLab<sup>2</sup>.

Our empirical study follows the methodology illustrated in Figure 1. It comprises three main steps, namely: A) project selection, B) setting up the codebases, and C) static analysis.

#### A. Project Selection

In total, our collected dataset comprises 74 ML projects; 32 projects come from finished Kaggle competitions, 38 from paperswithcode.com (of which 25 projects were from the Google-affiliated DeepMind), and 4 from reproducedpapers.org. It includes projects from academic papers, (student) reproductions, prize money awarding Kaggle competitions, as well as industry players such as Facebook, Nvidia and DeepMind. The dataset defines a list of Git repository URLs and allows for customising the dependencies of particular projects, when they

have not been properly defined in their respective repository. We elaborate on a number of characteristics of our dataset in Section III-A1, but first, we explain how we collected the projects in the dataset and what guidelines were used for doing so.

We aim for this dataset to be a systematically gathered set of projects, representative of the current, real-world state of ML and AI projects. To this end, we have created a set of guidelines for the inclusion of projects in the dataset, which can be found below. Each project included in the dataset...

- 1) ...must be hosted in an open-source Git repository.
- 2) ...must be written in Python 3.
- 3) ...must contain pure Python code and does not consist purely of Jupyter Notebooks. More specifically, a project should contain *either* a) at least 200 lines of pure Python code, even if the rest of the code is embedded in Jupyter Notebooks, *or* b) more lines of pure Python code than there are lines of Python code in all Jupyter notebooks of that project.
- 4) ...must implement an ML or AI model and may not be a library or tool for use in ML projects.
- 5) ...must be considered ‘deliverable’, i.e., *either* a) the project is part of or accompanies a published academic paper, *or* b) the project has been submitted to paperswithcode.com, reproducedpapers.org or a Kaggle competition (which has finished and declared the winners at the time of considering the project).

The first guideline limits our scope to open-source projects, as these are openly available to download and analyse.

The second and third guideline stem from a technical limitations, as Pylint only supports Python 3 and is only able to analyse pure Python files. Jupyter Notebooks are essentially JSON files, containing ‘cells’ with code in Markdown, Python, Julia, or a small selection of other different languages. While it is technically possible to convert the Python code embedded in these notebooks to pure Python files using a tool such as `nbconvert`, the produced code has a slightly different style than general Python modules, which invalidates certain Pylint rules. For example, the Pylint messages `pointless-statement`, `expression-not-assigned` and `wrong-import-statement` produce false positives in notebook-style code. Due to the lack of direct Pylint support for Jupyter Notebooks and since we do not want to selectively disable Pylint rules for notebook-extracted code as opposed to pure Python code, we decided to exclude projects that purely contain Jupyter Notebooks from our dataset. The minimum of 200 lines of pure Python code in the presence of larger Jupyter Notebooks was chosen such that this code is likely not to be purely utility code, but also contain part of the ML code.

The fourth guideline embodies that we are interested in analysing applications of ML rather than libraries used in their development, such as `tensorflow`, `pandas`, or `sklearn`.

The fifth and final guideline focuses on avoiding toy projects, unfinished projects, or projects still under development.

<sup>1</sup><https://pypi.org/project/findimports/>

<sup>2</sup><https://gitlab.com/bvobart/python-ml-analysis>

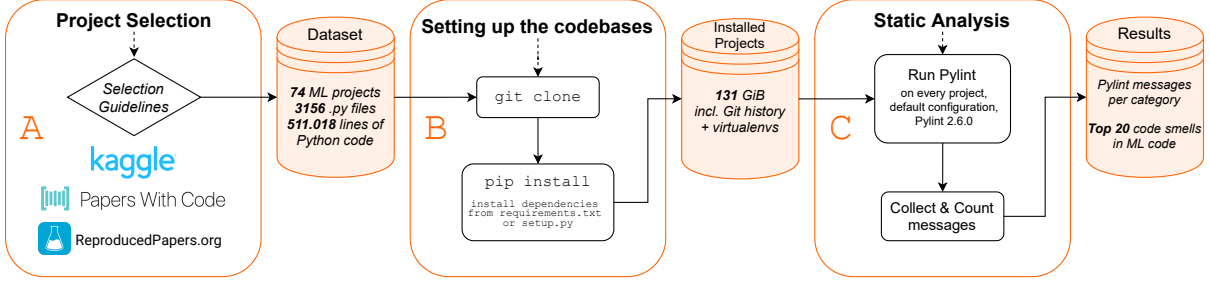


Fig. 1. Methodology Diagram.

1) *Dataset characteristics*: We measured several general characteristics of every project, which can be found in Table I. The 74 projects in the dataset contain a total of 3156 pure Python files, amounting to 511,018 lines of Python code, including empty lines. The median project has 17 pure Python files with 2848.5 lines of code, resulting in an average of 157.3 lines of Python code per file. The smallest project contained a single file with 58 lines of Python, while the largest project had 78572 lines of Python code across 229 files. This project was found to be embedding the code of several dependencies in its repository, multiple times.

#### B. Setting up the codebases

In this step, performed by our analysis tool, for each project, we clone the latest version of the project’s Git repository, create a virtual environment in it, ensure that there exists a file that specifies any necessary dependencies and then install those dependencies into the virtual environment. These need to be installed, such that our static analysis tool of choice is able to check whether imports resolve correctly and whether the imported libraries are used correctly. This is particularly interesting in the case of ML, as Sculley et al. [2] noted that ML projects have a high degree of glue code and so make extensive use of libraries. In total, the folder containing all of the 74 cloned projects from our dataset along with the accompanying virtual environments with their installed dependencies, amounts to 131 GiB.

Python projects can specify and install their dependencies in a variety of ways. The most common way to install a Python dependency is to use `pip`, the package manager that is installed alongside Python. It is common convention to specify a Python project’s list of dependencies in a requirements file called `requirements.txt`, which is conventionally placed at the root of the project’s source code repository. It is also possible to specify a `setup.py` file which allows the project to be built into a Python package, ready to be published to PyPI, `pip`’s default package index. `pip` can be configured to use other package indexes, but by default it can only install packages from PyPI, or directly from source through a Git repository URL or a local folder with a `setup.py`.

However, there are also other package managers / dependency management solutions such as Conda, Poetry, `pip-tools` and `Pipenv`, with the latter being directly endorsed in Python’s Packaging User Guide [14]. These tools each have their own way of specifying dependencies and – especially in Conda’s case – may use additional package indexes to

PyPI, which makes resolving these dependencies difficult. It is possible to use `pip freeze > requirements.txt`, which collects all Python packages and their exact versions installed in the current Python environment (disregarding by which means these packages were installed) and outputs them to a `requirements.txt` file. This approach is flawed though, as we explain in Section V-B.

Our analysis tool currently only supports installing dependencies with `pip` and expects a `requirements.txt` or `setup.py` file in their conventional location. The dataset also supports specifying a custom path to a `requirements.txt` file, or alternatively, the contents of a custom `requirements.txt` file for a project in the dataset. It is also possible to specify extra requirements that need to be installed *after* installing the dependencies from the requirements file. This is necessary for, e.g., Nvidia’s Apex library, which depends on PyTorch; when trying to run `pip install` on a requirements file containing both PyTorch and Apex’s Git repository URL (no matter the order), the installation of Apex fails because PyTorch is not yet installed. Only for projects that do not have a `requirements.txt` file, nor a manually defined one, our analysis tool uses `pipreqs`<sup>3</sup> to generate a `requirements.txt` file based on the libraries imported in the code.

Our analysis tool currently does not support using Conda, Poetry or `Pipenv` for resolving and installing dependencies. We therefore had to exclude one project that used Poetry and two projects that used Conda and solely specified a Conda `environment.yml` file, but no `requirements.txt` or `setup.py`. No projects that we came across were using `Pipenv`.

#### C. Static Analysis

This step is also performed by our analysis tool and concerns running the static code analysis tool Pylint (version 2.6.0) in its default configuration on all pure Python files in each project (but not on any of the dependencies). We choose Pylint for static code analysis as it is widely used and widely accepted in the Python community, as well as being highly configurable [6, 10]. It is also well integrated into IDEs such as PyCharm and VS Code. Furthermore, Bafatakis et al. [10] used it to measure coding style compliance in StackOverflow answers, Omari and Martinez [9] used it as a metric for the code quality of open-source Python projects, and Simmons

<sup>3</sup><https://github.com/bndr/pipreqs>

TABLE I  
CHARACTERISTICS OF OUR DATASET OF 74 ML PROJECTS

Characteristic	Min	Q1	Median	Q3	Max	Mean	Std Dev.
Number of pure Python files	1	8	17	36	730	43	95
Number of Jupyter Notebook files	0	0	0	1	52	3	8
Lines of Python code	58	1449	2849	5243	78572	6906	13568
Lines of Jupyter Notebook Python	0	0	0	197	43387	1008	5115
Avg. lines of Python code per Python file	58	108	157	214	1151	193	155
Avg. lines of Jupyter Notebook Python per Jupyter Notebook file	12	73	129	223	1610	256	335

et al. [6] used it in their code quality comparison between DS and non-DS Python projects.

Pylint provides an extensive set of messages, not only for stylistic issues, but also for issues regarding programming conventions, possible refactorings and other logical code smells. While Pylint is very configurable, we chose to use Pylint’s default configuration as it reflects the community standards, similar to Simmons et al. [6].

The code smells that Pylint reports are each identified by a symbol, such as `bad-indentation` or `import-error`, which is also how we refer to specific Pylint messages in this paper. Furthermore, these messages are divided into five categories (message types), which we describe below. The italic text is how Pylint describes the category.

- **Convention** – *for programming standard violation* – Messages in this category show violations of primarily code style conventions, as well as documentation conventions and Pythonic programming conventions.
- **Refactor** – *for bad code smell* – Messages in this category indicate that the smelly code should be refactored.
- **Warning** – *for Python specific problems* – This category includes many generic and Python-specific linting messages.
- **Error** – *for probable bugs in the code* – Messages in this category indicate problems in the code that are very likely to cause run-time problems.
- **Fatal** – *if an error occurred which prevented Pylint from doing further processing.*

Cloning and installing all projects, even though this is performed automatically by the tool, is the most time-consuming part of the analysis – it takes roughly three hours. With all projects already cloned and their dependencies already installed, the analysis of all 74 projects took 8m 53s using 12 threads on an Intel® Core™ i7-8750H processor.

Eight projects contained code that caused Pylint to crash during analysis, so we excluded these from the 82 projects we originally had in the dataset, bringing the total to 74. Several issues have been filed about this, including one by this paper’s first author<sup>4</sup>. This bug has since been fixed.

#### IV. RESULTS

Applying our methodology, we collected, installed, and analysed 74 ML projects. In this section, we present our results and answer to the research question posed in the introduction:

- **RQ** – *What are the most prevalent code smells in Machine Learning code?*

<sup>4</sup><https://github.com/PyCQA/pylint/issues/3986>

TABLE II  
DISTRIBUTION OF PYLINT MESSAGES PER CATEGORY PER PROJECT.

Category	Min	Q1	Median	Q3	Max	Mean	Std Dev.
Convention	2	57	226	799	9501	708	1361
Refactor	0	26	49	140	2437	183	433
Warning	11	74	356	824	14263	814	1826
Error	1	18	56	125	1696	129	234
Fatal	0	0	0	0	0	0	0

TABLE III  
TOP 10 CODE SMELLS OVERALL AS DETECTED BY PYLINT.

#	Smell	Frequency
1	unused-wildcard-import	26307
2	bad-indentation	19921
3	invalid-name	19905
4	line-too-long	10321
5	missing-function-docstring	6444
6	no-member	5860
7	duplicate-code	4649
8	trailing-whitespace	4477
9	redefined-outer-name	2548
10	missing-module-docstring	2504

To answer this, we first analysed the distribution of the amount of code smells per Pylint category per project, of which the characteristics can be found in Table II. The table shows the minimum, maximum, mean and median number of messages reported by Pylint for each category, as well as the 25th percentile (Q1), 75th percentile (Q3), and standard deviation (Std. Dev.). We use the median as the main measure of central tendency.

Our results show that Pylint messages in the Warning category are the most prevalent – the median project has 356 warnings – closely followed by messages in the Convention category with 226 messages for the median project. Messages in the Refactor and Error categories are less prevalent; respectively 49 and 56 such messages for the median project. However, especially given the Error category is meant for messages that show “probable bugs”, this is an interesting observation. Even more interesting, there was *no* project for which Pylint reported *no* error messages.

As a more direct answer to this research question, we measured across all projects in our dataset what the top 20 code smells per category are that Pylint reported, see Table IV. The top 10 messages that Pylint reported, disregarding category, are in Table III.

**Convention** – In this category we found that invalid naming, missing documentation (`missing-function-docstring`, `missing-module-docstring`, `missing-class-docstring` and `missing-docstring`) and improper organisation of imports (`wrong-import-position`, `wrong-import-order`,



TABLE IV  
TOP 20 MESSAGES PER CATEGORY REPORTED BY PYLINT ALONG WITH HOW OFTEN THEY WERE COUNTED ACROSS ALL PROJECTS.

Convention			Refactor			Warning			Error		
Symbol	Count		Symbol	Count		Symbol	Count		Symbol	Count	
1	invalid-name	19905	duplicate-code	4649		unused-wildcard-import	26307		no-member	5860	
2	line-too-long	10321	too-many-arguments	2158		bad-indentation	19921		import-error	1750	
3	missing-function-docstring	6444	super-with-arguments	1802		redefined-outer-name	2548		undefined-variable	471	
4	trailing-whitespace	4477	too-many-locals	1456		unused-import	2321		not-callable	397	
5	missing-module-docstring	2504	too-many-instance-attributes	658		arguments-differ	1678		no-name-in-module	326	
6	wrong-import-position	2286	no-else-return	509		unused-variable	986		no-value-for-parameter	168	
7	missing-class-docstring	2060	too-few-public-methods	438		attribute-defined-outside-init	962		function-redefined	100	
8	wrong-import-order	1750	no-self-use	422		unused-argument	902		unsubscriptable-object	100	
9	ungrouped-imports	367	useless-object-inheritance	351		abstract-method	841		bad-option-value	94	
10	import-outside-toplevel	285	too-many-statements	265		redefined-builtin	536		unexpected-keyword-arg	84	
11	consider-using-enumerate	256	too-many-branches	218		dangerous-default-value	447		relative-beyond-top-level	68	
12	missing-docstring	246	cyclic-import	108		reimported	322		assignment-from-no-return	27	
13	superfluous-parens	244	inconsistent-return-statements	71		wildcard-import	297		bad-super-call	21	
14	missing-final-newline	236	unnecessary-comprehension	48		logging-format-interpolation	288		redundant-keyword-arg	19	
15	multiple-statements	218	chained-comparison	46		pointless-statement	253		too-many-function-args	18	
16	trailing-newlines	176	consider-using-in	45		fixme	247		invalid-unary-operand-type	17	
17	bad-whitespace	166	simplifiable-if-expression	34		protected-access	223		no-self-argument	9	
18	unidiomatic-typecheck	78	literal-comparison	30		logging-fstring-interpolation	110		misplaced-bare-raise	9	
19	singleton-comparison	69	too-many-nested-blocks	26		f-string-without-interpolation	105		no-method-argument	6	
20	multiple-imports	53	no-else-raise	24		pointless-string-statement	101		access-member-before-definition	6	

ungrouped-imports, import-outside-toplevel and multiple-imports) were the most commonly recognised code smells in the Convention category.

**Refactor** – The most commonly recognised opportunities for refactoring pertained to duplicate code (4649), using too many arguments when defining a function or method (2158, too-many-arguments), and using an old style for calling `super` in the constructor of an inheriting class (1802, super-with-arguments), instead of using the Python 3 style where no arguments to `super` are necessary. It also shows that functions and classes are often too complex; Pylint reports 1456 functions that use too many local variables (too-many-locals), 265 that are too long (too-many-statements) and 218 that have too many branches, as well as 658 classes that have too many attributes on them (too-many-instance-attributes).

**Warning** – The most reported Warning messages, by far, are unused-wildcard-import (26307) and bad-indentation (19921). Code smells relating to import management, as already indicated in the Convention category, are also reflected in the Warning category with 26307 counts of unused wildcard imports, 2321 counts of unused imports, 322 counts of libraries that were imported multiple times in the same file (reimported) and 297 counts of wildcard imports. Aside from unused imports, unused variables (986) and unused arguments (902) are also common. Having variables that redefine (shadow) function or variable names from an outer scope (redefined-outer-scope) is also common with 2548 recognised cases, as is redefining Python’s built-in global names (536, redefined-builtin).

**Error** – Finally, in the Error category, with 5860 counts, the no-member message is the most prevalent, warning about the usage of non-existent attributes and methods on class instances and non-existent functions in Python modules. Import errors are the second most common with 1750 counts (i.e. on average 23.6 import errors per project), which are reported when a module (whether an external library or a module from a local file) contains imports that Pylint cannot resolve. The 326 no-name-in-module messages are also related to

these import problems, as they are emitted upon using a `from X import Y` style import, where X is resolved (so no import error is emitted), but Y is not found. Furthermore, the use of undefined variables and attempting to call uncallable objects are also prevalent.

## V. IMPLICATIONS

In this section, we discuss the implications of our results for ML developers. We start by elaborating upon the code smells that we have found to be most prevalent and continue with a discussion of problems regarding dependency management that we encountered while performing this research. We also argue how these problems affect the maintainability and reproducibility of the analysed ML projects.

### A. Explaining the most prevalent code smells

This section aims at providing an explanation for the prevalence of the most common code smells in our dataset of ML projects by investigating their occurrences.

**Error** – Most interestingly, we found that there were *zero* projects that had *zero* messages in the Error category. Only the *geomancer* project in the DeepMind research repository<sup>5</sup> had one error, namely a true positive no-name-in-module error message in the project’s test file.

We also found that no-member and import-error are the most reported code smells in this category. Upon manual inspection of these messages in several projects, we noticed that import errors have two primary causes, namely:

- **Bad specification of requirements** – Using the *kaggle-kuzushiji-recognition-2019* project as an example, we noticed that it was missing at least four dependencies in its (otherwise well-defined) requirements file. Imports of these missing dependencies were primarily found in the code of a dependency that the project’s authors had copied into their repository for some slight customisations, but were also found in other scripts in the repository.

<sup>5</sup><https://github.com/deepmind/deepmind-research>



- **Pylint producing false positives on local imports** – Taking the `navigan`<sup>6</sup> project as an example, even though we manually fixed the import errors relating to badly specified requirements, there were 28 errors remaining. These errors come from unresolved imports from local modules, i.e., Python files in the repository. In Python, a local module `utils.py` can be imported from other modules in the same directory using `import utils` and it is recommended (but not necessary) to add an `__init__.py` file to that directory to indicate that it is a Python package [15]. However, as a GitHub issue reports<sup>7</sup>, Pylint produces false positive import errors on local imports, but strangely *not* when the `__init__.py` file is *not* present.

As for `no-member` errors, in the `kaggle-kuzushiji-recognition-2019` project – which has 327 of them – these were primarily caused by false positives from Pylint on the majority of – if not all – usages of the `torch` library (i.e. `PyTorch`), including those of basic `PyTorch` functions like `torch.as_tensor`, `torch.tensor` and `torch.max`. The project with the most `no-member` errors, `RSNA-STR-Pulmonary-Embolism-Detection` (1549), showed the same trend, as did `DL-unet` and several other projects that we investigated. This is a known issue that has been reported to Pylint’s GitHub repository<sup>8</sup> of which the essence goes back as far as 2013 with a similar problem in the use of `NumPy`<sup>9</sup>. The reason that is stated in these issues, is that Pylint has trouble extracting the members and type information of libraries that are backed by bindings with the C programming language.

*Pylint cannot reliably check for correct uses of import statements; both local imports, as well as imports from C-backed libraries such as PyTorch, suffer from a high rate of false positives.*

This is especially concerning in the context of ML, as the majority of ML libraries are backed by C (to make them performant). The fix that the Pylint developers propose in the relevant GitHub issues all entail (partially) disabling the `no-member` rule, implying that Pylint cannot reliably check for correct uses of C-backed libraries. This is additionally concerning in the context of ML, as it has a high degree of glue code, i.e. code that is written to coerce data in and out of general-purpose libraries [2].

Additionally, the fact that Pylint fails to reliably analyse the usage of prominent ML libraries, provides a major obstacle to the adoption of Continuous Integration (CI) in the development environment of ML systems. If a static code analysis tool produces too many false positives, it will be noisy and counterproductive [12]. Thus, other important true positives may be overlooked.

*Additionally, the fact that Pylint fails to reliably analyse whether prominent ML libraries are used correctly, provides a major obstacle to the adoption of Continuous Integration (CI) in the development environment of ML systems.*

**Warning** – In this category, we found that one project (`kaggle-rsna2019_3rd_solution`) was responsible for 13917 of all 26307 `unused-wildcard-import` messages, with 53 `wildcard-import` messages. Since `unused-wildcard-import` are emitted per unused function imported with a wildcard import, this means that there were on average 263 unused imports per wildcard import in this project. Notably, most of these messages were also (contained in) instances of duplicate code. Such unused wildcard imports pollute a module’s namespace with the names of all imported functions, meaning there is a greater chance of (accidentally) redefining an outer name. Additionally, wildcard imports may also have unintended side-effects that can be very difficult to debug.

The tendency towards using wildcard imports may stem from the prototypical and experimental nature of ML projects, combined with the fact that it is simply easier for the developer to import everything from a library and use whatever they need, rather than import functions individually. Dead experimental codepaths as found by Sculley et al. [2] of which the imports still remain, can also be a cause of bad import management.

As for the many bad-indentation messages, these were dominated primarily by DeepMind projects that were using a different convention for indentation width, namely two spaces instead of four. This is not surprising since indentation width is a preference, where the PEP8 style guide<sup>10</sup> prescribes four spaces, but others such as Google’s TensorFlow style guide<sup>11</sup> prescribe two spaces.

**Refactor** – We found that duplicate-code is the most commonly reported refactoring opportunity. Having manually inspected a random subset of these messages and where they occur, we have noticed that these are primarily caused by ML developers having multiple permutations of similar ML models to perform the same task. Each model (experimental codepath) then uses a slightly different underlying algorithm or slightly different parameters and are each defined in their own file, likely in an attempt to find the best performing one. Yet instead of identifying the commonalities between these different models and abstracting them into modules that can be reused across their codebase, ML developers seem to prefer simply copy-pasting the files. However, more research into code reuse and duplication in ML code is required to truly understand this phenomenon and how it can be prevented.

*Code duplication is common in ML, but calls for more extensive research to truly understand to what extent and for what reasons this occurs, and how it can be avoided.*

<sup>6</sup><https://github.com/yandex-research/navigan>

<sup>7</sup><https://github.com/PyCQA/pylint/issues/3984>

<sup>8</sup><https://github.com/PyCQA/pylint/issues/{3510, 2708, 2067}>.

<sup>9</sup><https://github.com/PyCQA/pylint/issues/58>.

<sup>10</sup><https://www.python.org/dev/peps/pep-0008/>

<sup>11</sup>[https://www.tensorflow.org/community/contribute/code\\_style](https://www.tensorflow.org/community/contribute/code_style)

Regarding the high prevalence of the `too-many-arguments` and `too-many-locals` messages, it is congruent with previous work which shows that data science projects contain significantly more instances of these than traditional software projects [6]. Simmons et al. [6] also notes that these messages are related: function arguments namely also count as local function variables. By default, a `too-many-arguments` message is emitted when a function or method takes more than five arguments, while `too-many-locals` is emitted when a function or method contains more than 15 local variables. A possible cause for their prevalence, as Simmons et al. [6] note, are “*models with multiple hyperparameters that are either hard-coded as variables in the function definition or passed to the function as individual parameters rather than being stored in a configuration object.*”

**Convention** – While line length violations stem primarily from developer preference, invalid and improper naming is a problem not exclusive to Python [7]. Pylint by default emits a `invalid-name` message when it finds names that do not comply to PEP8, i.e. are improperly capitalised or less than three characters long (except in the case of inline variables). Indeed, shorter identifier names do take longer to comprehend than whole words [16], but Simmons et al. [6] aptly notes that this may not necessarily be the case in DS and ML code due to its heavily mathematical background. Thus, ML practitioners may find it easier to comprehend the details of a piece of ML code written so that it resembles the notation of the underlying mathematical model, including the names of the identifiers. Future research on this subject will have to show how this affects the readability of ML code.

*The PEP8 convention for identifier naming style may not always be applicable in ML code due to its resemblance with mathematical notation. Future research is required to investigate how this affects the readability of ML code.*

### B. Problems installing project dependencies

Setting up the projects’ codebases as detailed in Section III-B was not a trivial task. While 42 out of 74 projects did have a `requirements` file in their repository that installed without a hitch out of the box, there were 32 projects where a `requirements.txt` had to be generated or had to manually be created from inspecting the repository or manually modified from what was already in the repository. Furthermore, there were 13 projects that required installing extra dependencies after installing those in the `requirements` file. One of these projects had a valid `requirements` file – and specified the extra dependencies in their `ReadMe` – but the 12 others did not.

As for the projects for which a `requirements` file had to be manually created or modified, we made a few observations as to why this was needed. First, some projects did not contain a `requirements` file at all, but did specify instructions in the `ReadMe`, e.g., the `yolact_edge` project.

Secondly, some projects had simply made a small mistake in their manual maintenance of the `requirements` file, as was

the case with the `navigan` project. The project authors fixed the mistake less than a day after we filed an issue on their GitHub<sup>12</sup> about it.

Thirdly, some projects were relying on custom Docker containers for their runtime environment. These projects, e.g. `kaggle-imaterialist`, maintain a `Dockerfile` in their repository (sometimes with an additional `requirements` file) in which the project’s dependencies are installed, often without specifying exact dependency versions.

Finally, and most commonly (especially with the Kaggle projects), projects would contain a `requirements.txt` file that was likely the result of running `pip freeze` – a shell command that lists all the packages installed in the current Python environment, including their respective dependencies, along with their exact versions. However, there are three problems with this approach:

- **Difficult to maintain** – Since `pip freeze` lists *all* direct, indirect, runtime and development dependencies, without distinction, in alphabetical order, we conjecture that it is difficult for maintainers to assess whether a certain dependency can safely be upgraded without breaking their code or breaking any of their dependencies.
- **May result in unresolvable dependencies** – The resulting `requirements` file may contain dependencies sourced from different dependency management tools and package indexes. These dependencies may have slightly different package names across package indexes or have only published certain versions to e.g. Conda’s package index, but not to PyPI. There were also projects that depended on pre-release versions of certain libraries that are no longer available on PyPI (e.g., older nightly versions of Tensorflow packages).
- **May include unrelated dependencies** – Especially if the user is not installing their dependencies into a virtual environment, then the resulting `requirements` file may also include unnecessary, unrelated (and potentially unresolvable) Python dependencies, such as those used by their operating system or those used in other projects. For example, the `side_effects_penalties` in the DeepMind research repository depends on `youtube-dl` (even though the project has nothing to do with videos), as well as some dependencies from the operating system level such as `python-apt`, `python-debian` and `ufw`. The inclusion of the latter dependencies directly indicates that the project author was not using a virtual environment, but was instead using `sudo pip install` to install all of their Python dependencies.

*We have found serious issues with the specification of dependencies that present a major threat to the reproducibility and maintainability of Python ML projects. Further research needs to be undertaken to help ML practitioners avoid issues in the dependency management of their projects.*

<sup>12</sup><https://github.com/yandex-research/navigan/issues/1>

## VI. THREATS TO VALIDITY

### A. Validity of the dataset

Our dataset may not yet be fully representative of the real-world state of ML code, as it currently only contains open-source ML projects. Therefore, in future research, we want to collect a dataset of closed-source ML projects from the industry, such as ING’s AI-driven FinTech industry. We will use this both to compare the prevalence of code smells in these closed-source industry projects with that of open-source projects as presented in this paper, as well as to make our dataset more representative of the real-world state of ML and AI projects. We will also explore adding projects from the dataset published by Biswas et al. [13].

Furthermore, we currently do not perform any analysis on the code quality of Jupyter Notebooks, even though they are very popular and have emerged as a de facto standard for data scientists [17]. This was deliberate, as Pylint currently does not support directly analysing the Python code in Jupyter Notebook files and we wanted to avoid applying double standards to pure Python code and notebook Python code by extracting the notebook code into pure Python files. However, given their popularity, we do intend to perform future research on the code quality and linting of Jupyter Notebook code.

### B. Validity of Pylint

Due to its dynamically typed nature, linting Python code is notoriously difficult [6, 11]. It is therefore no surprise that Pylint contains bugs and limitations that cause false positives and false negatives. Pylint’s issue tracker on GitHub also reports 165 open and 501 closed issues regarding false positives<sup>13</sup> as of January 19th 2021. We have also noticed some of these shortcomings for ourselves during this research, as we have discussed in Section V-A. We mitigate this threat by manually checking a subset of projects to analyse potential false positives.

## VII. CONCLUSION

In this study we investigated the prevalence of code smells in ML projects. We gathered a dataset of 74 ML projects, ran the static analysis tool Pylint on them and collected the distribution of Pylint messages per category per project (Table II), the top 10 code smells in these projects overall (Table III), and the top 20 code smells per category (Table IV).

Moreover, by performing a manual analysis of a subset of the detected smells, we have found that code duplication is common in ML, but does require further research to understand to what extent this occurs and how it can be avoided. We also found that the PEP8 convention for identifier naming style may not always be applicable in ML code due to its resemblance with mathematical notation. This calls for additional research on how it affects the readability of ML code.

Most importantly, however, we have found serious issues with the specification of dependencies that present a major

threat to the reproducibility and maintainability of Python ML projects. Furthermore, we found that Pylint produces a high rate of false positives on import statements and thus cannot reliably check for correct usage of imported dependencies, including prominent ML libraries such as PyTorch. Both of these problems also provide a major obstacle to the adoption of CI in ML systems. Further research needs to be undertaken to help ML practitioners avoid issues in the dependency management of their projects.

## REFERENCES

- [1] M. Haakman, L. Cruz, H. Huijgens, and A. van Deursen, “AI lifecycle models need to be revised. An exploratory study in fintech,” *arXiv preprint arXiv:2010.02716*, 2020.
- [2] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, “Hidden technical debt in machine learning systems,” in *Advances in neural information processing systems*, 2015, pp. 2503–2511.
- [3] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, “Software engineering for machine learning: A case study,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 291–300.
- [4] D. Gonzalez, T. Zimmermann, and N. Nagappan, “The State of the ML-Universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR ’20, 2020, p. 431–442.
- [5] T. Menzies, “The Five Laws of SE for AI,” *IEEE Software*, vol. 37, no. 1, pp. 81–85, 2020.
- [6] A. J. Simmons, S. Barnett, J. Rivera-Villicana, A. Bajaj, and R. Vasa, “A Large-Scale Comparative Analysis of Coding Standard Conformance in Open-Source Data Science Projects,” in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM ’20, 2020.
- [7] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhenec, “Code smells and refactoring: A tertiary systematic review of challenges and observations,” *Journal of Systems and Software*, vol. 167, p. 110610, 2020.
- [8] K. Tómasdóttir, M. Aniche, and A. van Deursen, “The Adoption of JavaScript Linters in Practice: A Case Study on ESLint,” *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 863–891, 2020.
- [9] S. Omari and G. Martinez, *Enabling Empirical Research: A Corpus of Large-Scale Python Systems*, 10 2019, pp. 661–669.
- [10] N. Bafatakis, N. Boecker, W. Boon, M. Cabello Salazar, J. Krinke, G. Oznacar, and R. White, “Python Coding Style Compliance on Stack Overflow,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 210–214.
- [11] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu, “Understanding Metric-Based Detectable Smells in Python Software,” *Inf. Softw. Technol.*, vol. 94, no. C, p. 14–29, Feb. 2018.
- [12] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, “How developers engage with static analysis tools in different contexts,” *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, Mar 2020.
- [13] S. Biswas, M. J. Islam, Y. Huang, and H. Rajan, “Boa Meets Python: A Boa Dataset of Data Science Software in Python Language,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 577–581.
- [14] Python Software Foundation, “Managing Application Dependencies — Python Packaging User Guide,” 2021, (accessed: 19 Jan. 2021). [Online]. Available: <https://packaging.python.org/tutorials/managing-dependencies/#managing-dependencies>
- [15] —, “6. Modules — Python 3.7.9 documentation,” 2021, (accessed: 19 Jan. 2021). [Online]. Available: <https://docs.python.org/3.7/tutorial/modules.html#packages>
- [16] J. C. Hofmeister, J. Siegmund, and D. V. Holt, “Shorter identifier names take longer to comprehend,” *Empirical Softw. Engg.*, vol. 24, no. 1, p. 417–443, Feb. 2019.
- [17] J. M. Perkel, “Why Jupyter is data scientists’ computational notebook of choice,” *Nature*, vol. 563, no. 7732, pp. 145–147, 2018.

<sup>13</sup>See <https://github.com/PyCQA/pylint/issues?q=is%3Aissue+false+positive>

# 3

## MLLINT - DETECTING PROJECT SMELLS & IMPROVING SOFTWARE QUALITY

After our findings of bad dependency management practices in open-source ML projects and furious brainstorming on what the next direction of my thesis would be, the realisation dawned that a more holistic view on code smells would be required to accurately assess the software quality of ML applications: *project smells*. Combined with my desire to deliver a practical impact for ML practitioners, this gave rise to the development of an automated tool to detect such project smells in ML projects and help ML practitioners to mitigate them.

### 3.1 IMPLEMENTATION

Thus, `mlLint` was born. `mlLint` has been open-source from the start and currently lives under a GPLv3 license in a GitHub repository<sup>1</sup> on my personal GitHub account. It also has a public website<sup>2</sup> with documentation of its linting categories and rules, built using the static site generator tool Hugo<sup>3</sup>. `mlLint` itself is published to the Python package repository PyPI<sup>4</sup> such that it can easily be included in Python ML projects, though `mlLint` is not written in Python; it is instead written in Go<sup>5</sup>. Go is particularly well-suited towards command-line applications (which I had prior experience with), is very performant (especially in comparison to Python) and is generally a very well thought out programming language with few, but extremely powerful concepts. Go applications compile to a static binary and allow for easy cross-compilation to other operating systems and processor architectures. Python's *wheel*<sup>6</sup> binary distribution format makes it possible to package a compiled Go application into a Python package with a thin wrapper to forward shell arguments to the executable, making it usable for users in the Python ecosystem. `mlLint`'s source code repository contains an automated CI script to compile `mlLint`, package it into platform-specific Python wheels and set of Docker containers and publish them to PyPI and Docker Hub<sup>7</sup>. The result is that `mlLint` runs on Windows, MacOS and Linux, and is even usable on ARM-based architectures such as Apple M1 and the Raspberry Pi.

`mlLint` primarily produces Markdown output, which it by default pretty-prints to the terminal, but can also be streamed to a file or to the standard output. Even the documentation of `mlLint`'s rules and categories is written in Markdown, which is particularly useful in the generation of `mlLint`'s documentation website. Markdown is a very versatile output format that is widely supported by a plethora of tools, IDEs and development platforms, making it the logical choice for a human-readable, consistently formatted, easily portable, structured output document.

---

<sup>1</sup> <https://github.com/bvobart/mlLint>

<sup>2</sup> <https://bvobart.github.io/mlLint/>

<sup>3</sup> <https://gohugo.io/>

<sup>4</sup> <https://pypi.org/project/mlLint/>

<sup>5</sup> <https://golang.org/>

<sup>6</sup> <https://packaging.python.org/specifications/binary-distribution-format/>

<sup>7</sup> <https://hub.docker.com/r/bvobart/mlLint>

## 3.2 PEER INTERACTIONS

Throughout the development of `mlLint`, I have been in contact with various experienced ML engineers at ING. In February, I joined the GAP squad and ML Engineering chapter in ING Global Analytics, where I took part in their standups three days per week, their weekly update meeting, their bi-weekly ML Engineering chapter meetings and monthly ML Engineering chapter days. These taught me a great deal about the daily lives and activities of ML engineers, their development culture and the technologies they are using and experimenting with to increase the quality of their ML projects. They also provided me a podium to disseminate my research, as I presented two of their ML Engineering chapter meetings: the first in February about [van Oort et al., 2021] and my goal of developing a tool for detecting adherence to ML best practices, the second in June about `mlLint`, its goals, challenges and the planned method of evaluating its efficacy.

They also helped me make contact with other ML Engineers and ML Engineering chapters within ING. Most notably, I got into contact with the two chapter leads of the ML Engineering chapters of ING Frankfurt and Madrid. I met with either of them and a few people from their teams to pitch `mlLint`, learn more about their practices of ML project quality assurance and source ML projects and survey participants for the evaluation of `mlLint`. Their impressions were positive and they both invited me to speak at one of their ML Engineering chapter meetings. Thus, in June, I presented at a combined ML Engineering chapter meeting for both ING's Frankfurt & Madrid hubs, joined by around 40 of their data scientists and ML engineers.

One of the fruits of this social tree was the contact with a senior ML engineer from ING Frankfurt who voiced his enthusiasm to contribute to the development of `mlLint`. We met and decided that he would be best suited for researching usage patterns of and designing the linting rules on Data Quality practices with tools like GreatExpectations<sup>8</sup> and TFDV<sup>9</sup>. Unfortunately, before he could add significant value, he had other matters to attend to and left ING soon after, halting the progress on the Data Quality linter. Nevertheless, `mlLint` temporarily had a contributor.

Additionally, inspired by the ML project best practices of SE4ML[Serban et al., 2021], I got in contact with members Alex Serban and Joost Visser. We discussed ML project best practices and the challenges in mapping ML best practices to practical linting rules for ML projects. During the development of `mlLint` we also aligned my milestones with monthly meetings to discuss progress on `mlLint` and provide feedback on each other's work.

Finally, for a short while in March, I followed the lectures of a PhD course from the University of St. Gallen on Software Engineering for Artificial Intelligence (SE4AI)<sup>10</sup>. These provided interesting insights into academic perspectives on AI, the SE challenges that go with it and the research to mitigate those.

## 3.3 EVALUATION EFFORTS

When it came time to evaluate `mlLint`, we used the accrued connections within ING and our academic network to disseminate `mlLint` and the evaluation survey. The timing of this was somewhat unfortunate, as it was July and the holiday season had just commenced. Combined with the diminished social relations that perpetually working from home engenders, it encumbered communication

<sup>8</sup> <https://greatexpectations.io/>

<sup>9</sup> TensorFlow Data Validation: <https://github.com/tensorflow/data-validation>

<sup>10</sup> [https://docs.google.com/document/d/1\\_yqLNJEk2yBfd2mJxMtMz493IXU6Qzcb0sIeLLWb5kw](https://docs.google.com/document/d/1_yqLNJEk2yBfd2mJxMtMz493IXU6Qzcb0sIeLLWb5kw)

with several ML engineers in ING that had previously vowed to help evaluate `mlint`. This additionally played into our decision to publically disseminate `mlint` and the survey. Mid-August, we posted a tweet<sup>11</sup> about `mlint` that garnered 31 retweets and 41 likes, as well as a Reddit post<sup>12</sup> that received 112 upvotes and flourished an interesting discussion about the potential harms and benefits of using code quality linters.

Furthermore, to help gain more survey responses from data scientists and ML engineers from ING, I hosted a workshop presentation of `mlint` at ING Analytics. The goal was to explain what `mlint` is, describe some of the research behind it, give a quick demo and then let them run `mlint` on an ML project of their own, finishing with filling the survey. Since the majority of people left without filling the survey and since several others had indicated that they were interested, but not available to join, we decided to recreate the workshop in a video<sup>13</sup> for participants to follow along in their own time.

## 3.4 OTHER INFORMATION

At the end of May, I started developing repetitive strain injury (RSI) in my wrists. I visited a physiotherapist and purchased several RSI-friendly peripherals for my daily usage. I healed through their combined efforts and natural healing, though it still strained my development efforts on `mlint` for two months.

Finishing on a positive note, `mlint` was also used to help educate future data scientists and ML engineers about Release Engineering for ML Applications (REMLA)<sup>14</sup> in the homonymous MSc course in June by Sebastian Proksch and Luís Cruz. The course contained a tutorial on version controlling data with the Data Version Control (DVC) tool. This tutorial used `mlint` to verify DVC implementation practices in their example ML project and used the documentation on `mlint`'s data version control rules to teach about correct data version control practices. Additionally, in the project part of the course, one group of students was creating an automated Repository and Pipeline Generation Tool (RPGT)<sup>15</sup> that can generate a template ML project that conforms to all of `mlint`'s linting rules and uses `mlint` for continuous quality control on CI.

11 See <https://twitter.com/Bv0Bart/status/1426095280059994112>. Retweets and likes up to date as of 9 Oct. 2021

12 See [https://www.reddit.com/r/MachineLearning/comments/p3j2xh/pr\\_announcing\\_mlint\\_a\\_linter\\_for\\_ml\\_project/](https://www.reddit.com/r/MachineLearning/comments/p3j2xh/pr_announcing_mlint_a_linter_for_ml_project/). Upvotes up to date as of 9 Oct. 2021

13 [https://www.youtube.com/watch?v=s\\_uCvRkr20g](https://www.youtube.com/watch?v=s_uCvRkr20g)

14 <https://se.ewi.tudelft.nl/remla/>

15 <https://pypi.org/project/rpgt/>

# “Project smells” — Experiences in Analysing the Software Quality of ML Projects with `mllint`

Bart van Oort  
TU Delft  
AI for Fintech Research,  
ING  
Delft, Netherlands  
bart.van.oort@ing.com

Luís Cruz  
TU Delft  
Delft, Netherlands  
l.cruz@tudelft.nl

Babak Loni  
ML Engineering Chapter,  
ING  
Amsterdam, Netherlands  
babak.loni@ing.com

Arie van Deursen  
TU Delft  
Delft, Netherlands  
arie.vandeursen@tudelft.nl

## ABSTRACT

Machine Learning (ML) projects incur novel challenges in their development and productionisation over traditional software applications, though established principles and best practices in ensuring the project’s software quality still apply. While using static analysis to catch code smells has been shown to improve software quality attributes, it is only a small piece of the software quality puzzle, especially in the case of ML projects given their additional challenges and lower degree of Software Engineering (SE) experience in the data scientists that develop them. We introduce the novel concept of *project smells* which consider deficits in project management as a more holistic perspective on software quality in ML projects. An open-source static analysis tool `mllint` was also implemented to help detect and mitigate these. Our research evaluates this novel concept of project smells in the industrial context of ING, a global bank and large software- and data-intensive organisation. We also investigate the perceived importance of these project smells for proof-of-concept versus production-ready ML projects, as well as the perceived obstructions and benefits to using static analysis tools such as `mllint`. Our findings indicate a need for *context-aware* static analysis tools, that fit the needs of the project at its current stage of development, while requiring minimal configuration effort from the user.

## KEYWORDS

project smells, software quality, machine learning, `mllint`, code smells, context-aware, static analysis, dependency management, Python

## ACM Reference Format:

Bart van Oort, Luís Cruz, Babak Loni, and Arie van Deursen. 2021. “Project smells” — Experiences in Analysing the Software Quality of ML Projects with `mllint`. In *Proceedings of ICSE 2022 - Software Engineering in Practice (ICSE 2022 – SEIP)*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ICSE 2022 – SEIP, May 21–29, 2022, Pittsburgh, PA, USA

© 2021 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$0.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

The ubiquity of Machine Learning (ML) and Artificial Intelligence (AI) solutions to complex computing problems demands development processes to help transform a proof-of-concept ML experiment into a well-engineered ML application, running continuously in a production environment [3, 8, 12, 20]. These development processes on the one hand incorporate novel ideas to deal with the novel challenges that developing ML applications poses over developing traditional software applications, but on the other hand also include established Software Engineering (SE) best practices. After all, quoting Carleton et al. [6], “An AI system is a software-intensive system, and the established principles of designing and deploying quality software systems that meet their mission goals on time still apply”.

However, productionising is difficult, especially in the case of ML systems given their additional challenges, such as data management, testing and reproducibility [10, 14]. Traditional software engineering historically struggled with this too, but has seen the implementation of a host of tools to help with productionisation in various stages of the software development lifecycle. For example, using static analysis to enforce best practices and catch code smells, helps catch bugs earlier and improve software quality attributes, such as reliability, maintainability and reproducibility [11].

Especially in ML projects, though, code smells are only a small piece in the software quality puzzle. We noticed this first-hand in our previous research on the prevalence in code smells in open-source ML projects: nearly half of the analysed projects struggled with managing their code dependencies [23]. We realised that a more holistic approach to code smells, ‘*project smells*’, would be required.

To the end of automatically detecting such project smells and giving practical advice on how to fix those, we implemented `mllint`. `mllint`<sup>1</sup> is an open-source command-line utility to evaluate the software quality of Python ML projects by performing static analysis on the project’s source code, data and configuration of supporting tools. `mllint` aims to help ML practitioners in developing and maintaining production-grade ML and AI projects.

We argue that many data-driven companies may benefit from an SE for ML tool such as `mllint`. This research gauges how well these project smells as detected by `mllint` fit the context of ML development at our industrial partner ING. ING is a global bank and large software- and data-intensive organisation with a strong European base that offers retail and wholesale banking

<sup>1</sup><https://github.com/bvobart/mllint>



services to 38.5 million customers in over 40 countries [9] and has 15.000 employees in IT, software and data technology [1]. ING has extensive use-cases for increasing its business value with AI and ML, such as assessing credit risk, fighting economic crime by monitoring transactions and improving customer service. As part of a major shift in the organisation to adopt AI and ML and become data-driven, ING is defining standards for the different processes around the lifecycle of ML applications [8].

To measure the fit of project smells in this context, we qualitatively analyse the reports generated by `mlLint` on ING projects and combine them with feedback from ML practitioners. Additionally, we asked ML practitioners to run `mlLint` on their projects and provide us with their feedback on their experiences with `mlLint` and its concepts. By doing so, we aim to also uncover the obstacles towards implementing specific best practices, as well as the perceived benefits and drawbacks of using static analysis tools such as `mlLint` to verify SE practices in ML projects. Additionally, we investigate how ML practitioners perceive the difference in importance of `mlLint`’s linting rules on proof-of-concept projects versus production-ready projects, as the former may not require as rigorous software quality checks as the latter do.

More formally, our research questions are as follows:

- RQ1** How do the project smells as detected by `mlLint` fit the industrial context of a large software- and data-intensive organisation like ING?
- RQ2** What differences do ML practitioners perceive in the importance of `mlLint`’s linting rules between proof-of-concept and production-ready projects?
- RQ3** What are the main obstacles for ML practitioners towards implementing specific best practices?
- RQ4** What are the perceived benefits of using static analysis tools such as `mlLint` to verify SE practices in ML projects?

The rest of this paper is structured as follows. Section 2 describes influential research in the field of SE for ML that served as the background for this research. Section 3 elucidates the concept of *project smells* and `mlLint`, detailing two major challenges in its development. Section 4 explains the methodology used to answer our research questions. In Section 5, we present the findings from applying our methodology and answer our research questions. Section 6 then combines and discusses these findings along the themes of version controlling data, dependency management and static analysis tool adoption. We then discuss the threats to the validity of our research in Section 7 and conclude with an outlook on future work in Section 8.

The contributions of this research are as follows:

- The novel concept of *project smells* as a holistic perspective on software quality in ML projects.
- An open-source static analysis tool `mlLint`<sup>2</sup> to help with detecting and mitigating these project smells.
- Experiences, insights and perceptions on project smells in an industrial context.

## 2 BACKGROUND

Both Software Engineering and Machine Learning are well studied in literature, though their intersection is still an emerging field of research [3, 12, 14].

Sculley et al. [15] were among the first to investigate risk factors in the design of real-world ML systems at Google through the lens of technical debt. In doing so, they unearthed several anti-patterns in ML system design, including glue code—the tendency for ML applications to consist of code that glues together functionalities from various general-purpose libraries—and configuration debt—the tendency for both researchers and engineers to see configuration and configurability of the ML application as an afterthought [15]. Continued research at Google investigating production-readiness and the reduction of technical debt in ML systems, resulted in “*The ML Test Score*” [5]: a rubric with 28 specific tests and monitoring needs, along with a scoring system to determine the production-readiness of ML systems. These tests are split into four categories, namely *Data*, *Model*, *Infrastructure* and *Monitoring*, with each category containing seven tests. For every test, half a point is awarded for executing the test manually, documenting and distributing the results. A full point is awarded if that test is automated and runs regularly. The scores are then summed per category and the final production-readiness score is calculated by taking the minimum of these category scores. A score between 3 and 5 is interpreted as “*Strong levels of automated testing and monitoring, appropriate for mission-critical systems.*” [5]

Amershi et al. [3] at Microsoft also used experiences from engineering ML applications in their case study. Their study resulted in several best practices and three aspects of engineering ML / AI applications that makes them fundamentally different from traditional software applications. One such aspect is concerned with the discovery and management of data: ML applications also need to deal with finding, collecting, cleaning, curating and processing their input data. This data also needs to be stored and versioned, for which in contrast to code there were no well-designed technologies to do so [3]. Another challenge lies in the customisation and reuse of ML models on problems in different domains or with slightly different input formats, as this may require retraining or even replacing the model with new or additional training data. Finally, Amershi et al. [3] state that strict modularity between ML models is difficult to achieve, as models are not easily extensible and multiple models may interact with each other in unexpected ways. Kriens and Verbelen [10] recognise this and propose a partial solution in the form of OSGi-like metadata for ML models.

The aforementioned challenges are reflected in systematic literature reviews (SLRs) such as [14], [24] and [2]. Nascimento et al. [14] analysed the limitations and open challenges found in the SE for ML field of research, noting that testing, AI software quality and — again— data management are three of the main challenges faced by professionals in the field. They also report on several SE practices, approaches and tools for dealing with these challenges. On the topic of testing ML systems, Zhang et al. [26] performed an extensive SLR of various techniques to do this. Washizaki et al. [24] similarly performed an SLR on SE design patterns for ML techniques, identifying several good and bad patterns for engineering ML software. Muralidhar et al. [13] also identify MLOps anti-patterns. More

<sup>2</sup><https://github.com/bvobart/mlLint>



recently, Alamin and Uddin [2] conducted an in-depth literature review, resulting in a taxonomy of different quality assurance challenges for ML software applications, which includes dealing with data dependencies and ML-specific technical debt. Bogner et al. [4] further investigates technical debt in ML systems, identifying new forms of such debt, 72 anti-patterns (most of them relating to models and data) and 46 potential solutions to them.

Our research also builds on the work of SE4ML [19], who have identified 45 best practices for engineering trustworthy ML applications [16, 17, 22]. They also measured the adoption of these best practices under practitioners, both in academic and industry use [16]. Among others, their findings indicate that larger teams tend to adopt more best practices and that traditional software engineering practices tend to have a lower adoption than ML-specific best practices. More recently, they also studied challenges and solutions in an SLR about software architecture for systems with ML components [20]. Along with new ML-specific challenges, they also found that traditional software architecture challenges also play an important role in architecting ML systems.

Lastly, in our previous work, we analysed the prevalence of code smells in ML projects [23]. Aside from widespread code duplication in ML projects and several false positives in Pylint, we coincidentally found that nearly half of the projects we analysed struggled with dependency management, so much so that manual adjustments were needed to allow error-free installation of the Python libraries that they used. This severely hurts the maintainability and reproducibility of these projects.

### 3 MLLINT

Following from the related work, a pattern emerges suggesting that code smells are only a small factor in the overall software quality of ML applications. Code smells do “*have a strong relationship with quality attributes, i.e., with understandability, maintainability, testability, complexity, functionality, and reusability*” [11], but they do not paint the complete picture, especially in ML applications given the extra challenges in their development over traditional software applications.

Thus, to more accurately assess the software quality of ML applications, a more holistic approach would be required, where instead of *code smells*, we analyse *project smells*. Such project smells are concerned with deficits in how an ML project is managed, including poor dependency management (as outlined in [23]), lack of version control for code or data, lack of unit testing, lack of proper Continuous Integration (CI) configurations, or a lack of effective static analysis tooling. Code smells are also a subcategory of project smells.

To the end of automatically detecting such project smells and giving practical advice on how to fix those, we implemented `mllint`. `mllint` is a command-line utility to evaluate the software quality of ML projects written in Python by performing static analysis on the project’s source code, data and configuration of supporting tools. The aim of `mllint` is threefold. First, `mllint` aims to help data scientists and ML engineers in creating and maintaining production-grade ML and AI projects, both on their own personal computers as well as on CI. Secondly, it aims to help ML practitioners inexperienced with SE techniques explore and make effective

use of battle-hardened SE for ML tools in the Python ecosystem. Finally, `mllint` aims to help ML project managers assess the quality of their ML projects and receive recommendations on what aspects of their projects they should focus on improving.

### 3.1 Implementation

`mllint` analyses a project with linting rules in five categories, which roughly correspond to the project smells as previously outlined. These categories are based on well-known SE practices in traditional software application development, as well as SE for ML best practices from (grey) literature, such as SE4ML’s collection of best practices [17, 19] and Google’s Rules for ML [27]. Each category is described as follows.

**Version Control** This category comprises both version controlling source code (with Git), as well as version controlling data. The latter is particularly relevant to ML applications.

**Dependency Management** This category entails checking whether the project manages its code dependencies (e.g. used libraries) in a reproducible and maintainable manner, to mitigate the dependency management issues found in [23].

**Continuous Integration** The rule in this category checks whether the project has a CI configuration file. It should still be extended with rules to check whether the configuration is valid and to what degree the CI configuration uses appropriate tooling to continuously verify the project’s software quality.

**Code Quality** This category is concerned with code smells and runs a set of linters (Pylint, Mypy, Black, `isort` and Bandit) to detect and help mitigate them. It still needs to be extended with tools for detecting ML-specific code smells, such as `dslinter` [7].

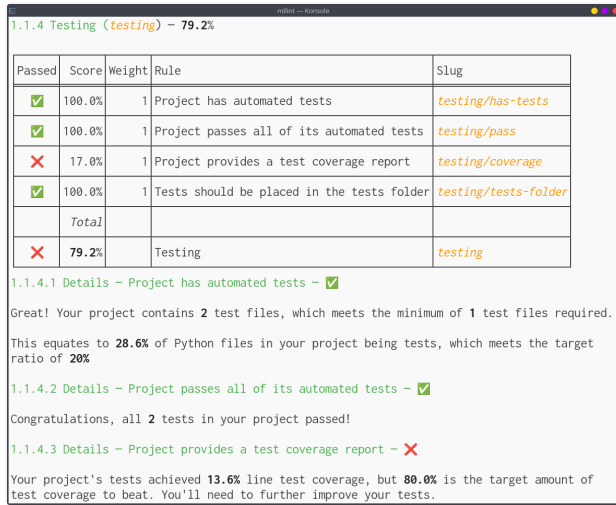
**Testing** This category analyses testing practices in the project by counting the number of test files, the number of tests passed and the test coverage. Since `mllint` performs *static* analysis, it will not run the tests, but instead expects a test- and coverage report from a prior test run.

Each category contains linting rules that analyse and score how the best practice referred to by the category is implemented in the project. For example, the *Version Control* category contains rules such as “*Project uses Git*”, “*Project should not have any large files in its Git history*” and “*Project uses Data Version Control*”. The checks imposed by these linting rules are based upon prevalent tooling and usage techniques found in the industry.

Additionally, users can define custom rules in their `mllint` configuration by referring to a custom program that performs this custom check and returns the resulting score and corresponding details. This allows users to implement their own linting rules for verifying internal team or company practices. Additionally, it can help with prototyping new rules for `mllint` before they are included in `mllint`’s core set of linting rules.

After its analysis, `mllint` outputs a Markdown-formatted report that is by default pretty-printed to the terminal. This report contains a score for each rule (between 0 and 100%), often along with details that explain the score, provide extra information derived from the analysis and / or provide recommendations on how to make the rule

<sup>3</sup><https://github.com/bvobart/mllint/blob/main/docs/example-report.md>



Passed	Score	Weight	Rule	Slug
✓	100.0%	1	Project has automated tests	testing/has-tests
✓	100.0%	1	Project passes all of its automated tests	testing/pass
✗	17.0%	1	Project provides a test coverage report	testing/coverage
✓	100.0%	1	Tests should be placed in the tests folder	testing/tests-folder
Total				
✗	79.2%		Testing	testing

1.1.4.1 Details - Project has automated tests - ✓  
Great! Your project contains 2 test files, which meets the minimum of 1 test files required.  
This equates to 28.6% of Python files in your project being tests, which meets the target ratio of 20%

1.1.4.2 Details - Project passes all of its automated tests - ✓  
Congratulations, all 2 tests in your project passed!

1.1.4.3 Details - Project provides a test coverage report - ✗  
Your project's tests achieved 13.6% line test coverage, but 80.0% is the target amount of test coverage to beat. You'll need to further improve your tests.

**Figure 1: Example snippet from an mllint report rendered to the terminal. The full report can be found on GitHub.<sup>3</sup>**

pass. For an example of such a terminal-rendered mllint report, see Figure 1.

The experienced practitioner might note that mllint in its current state primarily focuses on project smells that are also applicable to non-ML projects, but has few rules that specifically apply only to ML projects. There are two reasons for this: first, such more general SE tools and techniques (e.g. dependency management and linting for code quality) are more alien to data scientists than ML-specific tools, given the low degree of SE experience in data scientists. Secondly and more practically, given our limited amount of development resources for mllint, we have yet to implement more ML-specific linting rules.

In the following two sections, we explain two of the challenges faced in architecting mllint and its rules, along with how we approached them. Note that mllint is currently a research prototype. There are still many linting rules to implement and a host of ML project tools and architectures to support. As such, mllint is yet to reach its full potential, though it does pave the way towards using static analysis techniques to improve the quality of ML projects. mllint is built with an extensible architecture so that the list of supported practices can continuously be extended.

### 3.2 Challenge 1: Mapping high-level best practices to practical guidelines

The SE for ML best practices found in academic sources such as [17, 19] tend to be quite high-level: they explain a concept or technique for a project to adhere to, but often provide little direct, practical recommendations on how to implement it correctly. An example of this is the best practice to use static analysis tools for checking code quality [18], which does not recommend any specific linting tools to employ or what kinds of linting rules to enable – in part, to remain timeless and general. But especially within the plethora of language-supporting, supplementary tools and libraries that exists within

the Python ecosystem, it can be very difficult and time-consuming to find the right tools or configuration.

The aim of mllint is therefore to give practical advice to its users; concrete tools, techniques and guidelines that the user can implement such that the SE for ML best practices are fulfilled, along with automated checks to detect the degree of adoption. However, figuring out which exact tools to advocate for implementing which best practice is non-trivial, especially as programming ecosystems change. This is best done by looking at what tools are prevalent and popular in the industry.

Additionally, mllint is an automated, command-line tool, so based only on the project’s source code (e.g. the contents of its Git repository), it has to be able to reliably computationally check whether the project adheres to each practice. This makes it difficult, in some cases impossible, to verify whether certain team- or governance-related best practices are being upheld. Two examples of this are “Establish Responsible AI Values” and “Perform Risk Assessments” [19]. These are team or company processes that a source code analysis tool such as mllint will not be able to enforce.

To find appropriate linting rules and practical advice, we started by exploring the practical side of the SE for ML landscape and how their best practices relate to practical implementations. This was done by estimating the measurability of the best practices from SE4ML [17, 19] and Google’s Rules for ML [27]. For each practice, we explored ways to detect adherence to it in the source code of an ML project, how reliable such an approach would be and, by extension, how feasible it would be to reliably and accurately measure adherence to this practice. Our indication of measurability was given as one of five colours between red (not measurable), yellow (technically measurable, but likely to be unreliable or inaccurate) and green (measurable in a reliable and accurate way).

As an example, consider the best practice to use Continuous Integration [19]. This was marked yellow, as it is easily possible to detect whether a project has a CI configuration in its repository – and CI configurations are also machine-readable – but it is difficult to determine whether this configuration contains an appropriate set of CI jobs for the project. By contrast, the best practice “Check that Input Data is Complete, Balanced and Well Distributed” [19] was marked green, since this data should be available through the software repository and only requires a few statistical checks on the data, possibly through tools like GreatExpectations<sup>4</sup> or TensorFlow Data Validation<sup>5</sup>. Finally, the aforementioned best practice to “Establish Responsible AI Values” was marked red, as this is a team / organisational value that cannot be deduced from the project’s software repository.

After this measurability analysis, we simply picked the low-hanging fruits, i.e., the most measurable, yet also easy to implement best practices to become our first best practices. An iterative approach was then taken in constant collaboration with experienced ML engineers from ING to determine which best practices were most useful next.

<sup>4</sup><https://greatexpectations.io/>

<sup>5</sup><https://github.com/tensorflow/data-validation>

### 3.3 Challenge 2: Heterogeneity of ML projects

Another big challenge for `mlLint` comes from the many different kinds of ML projects. An ML project could be plain-old Python using basic ML libraries, but could also be based on a framework like TensorFlow or PyTorch, for each of which a different project architecture and tooling might be preferable. If a company has their own ML infrastructure or platform, then this could also impose different requirements to the project’s layout and tooling. Furthermore, if the company uses proprietary tools for fulfilling certain practices, `mlLint` may not recognise them and will not be able to assess whether the best practice is followed correctly, resulting in incorrect recommendations. All in all, some linting rules may not make sense on certain projects, or need to adapt what they recommend for different kinds of projects.

Aside from the technical differences, ML projects may also differ in maturity. An ML project that is only a proof of concept does not need to be as highly engineered, reproducible and maintainable as a production-ready or fully productionised project. Surely, it should get its basics right, as the best practice “*Keep the first model simple and get the infrastructure right*” (rule #4 of Google’s Rules of ML [27]) also endorses, but as an example, it may not be worthwhile fixing all linter warnings or achieving full test coverage. The more mature the project, the more important these engineering principles become though. Since `mlLint`’s recommendations may steer the engineering process, `mlLint` should account for differences in maturity, by adjusting the weights of its rules to match what is important to the project at the current stage of development. This paper therefore also investigates the perceived differences in the prioritisation of each of `mlLint`’s rules.

Finally, there will always be tools, techniques and practices that `mlLint` will not recognise or have linting rules for, such as proprietary tools and internal company / team practices. To provide some degree of support for such cases, `mlLint` allows users to define custom rules in its configuration that run some arbitrary script or program to score and provide recommendations on a custom practice. Such custom rules also provide a testing ground for new linting rules that may later be published as a plugin to `mlLint`, or even be built into `mlLint`.

Summarising, the challenge of heterogeneity of ML projects to tools like `mlLint` is still an open challenge. However, our proposals to solving it may at least limit its impact. These include configurability of enabled rules (with sensible defaults), automatic adaptation of linting rules to different kinds of technology stacks present in projects, and custom linting rules.

## 4 METHODOLOGY

To answer the research questions posed in the introduction, we employed a mixed-methods approach. An overview of this is displayed in Figure 2. First, we gathered and qualitatively analysed the `mlLint` reports of eight ML projects at ING. Secondly, we asked and encouraged ML practitioners from ING and open-source communities alike, to try `mlLint` on one or more of their projects and evaluate the reports that it produced. Subsequently, we ran a survey with 22 users of `mlLint` to evaluate the efficacy of the tool, as well as gather insights on how users prioritise each of the implemented rules. We also used insights from informal, open-ended interviews

performed with ML practitioners within ING, partially instigated by a desire for deeper elaboration on some of the survey participants’ answers.<sup>6</sup>

### 4.1 Qualitative analysis of `mlLint` reports

To help answer RQ1, we gathered and qualitatively analysed the reports that `mlLint` generated for eight ML applications within ING. For seven of these projects we got access to the source code and ran `mlLint` on it ourselves. For the other, we asked one of its developers to run `mlLint` on their project themselves and forward us the report, which we then discussed with the developer.

The analysis of the reports then consists of three stages. First, we manually browse through each report to inspect the scores and details for each linting rule, assessing what project smells were detected and how deeply ingrained in the project they are. Are they simple oversights during development? Are they `mlLint` false positives? Were the developers unaware of the advocated best practice so far? Does the project show evidence of them applying the best practice, but in way different to what `mlLint` recognises, or did the developers actively choose not to implement the best practice being advocated by the rule? Whenever uncertain, we ask the project’s developers for further clarification and verification of these ideas.

Secondly, combining the insights taken from multiple reports, we try to deduce patterns in the prevalence of the detected project smells. Which project smells are most and least often detected? Are there any project smells that are systemically ignored, or do the developers have suitable alternatives for these practices? This results in a list of key ideas and insights about `mlLint`’s project smells.

Finally, in order to validate the findings and gauge the significance of each of the key ideas taken from stage 2 within the context of ING, we discuss the findings and key ideas with the other authors and experienced ML engineers from ING.

### 4.2 Survey

Based on our research questions, we designed a survey to evaluate the efficacy of `mlLint` as a tool and gather insights on the prioritisation of each rules. The survey starts by gathering demographics such as the participant’s profession, the type of organisation they work in, their team size & composition and their experience in the fields of SE and ML. We then asked participants about their experiences with `mlLint` by asking about their first impressions, the amount of `mlLint` recommendations they have or would apply to their ML projects, and how much they agree or disagree with a few statements on how helpful the descriptions of `mlLint`’s rules are and whether they would consider employing `mlLint` in their ML project development and code review process.

Next, in separate questions, we ask what benefits and drawbacks static analysis tools such as `mlLint` have for the participant in validating SE practices for ML projects, which we use for answering RQ4 and RQ3 respectively. We also ask questions about rules that the user disabled or was not able to implement and what was obstructing them in doing so, which we also use for answering RQ3. Furthermore, we ask participants what features or rules they

<sup>6</sup>While the survey was anonymous, the participant could fill in an email address for us to contact if answers were found to be either unclear or particularly interesting.

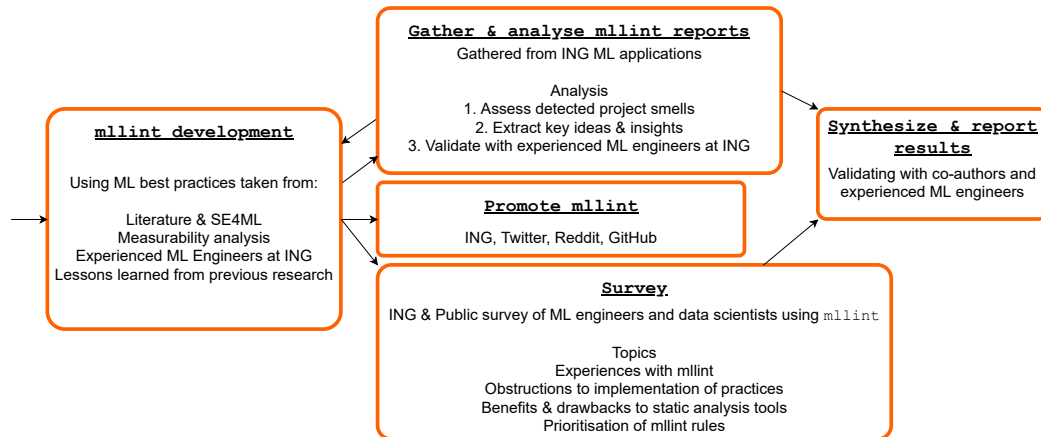


Figure 2: Overview diagram of the methodology used in this paper.

think `mllint` is still missing and what features they think could be improved.

In the final part of the survey, to answer RQ2, we ask participants to rate the importance of each of the linting rules currently implemented in `mllint`. Possible answers are on a Likert-scale, ranging from ‘*Not important*’ to ‘*Absolutely Essential*’, with the addition of an ‘*I don’t know*’ option. Since the priority of each rule may be different in different stages of the lifecycle of an ML project, we ask the participant to do this for both a *proof-of-concept* and a *production-ready* ML project. Since it may not be entirely clear what these terms entail, we provide the user with the following definitions:

**Proof-of-concept project** A project that primarily serves as an example to show that the concept behind the project works and will scale. Imagine that this is to show your supervisors that it is worthwhile to further develop this project into one that can eventually be deployed to the production environment.

**Production-ready project** A project that is mature enough to be deployed to the production environment (or already is). This requires rigorous project quality standards, such that the application is stable and will behave as expected.

For open-ended questions, we codified each of the answers by going through each answer, sentence by sentence, marking the topics that they discuss and denoting their sentiment towards it; are they being positive or negative, or listing advantages, disadvantages or caveats to take into account? We also took note of any specific, insightful remarks that the answers made. As an example, the phrase “*mllint provides a good checklist of things to do to improve ML project quality*” would be marked as having a positive sentiment and tagged with ‘*quality checklist*’, ‘*project quality*’ and ‘*guides planning*’.

The survey was spread among ML practitioners at ING through their data science and data engineering mailing lists and Slack channels and the AI for FinTech Research Lab. It was also presented at ING’s ML engineering chapters and two workshop sessions were held at ING Analytics, one live and one pre-recorded. Furthermore, we publicised `mllint` and a public copy of the survey through our academic network, a tweet and a Reddit post. The discussion

that the Reddit post triggered, as well as notes from open-ended discussions conducted with experienced ML engineers from ING after the survey, were codified similarly to the survey answers and used to answer RQ3 and RQ4.

In total, 22 people filled in our survey, most of them ML engineers, of which one chapter lead ML Engineering, two chapter leads Data Science, two ML researchers and two PhD students. 14 participants work at ING, 4 at some other non-tech company and 4 at a university or non-commercial research lab. On average, participants had between 2 to 6 years of experience at developing ML applications and between 4 to 10 years of experience in Software Engineering (defined in the survey as “*designing, implementing, testing and maintaining complex software applications*”). They tend to work in teams of 6 to 9 members, of which on average 4 have a strong background in SE.

Most participants noted they had used `mllint` on one project at the time of filling in the survey, only three participants had run `mllint` on two to five projects. Initial impressions are overall positive, with participants noting that the terminal interface is pretty and that the reports are well-organised, as well as “*informative to people unfamiliar with ML tooling and/or Python workflows*”. One participant even noted the tool “*should be a standard on ML projects*”. However, participants also noted that `mllint` is still early in development and that some were overwhelmed with the amount of terminal output, especially in the presence of many code quality linter messages.

## 5 RESULTS

Applying our methodology, we found the following answers to our research questions.

### 5.1 RQ1: How do the project smells as detected by `mllint` fit the industrial context of a large software- and data-intensive organisation like ING?

In total, eight ML projects at ING were analysed. Four of these were proof-of-concept projects, two projects were production-ready, one project was in the process of being made production-ready and

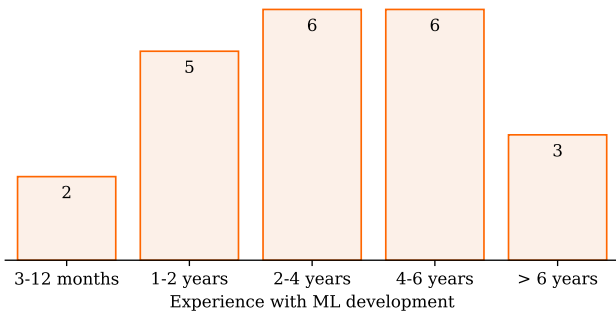


Figure 3: Countplot of ML experience among our survey participants.

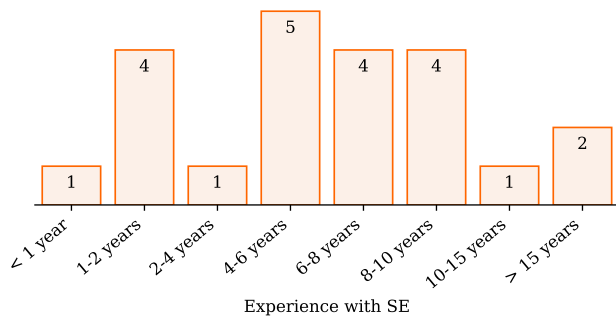


Figure 4: Countplot of SE experience among our survey participants.

one project was an example project. Listing by `mlLint` category as described in subsection 3.1, our key findings and observations are as follows<sup>7</sup>.

#### Version Control

Every project was using Git to version control their code. Three projects had large files in their Git history, some of it training data, some of it large Jupyter Notebook files. However, none of the projects that we analysed were version controlling their data using the Data Version Control (DVC)<sup>8</sup> tool, though it is known that *some* projects at ING do use it. Data acquisition methods differ per project: some receive it at run-time, one had scripts to retrieve the data from an external database, some instructed the user to download the data from an internal document sharing platform.

#### Dependency Management

Dependency management was done well in two projects, in one project not at all and in other projects with a combination of `requirements.txt` and `setup.py`, of which `mlLint` doesn’t recognise whether it is used in an effective, maintainable and reproducible way. Manual inspection showed that these projects do groom their `requirements.txt` files, there was no evidence of direct `pip freeze` usage as was prevalent in [23] and some of these projects were neatly separating their runtime dependencies from development dependencies. However, there were also two projects that duplicated the contents of their `requirements.txt` in their `setup.py`.

#### Code Quality

The example project and (being made) production-ready projects

<sup>7</sup>We intentionally omit the CI category, as its implementation in `mlLint` has a false positive.

<sup>8</sup><https://dvc.org/>

adopt static analysis tools to lint for code smells, as indicated by instructions to run linters in the documentation or linter configurations in their repositories. These projects are not free of code smells though, as particularly Pylint was eager to complain, though it is disputable what degree of its messages were false positives or irrelevant. The other proof-of-concept projects, however, were not using static analysis tools, as shown by their lack of linter configuration, lack of linter usage instructions and abundance of detected code smells.

#### Testing

The two production-ready projects and example project have automated tests that also pass. Two proof-of-concept projects had varying amounts of tests, but some of them fail due to import errors<sup>9</sup>. The other three projects, including the one being made production-ready, did not have any tests.

### 5.2 RQ2: What are the differences between perceptions on `mlLint`’s linting rules for proof-of-concept versus production-ready ML projects?

From our survey, we have gathered the following results, as listed by linting rule (sub-)category. For the average importance, we encoded our five Likert-scale answers to integers between -2 and 2, and took the mean of the responses. For the range of importance, we subtracted and added the standard deviation from / to the mean, then rounded to the nearest integer, mapping back to a Likert-scale answer.

#### Version Control – Code

For both proof-of-concept as well as production-ready projects, survey participants on average find usage of Git in ML projects between *moderately important* and *absolutely essential*, averaging *very important*. For production-ready projects, usage of Git is even unanimously seen as *absolutely essential*.

#### Version Control – Data

The importance of the rules on the (correct) use of DVC is disputed: for proof-of-concept projects, survey participants find this between *not important* and *very important*, averaging to *slightly important*. For production-ready projects, survey participants find this between *slightly important* and *absolutely essential*, averaging to *very important*. Note, however, that the rules in this category primarily relate to the usage of the tool DVC, rather than the actual practice of version-controlling data, for which there exist many other options besides DVC.

#### Dependency Management

For proof-of-concept projects, the use of proper dependency management tooling is found to be between *slightly important* and *absolutely essential*, averaging *very important*. While all other rules on proof-of-concept projects were lowest rated as *not important*, this rule was the only rule to be lowest rated as *slightly important*. For production-ready projects, this rule was rated between *very important* and *absolutely essential*, averaging *absolutely essential*, with the lowest rating being *moderately important*.

For both types of projects, however, the importances of using a single dependency manager and making a correct distinction

<sup>9</sup>Note: this may also be caused by a misconfiguration on our end, though where available we did diligently follow instructions in the repository for running the tests.

between runtime and development dependencies, was disputed. The former was rated between *slightly* vs. *moderately important* and *absolutely essential*, averaging *moderately* vs *very important*. The latter was rated between *not* vs. *slightly important* and *very important* vs. *absolutely essential*, averaging *moderately important*.

#### Continuous Integration

For proof-of-concept projects, the use of CI was rated between *slightly* and *very important*, averaging *moderately important*. For production-ready projects, this was rated between *moderately important* and *absolutely essential*, averaging *very important*.

#### Code Quality

The recommendation to use code quality linters does see a significant shift in importance between proof-of-concept and production-ready projects. For a proof-of-concept project, our survey participants rate this between *not important* and *very important*, averaging *moderately important*. For a production-ready project, they rate this between *moderately important* and *absolutely essential*, averaging *very important*.

As for the actual linting tool being used, there is no significant difference in the perceived importance. There is a slight tendency towards the code formatting tool Black in proof-of-concept projects and towards the security-focused linter Bandit in production-ready projects. Overall, we find that the *usage* of code quality linters is more important than a total absence of linter warnings.

#### Testing

The importance of having automated tests in a proof-of-concept ML project is disputed and perceived to be between *slightly* and *very important*, averaging *moderately important*. For a proof-of-concept project, however, their importance is significantly higher, between *moderately important* and *absolutely essential*, averaging *very important*. Passing the tests and having a test coverage report is also seen as *moderately* vs. *very important*.

### 5.3 RQ3: What are the main obstacles for ML practitioners towards implementing specific best practices?

Survey participants noted that out of `mlint`’s rules, they were most obstructed in implementing the practices about code quality linters and dependency management.

*“Linters, specially regarding code quality, can be overwhelming if not properly configured. Not all warnings pointed are necessarily bad for your code, not all justified warnings are equally bad, so it needs to be used parsimoniously.”*

Regarding code quality, survey participants complained that linters generally suffer from a high degree of false positives and that configuring these linters is often cumbersome and time-consuming. They experience a catch-22 situation: on the one hand, using the default configuration leaving all rules enabled, in many cases results in an overwhelming amount of linter warnings that in the eyes of the user often do not relate to the project in a functionally meaningful way (e.g., trailing whitespace and proper docstring formatting, but also false positive type-checking errors). On the other hand, selectively enabling or disabling linting rules by configuring each linter for the project, is found to be time-consuming, difficult and cumbersome, especially for those inexperienced with the tool or the kind of linting rules and their importance to the quality of the

project. This is especially found to be difficult in a team situation, as each developer may have different preferences / opinions about specific linting tools and rules.

Regarding dependency management, while `mlint` recommends using Poetry or Pipenv, some survey participants note that they prefer to stick with Python’s standard `requirements.txt` and `setup.py` files. While they do acknowledge that these are easy to misuse, especially for those inexperienced with them, several arguments are made for them. First, they argue that a disciplined developer or team can still use Python’s standard tooling in an effective, maintainable and sufficiently reproducible manner, especially when combined with Docker. Secondly, if they are already sufficiently proficient at this, they do not want to re-learn to do with Poetry what they can already do with Pip. Thirdly, they note that they do not want to be forced to use external tooling (outside of Python) to interact with a project. Finally, they note that Poetry may conflict with other tools they are using in the project, such as Versioneer. Solving such conflicts creates extra overhead and may be further complicated by a smaller user base as opposed to standard Python tools to help with solving these conflicts.

Generalising with other mentions of less specific obstructions, we find that:

- (1) Configuration of tools, especially static analysis tools, is perceived as difficult, cumbersome and time-consuming.
- (2) False positives are a significant obstruction to the adoption of certain tools, especially static analysis tools. They produce noise and may distract the user towards irrelevant or trivial issues, which results in overhead to selectively ignore them or adjust the configuration of the tool to match their preferences.
- (3) While some tools are recognised to be useful to inexperienced practitioners in reducing mistakes, experienced practitioners prefer to use tooling they are already used to, instead of having to learn to use a new tool. They may also be hesitant to add more tools to a tool stack they already deem sufficient.
- (4) New tooling may conflict or have unexpected interactions with tooling that is already in use. Solving these problems causes extra overhead for practitioners. The expectancy of such problematic interactions also creates apprehension towards adopting these tools.

### 5.4 RQ4: What is the perceived benefit of using static analysis tools such as `mlint` to check SE guidelines in ML projects?

Participants note that they find tools such as `mlint` to be particularly useful for enforcing best practices, maintaining consistency in the project and discovering useful new tools in the Python ecosystem to improve their project quality. They also mention that static analysis tools such as `mlint` can help guide the direction that further development efforts should take in the road towards productionisation of the project. In doing so, these tools’ reports can be used as a project quality checklist.

*“It is a great way to enforce best practices, avoid common pitfalls and automate “common sense”. Without such tools it’s easy to be sloppy on project quality.”*



Participants find static analysis tools such as `mlLint` to also be useful for doing code review by helping to “*bring attention of a reviewer to potentially problematic pieces of code introduced, specially when integrated to automated CI pipelines.*”. Usage of these tools in CI is a popular suggestion, though varying suggestions are made as to the frequency of running them (e.g. on every pull request, at every release, or before finalising the project).

## 6 DISCUSSION

This section collects, combines and discusses our findings, sectioned along three themes: version controlling data, dependency management and static analysis tool adoption.

### 6.1 Data version control

While the results from RQ1 showed that none of the projects we analysed were using the tool DVC, this does not necessarily mean that industry ML practitioners are not version controlling their data. Validating with experienced ML engineers at ING, we find various ways in which data is managed: some projects only require data from the user at run-time, some have data small enough to fit in the code repository, but most prevalently, data is either pulled from a Hadoop filesystem or shared through other internal data sharing solutions, requiring the ML developer to download it manually. One ML engineer mentioned that they had experimented with DVC before, but found that it produced some overhead and preferred to stick with the semi-versioned workflow that they already had. Each of these methods has a varying degree of version control and a varying suitability towards certain types of data (consumption).

Overall, there seems to be a lack of standardised tooling for dealing with varying kinds of data dependencies, as practitioners generally stick with what is most practical or known to them. This presents a significant challenge to static analysis tools for detecting data version control techniques.

### 6.2 Dependency management

Overall, dependency management is perceived to be *very important* vs. *absolutely essential* and is done better in our industrial context than in the open-source context seen in [23], though practices still differ significantly between projects and developers. Some prefer to use external tooling such as Poetry, others prefer to create a manual workflow around Python’s standard `requirements.txt` and `setup.py` files. Such a workflow was argued by several ML engineers to be effectively usable with disciplined and sufficiently experienced users, though they do recognise that they are prone to misuse with less experienced users.

This is particularly troubling in ML project development, given the gap in SE experience in data scientists. Unlike other popular language ecosystems such as NodeJS (npm or yarn), Go (built-in) and Rust (cargo), which external tools like Poetry take heavy inspiration from, the Python language ecosystem still lacks a standardised, easy-to-use, consistently used, maintainable and reproducible method of managing code dependencies.

### 6.3 Static analysis tool adoption

Most notably, our results show a mixed sentiment towards static analysis tools. Combining our findings from RQ1 and RQ2 on code

quality project smells, we identify a tendency against using these linters during the development phase of the project and instead only adopting them during the productionisation phase, as an after-the-fact check on code quality. However, research has shown that linters are particularly useful during development for automatically fixing code styling (maintaining code consistency), avoiding complex code and finding potential bugs early [21]. Especially in ML applications, where one run of the program could take in the order of minutes or hours, linters can save the user from an unfortunate typo near the end of the program that would void all the time spent running it so far.

So why do practitioners refrain from adopting static analysis tools during their ML project development? Findings from RQ3, corroborated by existing research [21], show two primary obstructions towards their adoption: a high rate of false positives and cumbersome, time-consuming configuration.

False positives in static analysis tools—which may also include real but irrelevant warnings—are particularly common in dynamically typed, interpreted languages such as Python. Our previous research also found Pylint to produce a high rate of false positives on imports, both local imports as well as prominent ML libraries [23]. This problem is easier stated than solved, however, which calls for more development efforts and research into preventing false positives in static analysis on Python code, both from a tooling and linguistic perspective.

Surely, practitioners could also selectively disable the rules that produce false positives in the configuration of their linters. However, as our findings corroborate, creating and maintaining linter configurations is also a significant challenge in their adoption [21]. Practitioners, especially those in a team and / or inexperienced with the linter or their importance to the code quality of the project, find it difficult, cumbersome and time-consuming to define their standards and configure their linters to fit them.

Thus, for tool developers to have their tools become widely used, there seems to be an inherent trade-off between having a tool that is abstract or malleable enough to fit as many applications as possible (without producing false positives), while also requiring as little configuration effort from the user as possible. To achieve this, one set of recommendations is to simplify the configuration of the tool as much as possible: do not give the user unnecessary configuration options [25]. Similar to tools like `gofmt`, `govet` and `Black`: set standards and defaults that every user can agree on, then provide users with minimal knobs to adjust these, which barely leaves any room for bikeshedding.

However, many static analysis tools, including `mlLint`, are too complex or deal with too opinionated subjects to set *one* standard. Thus, there is also a need for *context-aware* static analysis tooling: by either automatically detecting or having the user configure in a quick and simple way what the context of the project / code under review is, the tool can automatically adjust its linting practices to conform to the user’s needs. In the case of `mlLint`, this context would include the project’s (desired) maturity (e.g. toy, proof-of-concept, production-ready), use-case or tool stack. As an example of the latter, a project that uses TensorFlow is better off using TFDV for data validation purposes than GreatExpectations. The same holds for other peripheral tooling. Such context-aware static analysis functionality could be realised by using presets or profiles, similar

to ESLint [21] and `isort`, though more research on this subject is recommended.

## 7 THREATS TO VALIDITY

### 7.1 Analysed projects & survey participants

In the qualitative analysis of `mlLint` reports of ING ML projects, we were only able to get access to or receive `mlLint` reports for eight projects. There may also be a selection bias in the projects that we got access to, as we were unable to get access to ML projects of more sensitive natures, due to internal regulations around either what data they process or what functionality they perform. We are therefore unsure of the generalisability of this dataset towards the state of ML projects at ING. We attempted to mitigate this problem by validating our generalised findings with experienced ML engineers at ING.

The limited number of survey participants (22) also presents a challenge to this research’s validity. It was difficult to gain survey responses, in part also since it required participants to have experience running `mlLint` on one of their own ML projects. However, the more qualitative nature of our survey questions and analysis of the answers, combined with validation with experienced ML engineers, may help to mitigate this challenge.

We also find a lack in diversity among our survey participants, given that most of the participants are ML engineers with extensive experience in SE. While it is certainly interesting to research how our tool relates to experienced engineers, it would be interesting to also include more data scientists with little experience in SE in our survey.

### 7.2 Construct validity

For this research, we primarily focused on the project smells that `mlLint` is able to detect reliably. However, there are many more project smells to be catalogued and to be implemented such that `mlLint` recognises them. Additionally, there are many more tools, techniques and SE practices that `mlLint` does not yet recognise or recommend, but are valid ways of mitigating project smells. Some of `mlLint`’s linting rules may currently also pertain more to the use of a tool that implements a certain ML project best practice, than to the practice itself, such as the rules about data version control with DVC. Therefore, until extended to support more tools, techniques and practices, `mlLint` can only reliably detect a limited set of project smells, consequently limiting the scope of this research.

Furthermore, for the RQ4, we focus on *perceived* benefits of static analysis tools, though it may be more interesting to investigate *observable* benefits. This could be done, for example, by asking ML practitioners to use `mlLint` in the development of their ML projects for a prolonged period of time and then seeing how the software quality of their ML projects increases over time. While this was considered, we acknowledge that `mlLint` is not yet mature enough to accurately measure the full software quality of an ML project. To the best of our knowledge, there is also no other tool that can accurately measure the full software quality of an ML project, without limiting its scope to only one or a few aspects of software quality. This poses a challenge to future research on this subject.

## 8 CONCLUSION & FUTURE WORK

In conclusion, this research introduced the novel concept of *project smells* as a more holistic view over code smells for assessing the software quality in ML projects and implemented a novel static analysis tool, `mlLint`, to help detect and mitigate these deficits in ML project management. We investigated how these project smells fit the industrial context of the large software- and data-intensive organisation ING, as well as the perceived importance of `mlLint`’s linting rules for ML practitioners on proof-of-concept versus production-ready projects. We also investigated the primary obstacles towards implementing solutions to these project smells and the perceived benefits of using static analysis tools such as `mlLint` to verify SE practices in ML projects.

### 8.1 Future Work

Future work should primarily focus on further development of `mlLint`, formally defining project smells and analysing their observable impact on the software quality of ML projects. Another interesting research direction is to investigate how ML development ecosystems can be redesigned or extended to inherently prevent such project smells from occurring.

Regarding the development of `mlLint`, there is a need to implement more linting rules (particularly ML-specific rules), such as on asserting data quality, having a reproducible, end-to-end ML pipeline and linting for ML-specific code smells. Additionally, existing linting rules should be extended with more in-depth, context-aware analysis of the project under review, with support for more toolsets and ML frameworks.

Future research could also investigate how *context-aware* static analysis can be applied in practice and how existing static analysis tools can adopt this. For `mlLint` it would entail implementing methods for detecting or in a simple way configuring the technology stack or ML framework used in the project and the project’s (desired) maturity. Based on this information, `mlLint` can then selectively disable non-applicable rules and adjusts the weights of other rules to fit with what is important for the project at that stage of development. However, the effort required to configure `mlLint` correctly, must also be minimal.

## ACKNOWLEDGMENTS

We would like to thank ING, their AI for FinTech Research Lab, GAP squad and ML Engineering chapters for providing their expertise throughout the development of `mlLint` and helping with its evaluation. Specifically, we would like to thank Elvan Kula and Hadi Hashemi for supervising this paper’s first author within ING. Additionally, we want to thank SE4ML, specifically Alex Serban and Joost Visser, for their insights, ideas, expertise and periodic feedback during the development of `mlLint`.

## REFERENCES

- [1] AFR. 2021. AI for Fintech Research. <https://se.ewi.tudelft.nl/ai4fintech/> (accessed: 06 Oct. 2021).
- [2] Md Abdullah Al Alamin and Gias Uddin. 2021. Quality Assurance Challenges for Machine Learning Software Applications During Software Development Life Cycle Phases. arXiv:2105.01195 [cs.SE]



- [3] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada) (ICSE-SEIP '19). IEEE Press, 291–300. <https://doi.org/10.1109/ICSE-SEIP.2019.00042>
- [4] Justus Bogner, Roberto Verdecchia, and Ilias Gerostathopoulos. 2021. Characterizing Technical Debt and Antipatterns in AI-Based Systems: A Systematic Mapping Study. In *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 64–73. <https://doi.org/10.1109/TechDebt52882.2021.00016>
- [5] Eric Breck, Shaoqing Cai, Eric Nielsen, M. Salib, and D. Sculley. 2017. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. *2017 IEEE International Conference on Big Data (Big Data)* (2017), 1123–1132.
- [6] Anita D. Carleton, Erin Harper, Tim Menzies, Tao Xie, Sigrid Eldh, and Michael R. Lyu. 2020. The AI Effect: Working at the Intersection of AI and SE. *IEEE Software* 37, 4 (2020), 26–35. <https://doi.org/10.1109/MS.2020.2987666>
- [7] M.P.A. Haakman. 2020. *Studying the Machine Learning Lifecycle and Improving Code Quality of Machine Learning Applications*. Master’s thesis. Delft University of Technology.
- [8] 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).
- [9] ING Bank N.V. 2021. ING at a glance | ING. <https://www.ing.com/About-us/Profile/ING-at-a-glance.htm> (accessed: 06 Oct. 2021).
- [10] Peter Kriens and Tim Verbelen. 2019. Software Engineering Practices for Machine Learning. arXiv:1906.10366 [cs.SE]
- [11] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610.
- [12] T. Menzies. 2020. The Five Laws of SE for AI. *IEEE Software* 37, 1 (2020), 81–85. <https://doi.org/10.1109/MS.2019.2954841>
- [13] 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. arXiv:2107.00079 [cs.LG]
- [14] Elizamary Nascimento, Anh Nguyen-Duc, Ingrid Sundbø, and Tayana Conte. 2020. Software engineering for artificial intelligence and machine learning software: A systematic literature review. *CoRR* abs/2011.03751 (2020). arXiv:2011.03751 <https://arxiv.org/abs/2011.03751>
- [15] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*. 2503–2511.
- [16] Alex Serban, Koen van der Blom, Holger Hoos, and Joost Visser. 2020. Adoption and Effects of Software Engineering Best Practices in Machine Learning. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (Bari, Italy) (ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. <https://doi.org/10.1145/3382494.3410681>
- [17] Alex Serban, Koen van der Blom, Holger Hoos, and Joost Visser. 2021. Practices for Engineering Trustworthy Machine Learning Applications. arXiv:2103.00964 [cs.SE]
- [18] Alex Serban, Koen van der Blom, Holger Hoos, and Joost Visser. 2021. SE-ML | Use Static Analysis to Check Code Quality. [https://se-ml.github.io/best\\_practices/03-use\\_static\\_analysis/](https://se-ml.github.io/best_practices/03-use_static_analysis/) (accessed: 06 Oct. 2021).
- [19] Alex Serban, Koen van der Blom, Holger Hoos, and Joost Visser. 2021. SE-ML Engineering best practices for Machine Learning. <https://se-ml.github.io/practices/> (accessed: 06 Oct. 2021).
- [20] Alex Serban and Joost Visser. 2021. An Empirical Study of Software Architecture for Machine Learning. arXiv:2105.12422 [cs.SE]
- [21] Kristín Tómasdóttir, Maurício 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 (2020), 863–891. <https://doi.org/10.1109/TSE.2018.2871058>
- [22] Koen van der Blom, Alex Serban, Holger Hoos, and Joost Visser. 2021. AutoML Adoption in ML Software. In *8th ICML Workshop on Automated Machine Learning (AutoML)*. <https://openreview.net/forum?id=D5H5LjwvIqt>
- [23] Bart van Oort, Luís Cruz, Maurício Aniche, and Arie van Deursen. 2021. The Prevalence of Code Smells in Machine Learning projects. In *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*. 1–8. <https://doi.org/10.1109/WAIN52551.2021.00011>
- [24] Hironori Washizaki, Hiromu Uchida, Foutse Khomh, and Yann-Gael Gueheneuc. 2019. Studying Software Engineering Patterns for Designing Machine Learning Systems. arXiv:1910.04736 [cs.SE]
- [25] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have given Me Too Many Knobs!: Understanding and Dealing with over-Designed Configuration in System Software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (Bergamo, Italy) (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 307–319. <https://doi.org/10.1145/2786805.2786852>
- [26] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering* (2020), 1–1. <https://doi.org/10.1109/TSE.2019.2962027>
- [27] Martin Zinkevich. 2021. Rules of Machine Learning: | ML Universal Guides | Google Developers. <https://developers.google.com/machine-learning/guides/rules-of-ml> (accessed: 06 Oct. 2021).

# 4

## CONCLUSION & FUTURE WORK

This MSc thesis has studied and shown the prevalence of code smells in Python ML applications, finding serious issues in the specification of code dependencies in these projects along the way. Inspired by a need for a more holistic viewpoint on ML project software quality than code smells, this research introduces the novel concept of *project smells*, which include code smells, but also encompasses deficits in (technological) project management, such as poor dependency management and lack of static analysis. I have also developed the open-source static analysis tool `mlLint` to help ML practitioners in detecting and mitigating these project smells. Though an ambitious project, this tool sparked the interest of ML practitioners both within ING as well as in the public and academic communities. Finally, in the case study with ING, it was investigated how the project smells fit the context of ING's FinTech industry, how ML practitioners perceive the importance of `mlLint`'s linting rules and the considerations in adopting static analysis tools such as `mlLint`.

### 4.1 FUTURE WORK

On the topic of code smells, this research was primarily concerned with non-ML-specific code smells in open-source Python ML projects. It would be interesting to replicate this study on closed-source ML projects from the industry, as well as on code extracted from Jupyter Notebooks. Alternatively, it could be interesting to use code smell detection tools in other languages used to build ML applications, such as Julia, R or Java.

Additionally, it may be interesting to further investigate ML-specific code smells, directly continuing the work of [Haakman \[2020\]](#) and his `dslinter` tool. This Pylint plugin still contains a few false positives that could be remedied and could also be extended with novel ML-specific code smells, possibly pertaining to the use of other popular ML frameworks such as PyTorch or TensorFlow. Potentially, the authors of these frameworks could also consider developing their own static analysis tools / plugins to help their users in detecting common misuses, simultaneously hinting towards correct usage.

Regarding `mlLint`, there is still a plethora of possible tools, techniques and project configurations to support and provide useful recommendations on. There are also many ML best practices yet to be enforced by `mlLint` that ML practitioners could benefit from. Most strongly, I recommend asserting *Data Quality* practices to ensure consistent quality of input data for the ML algorithm. Other areas of project management where project smells could likely be defined and detected are in deployment practices, documentation, monitoring, folder structure, ML-specific testing techniques, explainability of used models and checks on automation practices. Additionally, `mlLint` could be extended with linters for ML-specific code smells and existing `mlLint` rules can be improved to support more in-depth analyses of the project. All in all, there is a broad horizon of new possibilities for asserting ML project software quality.

It may also be interesting to investigate new features for `mlLint`. For example, it could be interesting to implement a feature to allow `mlLint` to fix detected problems for the user. This could help

ML practitioners in fixing their project smells, without requiring extensive knowledge of SE tools and techniques. Alternatively, new research could consider how `mlLint` can educate data scientists on state-of-the-art, technological SE and ML engineering practices. One possible approach to such education includes extending `mlLint`'s rule documentation with extra information or educational resources that teach about the values behind each of the linting rules, making it a self-help tool that also assesses the user's skill in implementing these practices on their own ML project. Teams, squads, chapters or other organisations can also teach and verify internal practices on their projects through the implementation and use of custom `mlLint` rules. Educating data scientists as they progress along their usual development routines, armed with `mlLint` to help them practically implement the learned practices, would allow them to learn in a productive manner on *actual* ML projects, rather than through contrived examples.

Most notably though, it should be investigated how linting profiles or presets can be applied to allow `mlLint` to detect and conform its recommendations to a specific project architecture and maturity. After all, a project that uses PyTorch requires different tooling than TensorFlow projects. Similarly, a proof-of-concept project does not need as strict software quality requirements as a production-ready project, as this research has corroborated. However, too much configuration options hurts the adoption of static analysis tools [Xu et al., 2015]. Thus, `mlLint` should be made *context-aware*: able to conform to the user's expectations for the project with minimal need for configuration, while still enforcing quality standards.

## BIBLIOGRAPHY

- Haakman, M. (2020). Studying the Machine Learning Lifecycle and Improving Code Quality of Machine Learning Applications. Master's thesis, Delft University of Technology.
- Haakman, M., Cruz, L., Huijgens, H., and van Deursen, A. (2021). AI lifecycle models need to be revised. An exploratory study in FinTech. *Empirical Software Engineering*.
- ING Bank N.V. (2021). ING at a glance | ING. <https://www.ing.com/About-us/Profile/ING-at-a-glance.htm>. (accessed: 29 Sept. 2021).
- Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-E., and Dennison, D. (2015). Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*, pages 2503–2511.
- Serban, A., van der Blom, K., Hoos, H., and Visser, J. (2021). SE-ML Engineering best practices for Machine Learning. <https://se-ml.github.io/practices/>. (accessed: 06 Oct. 2021).
- van Oort, B., Cruz, L., Aniche, M., and van Deursen, A. (2021). The Prevalence of Code Smells in Machine Learning projects. In *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, pages 1–8.
- van Oort, B., Cruz, L., Loni, B., and van Deursen, A. (2022). “Project Smells” — Experiences in Analysing the Software Quality of ML Projects with mllint. *Submitted to ICSE-SEIP*.
- Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., and Talwadker, R. (2015). Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, page 307–319, New York, NY, USA. Association for Computing Machinery.



## TABLE OF LINTING RULE PRIORITISATION SURVEY RESULTS

To answer RQ2 in the paper about project smells and `mlLint` as shown in [Chapter 3](#), we ran a survey in which we asked participants to rate the importance of each of `mlLint`'s linting rules. This appendix contains the full table of results from the analysis of those survey responses.

The five possible Likert-scale answers were encoded to integers between -2 and 2. These numeric values have been included in this table to show the spread of each result.

Note that the values for *Mean +/- StdDev* were clamped to be real values between -2 and 2.

### A.1 PROOF-OF-CONCEPT PROJECT

Name	Min	Mean - StdDev	Mean	Mean + StdDev	Max	Unknown / unsure
0 Project uses Git (version-control/code/git)	<b>Not important</b> (-2.0)	<b>Moderately important</b> (0.241)	<b>Very important</b> (1.273)	<b>Absolutely Essential</b> (2)	<b>Absolutely Essential</b> (2.0)	0
1 Project should not have any large files in its Git history (version-control/code/git-no-big-files)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.804)	<b>Very important</b> (0.524)	<b>Absolutely Essential</b> (1.851)	<b>Absolutely Essential</b> (2.0)	1
2 DVC: Project uses Data Version Control (version-control/data/dvc)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.699)	<b>Slightly important</b> (-0.526)	<b>Very important</b> (0.646)	<b>Absolutely Essential</b> (2.0)	3
3 DVC: Is installed (version-control/data/dvc-is-installed)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.788)	<b>Slightly important</b> (-0.6)	<b>Very important</b> (0.588)	<b>Absolutely Essential</b> (2.0)	2
4 DVC: Folder '.dvc' should be committed to Git (version-control/data/commit-dvc-folder)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.806)	<b>Moderately important</b> (-0.45)	<b>Very important</b> (0.906)	<b>Absolutely Essential</b> (2.0)	2
5 DVC: Should have at least one remote data storage configured (version-control/data/dvc-has-remote)	<b>Not important</b> (-2.0)	<b>Not important</b> (-2)	<b>Slightly important</b> (-1.105)	<b>Moderately important</b> (-0.169)	<b>Moderately important</b> (0.0)	3
6 DVC: Should be tracking at least one data file (version-control/data/dvc-has-files)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.974)	<b>Slightly important</b> (-0.632)	<b>Very important</b> (0.71)	<b>Absolutely Essential</b> (2.0)	3
7 DVC: File 'dvc.lock' should be committed to Git (version-control/data/commit-dvc-lock)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.925)	<b>Slightly important</b> (-0.579)	<b>Very important</b> (0.767)	<b>Absolutely Essential</b> (2.0)	3

	Name	Min	Mean - StdDev	Mean	Mean + StdDev	Max	Unknown / unsure
8	Project properly keeps track of its dependencies (dependency-management/use)	<b>Slightly important</b> (-1.0)	<b>Slightly important</b> (-0.591)	<b>Very important</b> (0.591)	<b>Absolutely Essential</b> (1.772)	<b>Absolutely Essential</b> (2.0)	0
9	Project should only use one dependency manager (dependency-management/single)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.96)	<b>Moderately important</b> (0.476)	<b>Absolutely Essential</b> (1.912)	<b>Absolutely Essential</b> (2.0)	1
10	Project places its development dependencies in dev-dependencies (dependency-management/use-dev)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.633)	<b>Moderately important</b> (-0.286)	<b>Very important</b> (1.061)	<b>Absolutely Essential</b> (2.0)	1
11	Project uses Continuous Integration (CI) (ci/use)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-1.44)	<b>Moderately important</b> (-0.19)	<b>Very important</b> (1.059)	<b>Absolutely Essential</b> (2.0)	1
12	Project should use code quality linters (code-quality/use-linters)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.605)	<b>Moderately important</b> (-0.227)	<b>Very important</b> (1.151)	<b>Absolutely Essential</b> (2.0)	0
13	All code quality linters should be installed in the current environment (code-quality/linters-installed)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.978)	<b>Slightly important</b> (-0.619)	<b>Very important</b> (0.74)	<b>Absolutely Essential</b> (2.0)	1
14	Pylint reports no issues with this project (code-quality/pylint/no-issues)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.952)	<b>Slightly important</b> (-0.8)	<b>Moderately important</b> (0.352)	<b>Absolutely Essential</b> (2.0)	2
15	Pylint is configured for this project (code-quality/pylint/is-configured)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.779)	<b>Slightly important</b> (-0.571)	<b>Very important</b> (0.636)	<b>Absolutely Essential</b> (2.0)	1
16	Mypy reports no issues with this project (code-quality/mypy/no-issues)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.934)	<b>Slightly important</b> (-0.684)	<b>Very important</b> (0.565)	<b>Absolutely Essential</b> (2.0)	3
17	Black reports no issues with this project (code-quality/black/no-issues)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.867)	<b>Slightly important</b> (-0.55)	<b>Very important</b> (0.767)	<b>Absolutely Essential</b> (2.0)	2
18	isort reports no issues with this project (code-quality/isort/no-issues)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.901)	<b>Slightly important</b> (-0.722)	<b>Moderately important</b> (0.456)	<b>Absolutely Essential</b> (2.0)	4
19	isort is properly configured (code-quality/isort/is-configured)	<b>Not important</b> (-2.0)	<b>Not important</b> (-2)	<b>Slightly important</b> (-0.889)	<b>Moderately important</b> (0.294)	<b>Absolutely Essential</b> (2.0)	4
20	Bandit reports no issues with this project (code-quality/bandit/no-issues)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.766)	<b>Slightly important</b> (-0.688)	<b>Moderately important</b> (0.391)	<b>Very important</b> (1.0)	6
21	Project has automated tests (testing/has-tests)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-1.433)	<b>Moderately important</b> (-0.095)	<b>Very important</b> (1.243)	<b>Absolutely Essential</b> (2.0)	1

Name	Min	Mean - StdDev	Mean	Mean + StdDev	Max	Unknown / unsure
22 Project passes all of its automated tests (testing/pass)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-1.242)	<b>Moderately important</b> (0.238)	<b>Absolutely Essential</b> (1.718)	<b>Absolutely Essential</b> (2.0)	1
23 Project provides a test coverage report (testing/coverage)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.611)	<b>Moderately important</b> (-0.333)	<b>Very important</b> (0.945)	<b>Absolutely Essential</b> (2.0)	1

## A.2 PRODUCTION-READY PROJECT

Name	Min	Mean - StdDev	Mean	Mean + StdDev	Max	Unknown / unsure
0 Project uses Git (version-control/code/git)	<b>Very important</b> (1.0)	<b>Absolutely Essential</b> (1.615)	<b>Absolutely Essential</b> (1.909)	<b>Absolutely Essential</b> (2)	<b>Absolutely Essential</b> (2.0)	0
1 Project should not have any large files in its Git history (version-control/code/git-no-big-files)	<b>Not important</b> (-2.0)	<b>Moderately important</b> (0.034)	<b>Very important</b> (1.143)	<b>Absolutely Essential</b> (2)	<b>Absolutely Essential</b> (2.0)	1
2 DVC: Project uses Data Version Control (version-control/data/dvc)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.632)	<b>Very important</b> (0.611)	<b>Absolutely Essential</b> (1.854)	<b>Absolutely Essential</b> (2.0)	4
3 DVC: Is installed (version-control/data/dvc-is-installed)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.803)	<b>Moderately important</b> (0.444)	<b>Absolutely Essential</b> (1.692)	<b>Absolutely Essential</b> (2.0)	4
4 DVC: Folder 'dvc' should be committed to Git (version-control/data/commit-dvc-folder)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.795)	<b>Moderately important</b> (0.5)	<b>Absolutely Essential</b> (1.795)	<b>Absolutely Essential</b> (2.0)	4
5 DVC: Should have at least one remote data storage configured (version-control/data/dvc-has-remote)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.81)	<b>Moderately important</b> (0.471)	<b>Absolutely Essential</b> (1.751)	<b>Absolutely Essential</b> (2.0)	5
6 DVC: Should be tracking at least one data file (version-control/data/dvc-has-files)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.76)	<b>Moderately important</b> (0.471)	<b>Absolutely Essential</b> (1.701)	<b>Absolutely Essential</b> (2.0)	5
7 DVC: File 'dvc.lock' should be committed to Git (version-control/data/commit-dvc-lock)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.869)	<b>Moderately important</b> (0.353)	<b>Absolutely Essential</b> (1.575)	<b>Absolutely Essential</b> (2.0)	5
8 Project properly keeps track of its dependencies (dependency-management/use)	<b>Moderately important</b> (0.0)	<b>Very important</b> (1.055)	<b>Absolutely Essential</b> (1.636)	<b>Absolutely Essential</b> (2)	<b>Absolutely Essential</b> (2.0)	0
9 Project should only use one dependency manager (dependency-management/single)	<b>Not important</b> (-2.0)	<b>Moderately important</b> (-0.37)	<b>Very important</b> (0.952)	<b>Absolutely Essential</b> (2)	<b>Absolutely Essential</b> (2.0)	1

	Name	Min	Mean - StdDev	Mean	Mean + StdDev	Max	Unknown / unsure
10	Project places its development dependencies in dev-dependencies (dependency-management/use-dev)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.813)	<b>Moderately important</b> (0.476)	<b>Absolutely Essential</b> (1.765)	<b>Absolutely Essential</b> (2.0)	1
11	Project uses Continuous Integration (CI) (ci/use)	<b>Not important</b> (-2.0)	<b>Moderately important</b> (0.41)	<b>Very important</b> (1.364)	<b>Absolutely Essential</b> (2)	<b>Absolutely Essential</b> (2.0)	0
12	Project should use code quality linters (code-quality/use-linters)	<b>Not important</b> (-2.0)	<b>Moderately important</b> (-0.039)	<b>Very important</b> (0.905)	<b>Absolutely Essential</b> (1.848)	<b>Absolutely Essential</b> (2.0)	1
13	All code quality linters should be installed in the current environment (code-quality/linters-installed)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-1.137)	<b>Moderately important</b> (0.238)	<b>Absolutely Essential</b> (1.613)	<b>Absolutely Essential</b> (2.0)	1
14	Pylint reports no issues with this project (code-quality/pylint/no-issues)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.832)	<b>Moderately important</b> (0.35)	<b>Absolutely Essential</b> (1.532)	<b>Absolutely Essential</b> (2.0)	2
15	Pylint is configured for this project (code-quality/pylint/is-configured)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.831)	<b>Moderately important</b> (0.4)	<b>Absolutely Essential</b> (1.631)	<b>Absolutely Essential</b> (2.0)	2
16	Mypy reports no issues with this project (code-quality/mypy/no-issues)	<b>Slightly important</b> (-1.0)	<b>Slightly important</b> (-0.944)	<b>Moderately important</b> (0.222)	<b>Very important</b> (1.388)	<b>Absolutely Essential</b> (2.0)	4
17	Black reports no issues with this project (code-quality/black/no-issues)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-1.043)	<b>Moderately important</b> (0.25)	<b>Absolutely Essential</b> (1.543)	<b>Absolutely Essential</b> (2.0)	2
18	isort reports no issues with this project (code-quality/isort/no-issues)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.602)	<b>Moderately important</b> (-0.111)	<b>Very important</b> (1.38)	<b>Absolutely Essential</b> (2.0)	4
19	isort is properly configured (code-quality/isort/is-configured)	<b>Not important</b> (-2.0)	<b>Not important</b> (-1.804)	<b>Moderately important</b> (-0.278)	<b>Very important</b> (1.249)	<b>Absolutely Essential</b> (2.0)	4
20	Bandit reports no issues with this project (code-quality/bandit/no-issues)	<b>Not important</b> (-2.0)	<b>Slightly important</b> (-0.738)	<b>Very important</b> (0.556)	<b>Absolutely Essential</b> (1.849)	<b>Absolutely Essential</b> (2.0)	4
21	Project has automated tests (testing/has-tests)	<b>Slightly important</b> (-1.0)	<b>Moderately important</b> (0.461)	<b>Very important</b> (1.364)	<b>Absolutely Essential</b> (2)	<b>Absolutely Essential</b> (2.0)	0
22	Project passes all of its automated tests (testing/pass)	<b>Slightly important</b> (-1.0)	<b>Very important</b> (0.654)	<b>Very important</b> (1.455)	<b>Absolutely Essential</b> (2)	<b>Absolutely Essential</b> (2.0)	0
23	Project provides a test coverage report (testing/coverage)	<b>Not important</b> (-2.0)	<b>Moderately important</b> (-0.283)	<b>Very important</b> (0.909)	<b>Absolutely Essential</b> (2)	<b>Absolutely Essential</b> (2.0)	0



## COLOPHON

This document was typeset using  $\LaTeX$ . The document layout was generated using a manually edited version of the MSc Geomatics Thesis Template on Overleaf by Hugo Ledoux<sup>1</sup>, which is based on the `arclassica` package by Lorenzo Pantieri, which is an adaption of the original `classicthesis` package from André Miede.

---

<sup>1</sup> <https://www.overleaf.com/latex/templates/msc-geomatics-thesis-template-tu-delft/yvjpkwvtkrwz>

