Automated Detection of Code Smells for Machine Learning Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Haiyin Zhang
born in Guangdong, China

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

AI for Fintech Research
ING Bank N.V.
Frankemaheerd 1
Amsterdam, the Netherlands
www.ing.nl

Automated Detection of Code Smells for Machine Learning Applications

Author: Haiyin Zhang
Student id: 5221749

Abstract

The popularity of machine learning has wildly expanded in recent years. Machine learning techniques have been heatedly studied in academia and applied in the industry to create business value. However, there is a lack of guidelines for code quality in machine learning applications. Although machine learning code is usually integrated as a small part of an overarching system, it usually plays an important role in its core functionality. Hence ensuring code quality is quintessential to avoiding issues in the long run. To help improve the machine learning code quality, we conducted two studies in this thesis. The first study proposes and identifies a list of 22 machine learning-specific code smells collected from various sources, including papers, grey literature, GitHub commits, and Stack Overflow posts. We pinpoint each smell with a description of its context, potential issues in the long run, and proposed solutions. In addition, we link them to their respective pipeline stage and the evidence from both academic and grey literature. The second study aims to develop a tool to improve code quality and study the prevalence of machine learning-specific code smells. We extend a static analysis tool *dslinter* and run it on both Python notebook datasets and regular Python project datasets. Moreover, we analyse the result to check the tool's validity and investigate the code smell prevalence in machine learning applications. The code smell catalog and *dslinter* together help data scientists and developers produce and maintain high-quality machine learning application code.

Thesis Committee:

Chair and University Supervisor:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University Supervisor:	Dr. L. Cruz, Faculty EEMCS, TU Delft
Committee Member:	Dr. J. Yang, Faculty EEMCS, TU Delft

Preface

It has been a wonderful thesis journey to spend nine months with a lot of great people :) Here I'd like to express my gratitude to all the people who helped, supported, and accompanied me through this journey.

First and foremost, I would like to thank my daily supervisor Dr. Luís Cruz, for his excellent academic guidance and constant encouragement throughout my entire thesis process. Thank you Luís, for introducing me to this interesting research topic, guiding me through the enlightening weekly meetings, providing me with detailed feedback and offering me a lot of chances to explore things. I'm fortunate enough to be one of your students. Thank you Prof. Arie van Deursen, for providing valuable feedback at the crucial points and keeping the project on the right track. In addition, my gratitude goes to ING AI For Fintech Lab (AFR) for the interesting weekly meet-ups and pleasant gatherings during my thesis journey. I also thank Dr. Jie Yang for co-reading my thesis and taking part in my thesis committee.

Next, I want to thank my family and friends, who always accompany me and help me. Thank you to all my friends, Danyao, Jie, Mingyu, Chadha, my roommates, teammates, study buddies at Building 28, and whoever we have a good time with. Thank you for all the lunches, dinners, happy weekends and laughing moments. Master thesis life would have been less fun without you. Thank you to my boyfriend Qingyuan, who has been supportive all the time and shared my happiness and sadness. Finally, I want to thank my parents for supporting me and always caring for me. Without all the help, I couldn't finish this journey.

I'm so grateful to have had the chance to study in the beautiful Delft for two years, which will definitely be an unforgettable memory in my life.

Haiyin Zhang
Delft, the Netherlands
June 30, 2022

Contents

List of Figures

Chapter 1

Introduction

Machine learning is gaining increasing interest in both academia and industry nowadays. The next generation of software systems will likely incorporate machine learning. However, it is challenging to construct production systems with machine learning components. 87% of data science projects never reach production, despite initial success with machine learning model training [1]. Efforts still need to be made to facilitate the production process of machine learning systems.

In this thesis study, we attempt to improve the reliability of the machine learning applications and thereby assist their production. We do this by identifying some coding issues in machine learning applications, developing a static analysis tool to help developers avoid these coding issues, and studying the prevalence of machine learning-specific code smells.

1.1 Objectives and Research Questions

This section briefly states the objectives and research questions of both studies performed in this thesis project.

Code Smells for Machine Learning Applications: This study aims to learn more about the code issues in machine learning applications. We defined the following research question for this study:

1. What are the recurrent code issues that may arise from the peculiarities of machine learning applications?

Automated Detection of ML-Specific Code Smells: The objective of this study is to learn how widespread these coding pitfalls are in the public repositories, and to what extent we can solve these problems by developing a tool specific for machine learning code smells. We defined the following research questions for this study:

1. How prevalent are the machine learning-specific code smells?

2. How accurate is the machine learning-specific code smell detection tool dslinter?

3. Is the prevalence of code smells different in Python notebooks and regular Python projects?

[1] Why do 87% of data science projects never make it into production?: `https://venturebeat.com/2019/07/19/why-do-87-of-data-science-projects-never-make-it-into-production/`

1.2 Approach

This section briefly introduces the study approaches of both studies.

Code Smells for Machine Learning Applications:

With the goal of identifying the coding issues in machine learning applications, we propose and identify a list of 22 machine learning-specific code smells collected from various sources, including papers, grey literature, GitHub commits, and Stack Overflow posts. We pinpoint each smell with a description of its context, potential issues in the long run, and proposed solutions. In addition, we link them to their respective pipeline stage and the evidence from both academic and grey literature.

Automated Detection of ML-Specific Code Smells:

We develop a static analysis tool `dslinter` with the purpose of automating the detection of ml-specific code smells. Then, we run `dslinter` on notebook and project datasets to research the prevalence of ml-specific code smells. Further analyses are conducted to understand the validity of the tool and the code quality difference between Python notebooks and regular Python projects.

1.3 Contributions

The main contributions of this thesis are as follows:

1. A catalog of machine learning-specific code smells, and a website to present all the smells [2].

2. A replication package [3] for identifying the code smells, which contains 1750 papers, 2170 grey literature entries, 87 GitHub commits and 491 Stack Overflow posts for empirical studies.

3. A research paper [4] "Code Smells for Machine Learning Applications", which has been accepted by the International Conference on AI Engineering - Software Engineering for AI (CAIN) 2022, to share the machine learning-specific code smells we identified with the research community.

4. An open-source static analysis tool `dslinter` to detect the machine learning-specific code smells, which is available at GitHub [5] and PyPI [6].

5. An empirical study on the prevalence of code smells in both public Python notebook datasets and regular Python project datasets.

6. A replication package [7] for the empirical study.

[2]Code Smell Catalog Website: `https://hynn01.github.io/ml-smells/`

[3]Replication Package 1: `https://github.com/Hynn01/ml-smells`

[4]Paper: `https://arxiv.org/pdf/2203.13746.pdf`

[5]dslinter at GitHub: `https://github.com/SERG-Delft/dslinter`

[6]dslinter at PyPI: `https://pypi.org/project/dslinter/`

[7]Replication Package 2: `https://github.com/Hynn01/dslinter-experiments`

1.4 Report Organization

The remaining of the thesis contains three chapters, one of each study and a conclusion chapter. Chapter 2 includes the study of identifying code smells for machine learning applications. Chapter 3 contains the study for building an automated code smell detection tool `dslinter` and the research on the prevalence of machine learning-specific smells in the public datasets. Each chapter stands on its own, containing an introduction, related work, methodology, results, discussions and implications, thread to validity, conclusions and future work. Chapter 4 describes the conclusions for the thesis and the future work. In the Appendix, the grey literature references for identifying the code smells can be found.

Chapter 2

Code Smells for Machine Learning Applications

2.1 Introduction

Despite the large increase in the popularity of machine learning applications [6], there are several concerns regarding the quality control and the inevitable technical debt growing in these systems [22]. Moreover, machine learning teams tend to be very heterogeneous, having experts from different disciplines that are not necessarily aware of Software Engineering (SE) practices backgrounds and there is a limited number of training and guidelines on machine learning-related software development issues. Hence, software engineering best practices are often overlooked when developing machine learning applications [24, 16]. Yet, previous research shows that practitioners are eager to learn more about engineering best practices for their machine learning applications [8].

There has been a lot of interest in various machine learning system artifacts, including models and data. Researchers make efforts to improve machine learning model quality [14] and data quality [11]. However, the quality assurance of machine learning code has not been highlighted [16]. Recent work studied the code quality for machine learning applications in a general way, finding some code quality issues such as duplicated code [29] and violations of traditional naming convention [24]. These works highlighted the fact that the existing code conventions do not necessarily fit the context of machine learning applications. For example, the typical math notation in data science tasks clashes with the naming conventions of Python [29]. Thus, we argue that more research is needed to accommodate the particularities of data-oriented codebases.

As an important artifact in the machine learning application, the quality of the code is essential. Low-quality code can lead to catastrophic consequences. In the meantime, different from traditional software, machine learning code quality is more challenging to evaluate and control. Low-quality code can lead to silent pitfalls that exist somewhere that affect the software quality, which takes a lot of time and effort to discover [33]. Therefore, it is non-trivial to improve the code quality during the development process and consider code quality assurance in the deployment process.

A common strategy to improve code quality is eliminating code smells and anti-patterns.

When we talk about code smells in this paper, we refer them to the pitfalls that we can inspect at the code level but not at the data or model level. We use the term “pitfall” to represent issues that degrade the software quality. Listing 2.1 shows an example of such pitfalls using Python and the Pandas library. In the red (-) part of the listing, an inefficient loop is created. A better alternative is highlighted in green (+), using Pandas built-in API to replace the loop, which operates faster. While some alternative solutions might lead to improvements in runtime efficiency, other solutions might be essential to prevent problems in the long run. For example, previous work shows that code smells affect the maintainability, understandability, and complexity of software [15].

Listing 2.1: Coding Pitfall Example from [9]

```
import pandas as pd
df = pd.DataFrame([1, 2, 3])

- result = []
- for index, row in df.iterrows();
-   result.append(row[0] + 1)
- result = pd.DataFrame(result)
+ result = df.add(1)
```

With the concern of improving machine learning application code quality and easing the machine learning development process, we conduct an empirical study to collect machine learning-specific code smells and provide practical recommendations about the quality in machine learning applications. Thus, we formulate the following research question: *What are the recurrent code issues that may arise from the peculiarities of machine learning applications?*

The main contributions of this chapter are:

1) A catalog of machine learning-specific code smells.

2) A dataset of 1750 papers, 2170 grey literature entries, 87 GitHub commits and 491 Stack Overflow posts for empirical studies.

The replication package for this study is available at `https://github.com/Hynn01/ml-smells`. The website with all the smells is published at `https://hynn01.github.io/ml-smells/`.

2.2 Related Work

Code smells are common poor code design choices that negatively affect the systems and violate the best practice or original design vision [15]. Martin Fowler introduced a general code smell list in his book [17]. Ever since then, code smells have been widely discussed in studies. Many empirical studies have linked code smell proliferation with decreased code quality, increased error proneness, and increased maintainability issues in the long term [25, 19, 31].

The prevalence of traditional code smells in machine learning projects was studied in van Oort et al.’s paper [29]. They ran Pylint on 74 machine learning projects and concluded the most frequent traditioal code smells. Yet, they noted that “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." This implies that the context of machine learning applications brings new challenges to the code quality. Therefore, our work addresses machine learning-specific code smells.

Even though there are few code smell studies specific to machine learning application coding, some researchers are studying refactoring and bugs associated with machine learning, which are related to machine learning coding patterns. Most relatedly, Tang et al. studied refactoring in machine learning programs by analyzing 26 projects [26]. They introduced 14 new machine learning-specific refactorings and seven new machine learning-specific technical debt. However, some of the machine learning programs they analyzed are machine learning tools, while we focus on machine learning applications. We argue that the underlying nature of machine learning libraries and tools is very different from applications. In addition, they focused on classifying different types of refactoring (e.g., "make algorithms more visible"), but they did not extract the code patterns that should trigger such a refactoring. We take a step further by addressing this question. Furthermore, we focus on code smells that cannot be identified by looking at general-purpose smells. For example, while it makes sense to have a type of refactoring for "duplicate model code", its code pattern is no different from the traditional smell "duplicated code". Our paper dives deep into code patterns and examples that are at the machine-learning library API usage level, which is different from their work.

Zhang et al. conducted an empirical study, mining Stack Overflow and GitHub commits to studying the TensorFlow bugs [33]. They proposed several bug patterns, which are helpful when debugging deep learning applications. Islam et al. followed up by inspecting Stack Overflow blogs and GitHub commits of five deep learning libraries, including Caffe, Keras, Tensorflow, Theano, and Torch [12]. They adopted some of the root causes of deep learning bugs from [33] and added more root causes. Also, they studied the impacts of bugs, the common patterns of the bugs, and the evolution of the bugs. Humbatova et al. created a comprehensive taxonomy of deep learning bugs by mining GitHub, mining Stack Overflow and interviewing developers [10]. The final taxonomy is quite thorough and detailed.

Our work differs from these two studies in four main reasons: 1) we formulate practical coding advice in the form of code smells, to improve the code and avoid potential issues in the long run, 2) we only focus on issues that can be inspected at the code level, 3) we not only focus on pitfalls that lead to potential bugs but also on performance, reproducibility, and maintainability issues, 4) we expand the scope of these smells beyond the deep learning discipline, focusing on other machine learning tasks provided in the libraries Scikit-Learn, Pandas, NumPy and SciPy.

Rajbahadur et al. collected eight data science project pitfalls from a paper and used a model-driven method to detect the pitfalls in the pipeline. Our study differs from theirs by inspecting the faults in the code level to assure the software quality [21]. Breck et al. learned from the experience with a wide range of production machine learning systems at Google and presented 27 machine learning-specific tests and monitoring needs [4]. However, it does not provide a concrete coding guideline. We go further by building a machine learning-specific code smell catalog and guide machine learning developers towards better coding practices by eliminating code smells.

2.3 Methodology

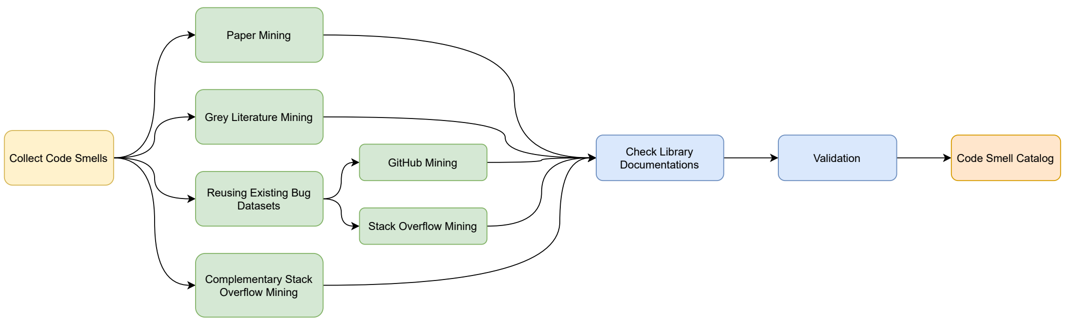


Figure 2.1: Methodology

To collect machine learning-specific code smells, we resort to academic literature, grey literature, community-based coding Q&A platforms (with Stack Overflow), and public software repositories (with GitHub). The general process is depicted in Figure 2.1. We mine papers, grey literature, reuse existing bug datasets, and conduct a complementary Stack Overflow mining. Then we triangulate our collected smells with the recommendations provided in the official documentation of machine learning libraries. In the end, two authors validate the code smell catalog.

2.3.1 Paper Mining

Our methodology for paper mining is described as follows, and shown as Figure 2.2:

1) Search on Google Scholar search engine: To collect papers that potentially contain code smells for machine learning projects, we use terms combining machine learning-related keywords and code quality-related keywords to search. Machine learning-related keywords include "Artificial Intelligence", "Machine Learning", "Deep Learning", "Neural Network" and "Data Science". Code quality-related keywords include "Technical Debt", "Refactoring", "Code Smell", "Code Quality", "Coding Best Practice", "Coding Anti-pattern" and "Common Coding Mistakes". We apply these queries (e.g., "Machine Learning Technical Debt") in the Google Scholar search engine, as presented in Figure 2.3. After analyzing papers from the initial result set, we reach a level of saturation for each query after consulting the first five pages of the result. Therefore, we consult the first five pages of the results for each query, i.e., the first 50 results, sorted by relevance at any time by any type. In total, there are 1750 papers ($5 \times 7 \times 50$).

2) Selection based on title and abstract: We observe that there are many papers studying machine learning for software engineering (ML4SE) and a few are about software engineering for machine learning (SE4ML). For example, for a paper titled "Comparing and experimenting machine learning techniques for code smell detection", we identify it as an

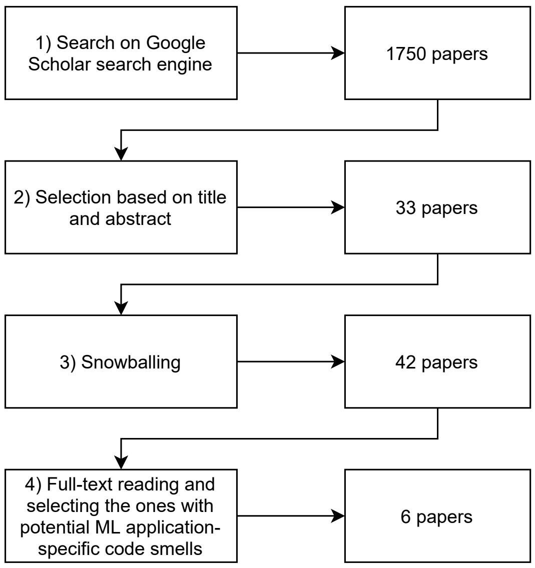

1) Search on Google Scholar search engine
1750 papers
2) Selection based on title and abstract
33 papers
3) Snowballing
42 papers
4) Full-text reading and selecting the ones with potential ML application-specific code smells
6 papers

Figure 2.2: Paper Selection Process

Table 2.1: Number of Selected Papers under Each Query (Duplicates Included)

	Technical Debt	Refactoring	Code Smell	Code Quality	Coding Best Practice	Coding Anti-pattern	Common Coding Mistakes
Artificial Intelligence	7	1	0	2	0	2	0
Machine Learning	6	1	0	2	7	3	3
Deep Learning	8	3	0	4	4	1	5
Neural Network	3	0	0	0	0	0	2
Data Science	2	0	0	0	2	1	0

ML-for-SE paper and exclude it from our study. When we cannot classify the paper from the title (e.g., “Toward deep learning software repositories”), we look into the abstract and decide whether to include it in our study. The numbers of selected papers under each query are listed in Table 2.1. After excluding the duplicated ones, our methodology yields 33 papers.

3) Snowballing: We apply the forward and backward snowballing method, i.e., browse the reference list of the 33 papers and the list where the paper is cited, select the paper based on the title and abstract as step 2) described, and delete the duplicated papers. We add nine papers after this step.

4) Full-text reading and selecting the ones with potential machine learning-specific code smells: We read the full text of the 42 papers and select the ones with potential machine learning-specific code smells. After this step, we get six final papers. The papers that contribute to the code smell catalog are listed as follows: [4, 9, 33, 12, 21, 10].

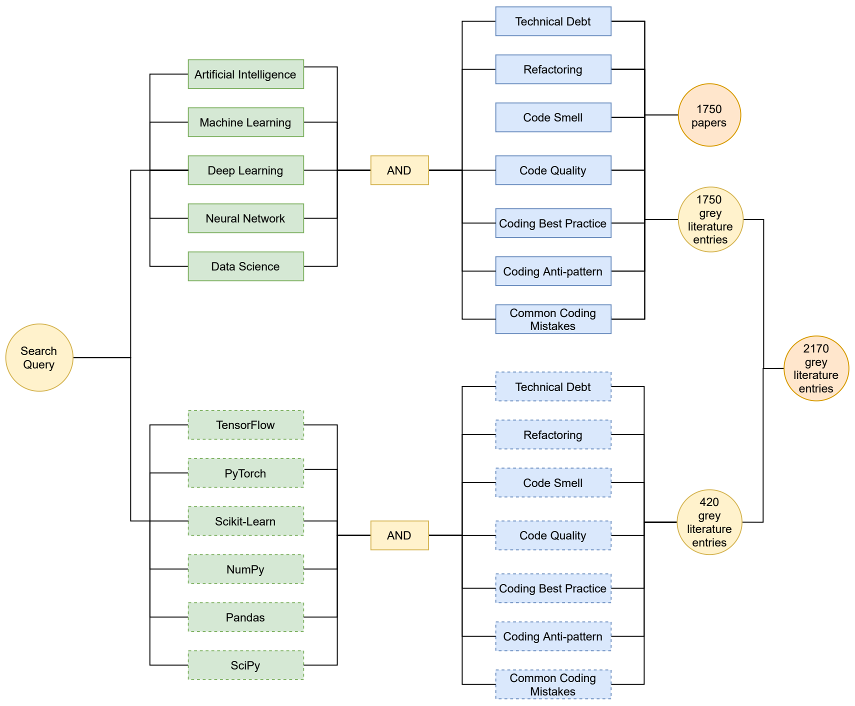


Figure 2.3: Search Query for Literature and Grey Literature

2.3.2 Grey Literature Mining

Many relevant pieces of knowledge about machine learning engineering are being published on the Web by experienced practitioners – for example, in the format of blog posts. Hence, we use grey literature as a relevant source for machine learning-specific coding advice in this study.

To collect online entries of grey literature, we first resort to the Google search engine with the same queries used above for the research literature (cf. Figure 2.3). We also apply a back-cutting strategy at the end of the fifth page of the result for each query. Hence, there are 1750 entries for this group of search queries, the same number as the paper selection queries.

Complementarily, we select six machine learning-related Python libraries, namely TensorFlow, PyTorch, Scikit-Learn, Pandas, NumPy, and SciPy, combine them with the code quality-related keywords mentioned in Section 2.3.1 and form a new group of search queries. Python is widely used for machine learning[1] and the six libraries are the most popular machine learning libraries [2], covering the two most important steps in machine learning application development – data processing and model training. For this group of search queries, we reach a level of saturation after analyzing the first result page. Therefore, we consult

[1]State of Data Science and Machine Learning 2021. https://www.kaggle.com/kaggle-survey-2021
[2]15 Python Libraries for Data Science You Should Know. https://www.dataquest.io/blog/15-python-libraries-for-data-science/

the first ten results (i.e., first page) the Google search engine provides. There are 420 entries ($6 \times 7 \times 10$) for this group of search queries. In total, there are 2170 entries for grey literature mining.

Since not all entries contain actionable coding advice, we select entries by 1) reading the title, 2) reading the first summary, and 3) reading the whole article. Many articles mention some common patterns in machine learning, but most of them are duplicated and are general advice that do not contain code-level pitfalls.

In the end, we identify eight cornerstone blog posts that contribute to the code smell catalog, as listed in Section A.1 in the Appendix: (1), (2), (3), (4), (5), (6), (7), (8).

2.3.3 Reusing Existing Bug Datasets

We reuse the dataset provided in the work by Zhang et al. [33] to mine code smells in Tensorflow applications. Zhang et al. mined the Tensorflow application bugs, analyzed the bugs pattern using 88 Stack Overflow posts as well as 87 GitHub commits and provided a replication package for these bugs (hereinafter called *"TensorFlow Bugs" replication package*). We reuse their replication package to extract recurrent pitfalls that may generalize to other projects and thus should be documented as code smells.

2.3.4 Complementary Stack Overflow Mining

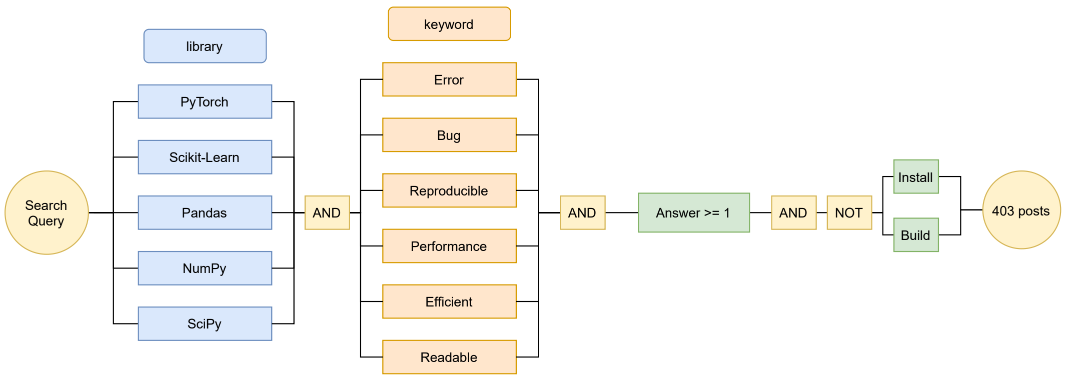


Figure 2.4: Search Query for Stack Overflow Mining

After reusing the existing bug datasets, we apply a similar study method to other machine learning libraries. We only check the posts on Stack Overflow at this part without GitHub commits. This is because all the issues have a similar pattern in GitHub and Stack Overflow, as noted by [12] and verified in the TensorFlow Bugs replication package [33].

1) Library Selection: We use five libraries: PyTorch, Scikit-Learn, Pandas, NumPy, and SciPy, excluding TensorFlow from the six libraries mentioned in Section 2.3.2.

2) Keyword Selection: To retrieve relevant entries from Stack Overflow we had to redefine our search keywords. We did so because Stack Overflow seldom hosts discussions that directly mention technical debt, smells or refactorings. Entries are mostly related to

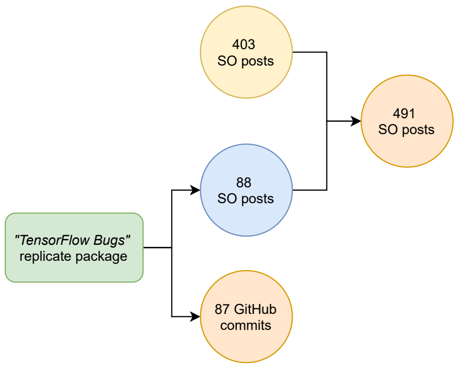


Figure 2.5: Total Number of Stack Overflow Posts and GitHub Commits

low-level and straight-to-the-point problems – e.g., performance issues or errors. Hence, keywords had to be adjusted. When collecting posts from Stack Overflow, we use six keywords that are related to typical software quality issues [5]: Error, Bug, Reproducible, Performance, Efficient, Readable.

3) Applying Search Terms: We use the notation provided by Stack Overflow to implement the refined queries listed in Figure 2.4. It has the following format:

```
[library] keyword answer:1 -install -build
```

- **`library`** refers to the name of the library we are targeting (e.g., PyTorch, Pandas, etc.).
- **`keyword`** refers to the software quality aspect (e.g., error, bug, etc.).
- **`answer:1`** refers to the entries having at least one answers.
- **-install -build** these are terms that we exclude from the result set. We filter out `install` and `build` because they typically yield results related to configuration and not the codebase.

Using the keyword "Error" to search gets the maximum number of posts among all libraries. Therefore, we rank all the posts by their votes after applying the term in the search engine, selecting the top 50 posts with the "Error" keyword, and selecting the top 10 posts with each of the rest keywords. If the number of posts is fewer than 10, we select all the posts. Then, we delete the duplicated posts for each library. In the end, we get 81, 68, 84, 88, and 82 posts respectively for PyTorch, Scikit-learn, Pandas, NumPy, and SciPy. Together with the *"TensorFlow Bugs" replication package* [33], we have 87 GitHub commits and 491 Stack Overflow posts in our dataset, as presented in Figure 2.5.

2.3.5 Validation

The first author collects all code smells from the empirical study (including paper, grey literature, GitHub and Stack Overflow mining) and discusses the code smell catalog with the second author. We conducted discussion meetings consisting of an introductory discussion of each smell, followed by the analysis of code examples where the code issue had been identified, and the collection of further evidence. We look for references in academic and grey literature that support that particular smell. In total, the first author collected 31 code smells, from which 9 were dropped.

2.4 Results

In this section, we describe 22 machine learning-specific code smells collected from our empirical study. For each smell, we provide a general description followed by the context of the smell, the problem of its occurrence, and the solution. In the end, we summarise all the smells, including the references supporting the smell, the stage of the machine learning pipeline where they are more relevant, and the main effect that arises from having those smells.

We use the notation (n) to cite entries from grey literature, as listed in Appendix A.1, where n refers to the nth element in the list.

Table 2.2: Code Smell Catalog

Code Smell	Pipeline Stage	Effect	Type	Literature	Grey Literature	GitHub Commits	SO Posts
Unnecessary Iteration	Data Cleaning	Efficiency	Generic	[9]	(6)(14)	(13)	
NaN Equivalence Comparison Misused	Data Cleaning	Error-prone	Generic	[9]			
Chain Indexing	Data Cleaning	Error-prone & Efficiency	API-Specific: Pandas		(30)		(31)(32)
Columns and DataType Not Explicitly Set	Data Cleaning	Readability	Generic		(7)		
Empty Column Misinitialization	Data Cleaning	Robustness	Generic		(7)		
Merge API Parameter Not Explicitly Set	Data Cleaning	Readability & Error-prone	Generic		(7)		
In-Place APIs Misused	Data Cleaning	Error-prone	Generic	[9]		(11)	
Dataframe Conversion API Misused	Data Cleaning	Error-prone	API-Specific: Pandas				(33)
Matrix Multiplication API Misused	Data Cleaning	Readability	API-Specific: NumPy		(35)		(34)
No Scaling before Scaling-Sensitive Operation	Feature Engineering	Error-prone	Generic		(2)(16)		(17)
Hyperparameter Not Explicitly Set	Model Training	Error-prone & Reproducibility	Generic	[9][4][10]			
Memory Not Freed	Model Training	Memory Issue	Generic	[10]	(5)(19)		(20)
Deterministic Algorithm Option Not Used	Model Training	Reproducibility	Generic	[4]	(9)		
Randomness Uncontrolled	Model Training & Model Evaluation	Reproducibility	Generic	[4]	(1)(5)(9)		(26)
Missing the Mask of Invalid Value	Model Training	Error-prone	Generic	[33][10]			(21)(22)(23)(24)
Broadcasting Feature Not Used	Model Training	Efficiency	Generic		(6)		
TensorArray Not Used	Model Training	Efficiency & Error-prone	API-Specific: TensorFlow 2		(6)		
Training / Evaluation Mode Improper Toggling	Model Training	Error-prone	Generic		(36)		
Pytorch Call Method Misused	Model Training	Robustness	API-Specific: PyTorch		(5)		
Gradients Not Cleared before Backward Propagation	Model Training	Error-prone	API-Specific: PyTorch		(36)		
Data Leakage	Model Evaluation	Error-prone	Generic	[9]	(8)		(27)
Threshold-Dependent Validation	Model Evaluation	Robustness	Generic	[21]			

2.4.1 Unnecessary Iteration

Avoid unnecessary iterations. Use vectorized solutions instead of loops.

Context Loops are typically time-consuming and verbose, while developers can usually use some vectorized solutions to replace the loops.

Problem As stated in the Pandas documentation (14): "Iterating through pandas objects is generally slow. In many cases, iterating manually over the rows is not needed and can be avoided”. In (6), it is also stated that the slicing operation with loops in TensorFlow is slow, and there is a substitute for better performance.

Solution Machine learning applications are typically data-intensive, requiring operations on data sets rather than an individual value. Therefore, it is better to adopt a vectorized solution instead of iterating over data. In this way, the program runs faster and code complexity is reduced, resulting in more efficient and less error-prone code [9]. Pandas’ built-in methods (e.g., join, groupby) are vectorized. It is therefore recommended to use Pandas built-in methods as an alternative to loops. In TensorFlow, using the *tf.reduce_sum()* API to perform reduction operation is much faster than combining slicing operation and loops.

2.4.2 NaN Equivalence Comparison Misused

The *NaN* equivalence comparison is different to *None* comparison. The result of *NaN* == *NaN* is False (40).

Context *NaN* equivalence comparison behaves differently from *None* equivalence comparison.

Problem While *None* == *None* evaluates to *True*, *np.nan* == *np.nan* evaluates to *False* in NumPy. As Pandas treats *None* like *np.nan* for simplicity and performance reasons, a comparison of *DataFrame* elements with *np.nan* always returns *False* [9]. If the developer is not aware of this, it may lead to unintentional behaviours in the code.

Solution Developers need to be careful when using the *NaN* comparison.

2.4.3 Chain Indexing

Avoid using chain indexing in Pandas.

Context In Pandas, *df[“one”][“two”]* and *df.loc[:,(“one”,“two”)]* give the same result. *df[“one”][“two”]* is called chain indexing.

Problem Using chain indexing may cause performance issues as well as error-prone code (30)(31)(32). For example, when using *df[“one”][“two”]*, Pandas sees this operation as two events: call *df[“one”]* first and call *[“two”]* based on the result the previous operation gets. On the contrary, *df.loc[:,(“one”,“two”)]* only performs a single call. In this way, the second approach can be significantly faster than the first one. Furthermore, assigning to the product of chain indexing has inherently unpredictable results. Since Pandas makes no guarantees on whether *df[“one”]* will return a view or a copy, the assignment may fail.

Solution Developers using Pandas should avoid using chain indexing.

2.4.4 Columns and DataType Not Explicitly Set

Explicitly select columns and set *DataType* when importing data.

Context In Pandas, all columns are selected by default when a *DataFrame* is imported from a file or other sources. The data type for each column is defined based on the default *dtype* conversion.

Problem If the columns are not selected explicitly, it is not easy for developers to know what to expect in the downstream data schema (7). If the datatype is not set explicitly, it may silently continue the next step even though the input is unexpected, which may cause errors later. The same applies to other data importing scenerios.

Solution It is recommended to set the columns and *DataType* explicitly in data processing.

2.4.5 Empty Column Misinitialization

When a new empty column is needed in a *DataFrame* in Pandas, use the *NaN* value in Numpy instead of using zeros or empty strings.

Context Developers may need a new empty column in *DataFrame*.

Problem If they use zeros or empty strings to initialize a new empty column in Pandas, the ability to use methods such as *.isnull()* or *.notnull()* is retained (7). This might also happens to initializations in other data structure or libraries.

Solution Use *NaN* value (e.g. "*np.nan*") if a new empty column in a *DataFrame* is needed. Do not use "filler values" such as zeros or empty strings.

2.4.6 Merge API Parameter Not Explicitly Set

Explicitly specify the parameters for merge operations. Specifically, explicitly specify *on*, *how* and *validate* parameter for *df.merge()* API in Pandas for better readability.

Context The *df.merge()* API merges two *DataFrame*s in Pandas.

Problem Although using the default parameter can produce the same result, explicitly specify *on* and *how* produce better readability (7). The parameter *on* states which columns to join on, and the parameter *how* describes the join method (e.g., outer, inner). Also, the *validate* parameter will check whether the merge is of a specified type. If the developer assumes the merge keys are unique in both left and right datasets, but that is not the case, and he does not specify this parameter, the result might silently go wrong. The merge operation is usually computationally and memory expensive. It is preferable to do the merging process in one stroke for performance consideration.

Solution Developer should explicitly specify the parameters for merge operation.

2.4.7 In-Place APIs Misused

Remember to assign the result of an operation to a variable or set the in-place parameter in the API.

Context Data structures can be manipulated in mainly two different approaches: 1) by applying the changes to a copy of the data structure and leaving the original object intact, or 2) by changing the existing data structure (also known as in-place).

Problem Some methods can adopt in-place by default, while others return a copy. If the developer assumes an in-place approach, he will not assign the returned value to any variable. Hence, the operation will be executed, but it will not affect the final outcome. For example, when using the Pandas library, the developer may not assign the result of *df.dropna()* to a variable. He may assume that this API will make changes on the original *DataFrame* and not set the in-place parameter to be *True* either. The original *DataFrame* will not be updated in this way [9]. In the *"TensorFlow Bugs" replication package*, we also found an example (11) where the developer thought *np.clip()* is an in-place operation and used it without assigning it to a new variable.

Solution We suggest developers check whether the result of the operation is assigned to a variable or the in-place parameter is set in the API. Some developers hold the view that the in-place operation will save memory. However, this is a misconception in the Pandas library because the copy of the data is still created. In PyTorch, the in-place operation does save GPU memory, but it risks overwriting the values needed to compute the gradient (10).

2.4.8 Dataframe Conversion API Misused

Use *df.to_numpy()* in Pandas instead of *df.values()* for transform a *DataFrame* to a NumPy array.

Context In Pandas, *df.to_numpy()* and *df.values()* both can turn a *DataFrame* to a NumPy array.

Problem As noted in (33), *df.values()* has an inconsistency problem. With *.values()* it is unclear whether the returned value would be the actual array, some transformation of it, or one of the Pandas custom arrays. However, the *.values()* API has not been not deprecated yet. Although the library developers note it as a warning in the documentation, it does not log a warning or error when compiling the code if we use *.value()*.

Solution When converting *DataFrame* to NumPy array, it is better to use *df.to_numpy()* than *df.values()*.

2.4.9 Matrix Multiplication API Misused

When the multiply operation is performed on two-dimensional matrixes, use *np.matmul()* instead of *np.dot()* in NumPy for better semantics.

Context When the multiply operation is performed on two-dimensional matrixes, *np.matmul()* and *np.dot()* give the same result, which is a matrix.

Problem In mathematics, the result of the dot product is expected to be a scalar rather than a vector (39). The *np.dot()* returns a new matrix for two-dimensional matrixes multiplication, which does not match with its mathematics semantics. Developers sometimes use *np.dot()* in scenarios where it is not supposed to, e.g., two-dimensional multiplication.

Solution When the multiply operation is performed on two-dimensional matrixes, *np.matmul()* is preferred over *np.dot()* for its clear semantic (34)(35).

2.4.10 No Scaling before Scaling-Sensitive Operation

Check whether feature scaling is added before scaling-sensitive operations.

Context Feature scaling is a method of aligning features from various value ranges to the same range (18).

Problem There are many operations sensitive to feature scaling, including Principal Component Analysis (PCA), Support Vector Machine (SVM), Stochastic Gradient Descent (SGD), Multi-layer Perceptron classifier and L1 and L2 regularization (2)(16). Missing scaling can lead to a wrong conclusion. For example, if one variable is on a larger scale than another, it will dominate the PCA procedure. Therefore, PCA without feature scaling can produce a wrong principal component result.

Solution To avoid bugs, whether feature scaling is added before scaling-sensitive operations should be checked.

2.4.11 Hyperparameter Not Explicitly Set

Hyperparameters should be set explicitly.

Context Hyperparameters are usually set before the actual learning process begins and control the learning process [9]. These parameters directly influence the behavior of the training algorithm and therefore have a significant impact on the model's performance.

Problem The default parameters of learning algorithm APIs may not be optimal for a given data or problem, and may lead to local optima. In addition, while the default parameters of a machine learning library may be adequate for some time, these default parameters may change in new versions of the library. Furthermore, not setting the hyperparameters explicitly is inconvenient for replicating the model in a different programming language.

Solution Hyperparameters should be set explicitly and tuned for improving the result's quality and reproducibility.

2.4.12 Memory Not Freed

Free memory in time.

Context Machine learning training is memory-consuming, and the machine's memory is always limited by budget.

Problem If the machine runs out of memory while training the model, the training will fail.

Solution Some APIs are provided to alleviate the run-out-of-memory issue in deep learning libraries. TensorFlow's documentation notes that if the model is created in a loop, it is suggested to use *clear_session()* in the loop (19). Meanwhile, the GitHub repository *pytorch-styleguide* recommends using *.detach()* to free the tensor from the graph whenever possible (5). The *.detach()* API can prevent unnecessary operations from being recorded and therefore can save memory (38). Developers should check whether they use this kind of APIs to free the memory whenever possible in their code.

2.4.13 Deterministic Algorithm Option Not Used

Set deterministic algorithm option to *True* during the development process, and use the option that provides better performance in the production.

Context Using deterministic algorithms can improve reproducibility.

Problem The non-deterministic algorithm cannot produce repeatable results, which is inconvenient for debugging.

Solution Some libraries provide APIs for developers to use the deterministic algorithm. In PyTorch, it is suggested to set *torch.use_deterministic_algorithms(True)* when debugging (9). However, the application will perform slower if this option is set, so it is suggested not to use it in the deployment stage.

2.4.14 Randomness Uncontrolled

Set random seed explicitly during the development process whenever a possible random procedure is involved in the application.

Context There are several scenarios involving random seeds. In some algorithms, randomness is inherently involved in the training process. For the cross-validation process in the model evaluation stage, the dataset split by some library APIs can vary depending on random seeds.

Problem If the random seed is not set, the result will be irreproducible, which increases the debugging effort. In addition, it will be difficult to replicate the study based on the previous one. For example, in Scikit-Learn, if the random seed is not set, the random forest algorithm may provide a different result every time it runs, and the dataset split by cross-validation splitter will also be different in the next run (8).

Solution It is recommended to set global random seed first for reproducible results in Scikit-Learn, Pytorch, Numpy and other libraries where a random seed is involved (1)(9). Specifically, *DataLoader* in PyTorch needs to be set with a random seed to ensure the data is split and loaded in the same way every time running the code.

2.4.15 Missing the Mask of Invalid Value

Add a mask for possible invalid values. For example, developers should wrap the argument for *tf.log()* with *tf.clip()* to avoid the argument turning to zero.

Context In deep learning, the value of the variable changes during training. The variable may turn into an invalid value for another operation in this process.

Problem Several posts on Stack Overflow talk about the pitfalls that are not easy to discover caused by the input of the log function approaching zero (21)(22)(23)(24). In this kind of programs, the input variable turns to zero and becomes an invalid value for *tf.log()*, which raises an error during the training process. However, the error's stack trace did not directly point to the line of code that the bug exists [33]. This problem is not easy to debug and may take a long training time to find.

Solution The developer should check the input for the *log* function or other functions that have special requirements for the argument and add a mask for them to avoid the invalid value. For example, developer can change *tf.log(x)* to *tf.log(tf.clip_by_value(x,1e-10,1.0))*. If the value of *x* becomes zero, i.e., lower than the lowest bound 1e-10, the *tf.clip_by_value()* API will act as a mask and outputs 1e-10. It will save time and effort if the developer could identify this smell before the code run into errors.

2.4.16 Broadcasting Feature Not Used

Use the broadcasting feature in deep learning code to be more memory efficient.

Context Deep learning libraries like PyTorch and TensorFlow supports the element-wise broadcasting operation.

Problem Without broadcasting, tiling a tensor first to match another tensor consumes more memory due to the creation and storage of a middle tiling operation result (6)(41).

Solution With broadcasting, it is more memory efficient. However, there is a trade-off in debugging since the tiling process is not explicitly stated.

2.4.17 TensorArray Not Used

Use *tf.TensorArray()* in TensorFlow 2 if the value of the array will change in the loop.

Context Developers may need to change the value of the array in the loops in TensorFlow.

Problem If the developer initializes an array using *tf.constant()* and tries to assign a new value to it in the loop to keep it growing, the code will run into an error. The developer can fix this error by the low-level *tf.while_loop()* API (6). However, it is inefficient coding in this way. A lot of intermediate tensors are built in this process.

Solution Using *tf.TensorArray()* for growing array in the loop is a better alternative for this kind of problem in TensorFlow 2. Developers should use new data types from libraries for more intelligent solutions.

2.4.18 Training / Evaluation Mode Improper Toggling

Call the training mode in the appropriate place in deep learning code to avoid forgetting to toggle back the training mode after the inference step.

Context In PyTorch, calling *.eval()* means we are going into the evaluation mode and the *Dropout* layer will be deactivated.

Problem If the training mode did not toggle back in time, the *Dropout* layer would not be used in some data training and thus affect the training result (36). The same applies to TensorFlow library.

Solution Developers should call the training mode in the right place to avoid forgetting to switch back to the training mode after the inference step.

2.4.19 Pytorch Call Method Misused

Use *self.net()* in PyTorch to forward the input to the network instead of *self.net.forward()*.

Context Both *self.net()* and *self.net.forward()* can be used to forward the input into the network in PyTorch.

Problem In PyTorch, *self.net()* and *self.net.forward()* are not identical. The *self.net()* also deals with all the register hooks, which would not be considered when calling the plain *.forward()* (5).

Solution It is recommended to use *self.net()* rather than *self.net.forward()*.

2.4.20 Gradients Not Cleared before Backward Propagation

Use *optimizer.zero_grad()*, *loss_fn.backward()*, *optimizer.step()* together in order in PyTorch. Do not forget to use *optimizer.zero_grad()* before *loss_fn.backward()* to clear gradients.

Context In PyTorch, *optimizer.zero_grad()* clears the old gradients from last step, *loss_-fn.backward()* does the back propagation, and *optimizer.step()* performs weight update using the gradients.

Problem If *optimizer.zero_grad()* is not used before *loss_fn.backward()*, the gradients will be accumulated from all *loss_fn.backward()* calls and it will lead to the gradient explosion, which fails the training (36).

Solution Developers should use *optimizer.zero_grad()*, *loss_fn.backward()*, *optimizer.step()* together in order and should not forget to use *optimizer.zero_grad()* before *loss_-fn.backward()*.

2.4.21 Data Leakage

Use *Pipeline()* API in Scikit-Learn or check data segregation carefully when using other libraries to prevent data leakage.

Context The data leakage occurs when the data used for training a machine learning model contains prediction result information (28).

Problem Data leakage frequently leads to overly optimistic experimental outcomes and poor performance in real-world usage [9].

Solution There are two main sources of data leakage: leaky predictors and a leaky validation strategy (29). Leaky predictors are the cases in which some features used in training are modified or generated after the goal value has been achieved. This kind of data leakage can only be inspected at the data level rather than the code level. Leaky validation strategy refers to the scenario where training data is mixed with validation data. This fault can be checked at the code level. One best practice in Scikit-Learn is to use the *Pipeline()* API to prevent data leakage.

2.4.22 Threshold-Dependent Validation

Use threshold-independent metrics instead of threshold-dependent ones in model evaluation.

Context The performance of the machine learning model can be measured by different metrics, including threshold-dependent metrics (e.g., F-measure) or threshold-independent metrics (e.g., Area Under the Curve (AUC)).

Problem Choosing a specific threshold is tricky and can lead to a less-interpretable result [21].

Solution Threshold-independent metrics are more robust and should be preferred over threshold-dependent metrics.

2.5 Discussions and Implications

The code smell catalog summarized from the empirical study is presented in Table 3.5. We collected 22 code smells in total and linked the smells to four pipeline stages: Data

Cleaning, Feature Engineering, Model Training, and Model Evaluation. Possible impacts of the smells on application codes include being error-prone, less efficient, less reproducible, causing memory issues, less readable, and less robust. In addition, 16 smells are generic smells, while 6 are API-specific smells. Generic smells occur regardless of which library the developer uses, while API-specific smells depend on a specific library API design.

The catalog helps understand prevalent flaws in machine learning application development by investigating recurrent code issues from various sources. Since many data scientists do not have a software engineering background and are not up-to-date with the best practices from the software engineering field, our catalog of smells mitigates this barrier by providing some guidelines when developing machine learning applications.

Machine learning libraries are being regularly improved with new versions. We reused the "TensorFlow Bugs" replication package and found that many instances have already been deprecated because TensorFlow has upgraded to version 2. Hence, we expect that new API-specific code smells will appear with new versions and library features. In fact, our results showcase that most API-related smells are only reported by grey literature in general instead of literature. We argue that collecting a catalog of code smells helps in promoting a continuous effort between practitioners and academics.

The ecosystem of AI frameworks is changing very fast, which means that some smells might become obsolete in the meantime. In our catalog, we anticipate that three smells can be considered temporary smells: *Dataframe Conversion API Misused*, *Matrix Multiplication API Misused* and *Gradients Not Cleared before Backward Propagation*. While other smells are perceived to last for a long time, temporary smells might be deprecated in a few years. Yet, these three smells are important and should be flagged to help practitioners prevent issues downstream.

2.5.1 Implications to Data Scientists and Machine Learning Application Developers

This catalog contains smells from heterogeneous sources, existing in different stages, and will trigger various effects. For instance, the *Unnecessary Iteration* code smell describes the inefficient code structure and it often occurs at data cleaning stages. Another code smell *Hyperparameter Not Explicitly Set* indicates irreproducible code and it is at model training stage. Data scientists and machine learning application developers can check these aspects while checking their code.

Some code smells appear multiple times in different sources – both from academic and grey literature. For example, *Missing the Mask of Invalid Value* is referenced in two instances of academic literature and four from Stack Overflow posts. Practitioners can use this as an indication of the relevance of smells and use the references to learn more about them.

Machine learning application developers, especially data scientists with little software engineering experience, can use the catalog to build awareness of the pitfalls and best practices highlighted in this study and strive to prevent these errors from their code. We assume that knowing code smells can shorten the time of development and help assure high-quality software in production. Future work will validate whether eliminating these code smells will lead to more accurate results during training, better hyperparameter optimization, clearer and higher quality code, and less maintenance effort.

2.5.2 Implication to Machine Learning Library Developers

Some smells in the catalog stem from the fact that APIs require a particular usage pattern that is not intuitive to their users. For example, the smell *Dataframe Conversion API Misused* smell could be eradicated if the API method *df.values()* would be deprecated and completely replaced by *df.to_numpy()*. In another example, the *Gradients Not Cleared before Backward Propagation* smell could be avoided if the API already took care of combining gradient clear and backward propagation for its users, since this is the commended approach. Hence, our results show how the design of library APIs plays an important role in avoiding potential issues in projects.

Some of the smells we identify are reported in the official documentation of the libraries. Yet, there is still code being created that does not comply with these recommendations. For example, the effect of index chaining (cf. Section 2.4.3) appears in code examples provided by Stack Overflow although it is explained in the Pandas documentation. This indicates that many developers are struggling to follow the documentation strictly. It might stem from the fast iteration cycles in the development process of teams or from the developer's lack of experience in that particular library. We argue that passively indicating warnings on documentation might not be sufficient. It is important that library developers and maintainers are actively engaging in community forums, such as Stack Overflow, to help the community avoid non-obvious issues.

Finally, it is important that library maintainers promote and reach out to existing projects that aim at helping the development of machine learning software – i.e., static code analysis tools, testing tools, quality auditors, experiment trackers, and so on. Library developers know better than anyone what is the optimal way of leveraging their libraries. Hence, their contribution is crucial in the development of coding tools that support best practices.

2.5.3 Implication to Code Analysis Tool Developers

As some code smells cannot be addressed by designing better APIs, the static analysis tool can help promote best practices and warn pitfalls to the application developers.

This research serves as the base for future work on automated tools to detect these unwanted code patterns. Automated tools can minimize the developer's effort to discover the code smells and eliminate them, providing support for code quality assurance. Because humans are occasionally forgetful, it is preferable to have a technology that expressly checks whether best practices are being followed.

In addition, we observe that some code smells are related to the context. This is aligned with previous work that proposes context-aware code analysis tools for machine learning applications [13]. For example, PyTorch library developers recommend application developers to use the deterministic option during the development but not set it in the production code due to the consideration for performance. Therefore, the automated tool can have different configuration settings. For example, according to the pipeline stage, it can have a development setting and a deployment setting.

2.5.4 Implication to Students

As mentioned by [9], many graduates in the industry do not have formal education on machine learning application development since it requires a combination of software engineering and data science practices. Students can use this catalog to learn more about the common anti-patterns in machine learning application development and prepare for future jobs.

2.6 Threats to validity

In our study, the first author performs manual code smell inspections, which can be biased due to the different understanding of machine learning code. To alleviate this threat, the second author reviews all instances of code smells, followed by a discussion between the first two authors.

In both the academic and grey literature survey, the initial selection of keywords in the search query might miss relevant entries. To mitigate this threat, we iteratively refine the search keywords based on retrieved relevant content. In addition, we apply forward and backward snowballing to complement the search.

Moreover, since we use a back-cutting strategy on the grey literature search, the quality of the search results depends on the accuracy of the Google search engine's relevance sorting algorithm, which is beyond our control. The results are collected from the first author's Google account, and they might vary across users. However, we believe that this has minimal impact on the result set of our study.

When mining Stack Overflow entries and GitHub commits, we inspect 88 GitHub commits and 491 Stack Overflow posts in total. It is unclear how generalizable our results are. To cover the most common mistakes in the machine learning application practice in a generalizable way, we use the "highest voted" criteria to select instances from Stack Overflow. We anticipate that less-voted instances may also contain machine learning code issues. Increasing the result set would not be feasible in a manual inspection. Yet, we argue that the highest voted instances provide an interesting snapshot with the most relevant issues.

This study focuses on six Python machine learning libraries and frameworks. There are several other machine learning frameworks that might lead to particular code smells. However, it would not be feasible to apply our methodology in all the libraries out there. Hence, we reduce this threat by selecting the most popular frameworks.

Finally, we acknowledge that there are more warnings within libraries documentation that can become code smells. However, we only consider warnings that have allegedly led to real code issues, as observed in other sources (e.g., Stack Overflow).

2.7 Conclusions and Future Work

In this paper, we conducted an empirical study to collect the code smell specific for machine learning applications. We collected the code smells from various sources, including mining 1750 papers, mining 2170 grey literature entries, using the existing bugs datasets including 88 Stack Overflow posts and 87 GitHub commits and gathering 403 complementary Stack Overflow posts. We analyzed the pitfalls mentioned in the posts and decided whether to take it as a code smell. We collected 22 code smells, including general and API-specific smells. We also classified the code smell by different pipeline stages and its effect. We want to raise the discussion about machine learning-specific code smell and help improve code quality in the machine learning community in this way.

Future work will include a quantitative large-scale validation of the code smell catalog. We would like to interview machine learning practitioners and mine code changes in GitHub repositories to validate and improve the catalog. In addition, we plan to implement a static analysis tool that automatically detect these smells to promote best practices in machine learning code. Finally, it would be interesting to study the prevalence of these code smells in real-world machine learning applications and explore the benefits of using a catalog of machine learning-specific code smells.

Chapter 3

Automated Detection of ML-Specific Code Smells

3.1 Introduction

The popularity of machine learning (ML) algorithms has exploded in recent years, to the point where it has become a critical component of the product. Machine learning has been integrated into autonomous driving, medical decision-making, and financial fraud detection systems. As the core component of the system, machine learning code should be thoroughly assessed before it can be deemed trustworthy.

However, due to the dynamic nature of machine learning programs, their reliability can be challenging to ensure [3]. The algorithmic result of the machine learning applications could be nondeterministic, and the bugs might occur after a lengthy training period [33]. This may cause the end users to fail to trust the machine learning model and the developers to spend an excessive amount of time debugging. The reliability of machine learning systems, therefore, has become an important topic.

Traditional software systems resort to several tools to ensure system reliability. For example, static analysis tools are used to improve code quality and alleviate technical debt. However, few tools are developed and adopted to mitigate technical debt issues in machine learning projects. According to [23], static analysis tools, one of the best practices in the traditional software system, have a low adoption rate in machine learning projects. One reason for the low adoption rate is the significant number of false positives generated when using such tools [28]. Due to the fact that common code patterns are not precisely linked with machine learning code, a static analysis tool may produce a large number of false positives.

The findings from Chapter 2 indicate that some machine learning-specific issues are stated in papers and grey literature. We do not yet know how widespread these coding pitfalls are in the public repositories. We are interested in finding out how frequently those issues occur, to what extent we can solve these problems by developing a tool specific for machine learning code smells and what are the differences between notebooks and regular projects for machine learning code quality. Therefore, we define the following research questions:

RQ1: How prevalent are the machine learning-specific code smells?

RQ2: How accurate is the machine learning-specific code smell detection tool dslinter?

RQ3: Is the prevalence of code smells different in Python notebooks and regular Python projects?

To answer these research questions, we develop a static analysis tool specifically for machine learning code named `dslinter`. We apply the tool to four datasets to assess how widespread the code smells are in machine learning projects and to what extent the tool can improve the software quality.

The contributions of this research are as follows:

1) Extending an open-source static analysis tool `dslinter` to detect the machine learning-specific code smells, which is available at GitHub [1] and PyPI [2].

2) An empirical study on the prevalence of code smells in both public notebook datasets and regular project datasets.

3) A replication package [3] for the empirical study.

3.2 Related Work

In this section, we introduce the related work. We cover the related work in three dimensions: tools for improving the quality of machine learning applications, static code analysis tools, and code smell prevalence studies.

3.2.1 Tools for improving the quality of machine learning applications

Recently, some tools for ensuring the quality of machine learning software have been developed.

A novel quality control method was proposed in Rajbahadur et al.'s paper [21]. They built a model-driven tool called Pitfalls Analyzer to detect the pitfall in the data science pipeline. By running their tool on 11 data science pipelines, they verified that the tool can successfully identify every pitfall they identified manually.

Oort et al. proposed and developed a static analysis tool for machine learning project management [30]. Their tool checks whether the SE4ML best practices are followed and provides suggestions on low abstraction level best practices, including whether linters are used to ensure the code quality. They evaluated the concept of project smells and the tool in a global bank ING, and found that the developers think the code quality is very important for a production-ready project.

Hynes et al. introduced a data linter to check the data quality in deep neural networks (DNNs) [11]. They defined 13 rules, which can be roughly grouped into three categories: miscodings of data, outliers, and packaging errors. The linter helps increase the DNN model's precision from 0.48 to 0.59 in the most promising outcome.

These tools significantly help with machine learning experiment management and software quality assurance. However, tools for assisting with machine learning-specific code

[1]dslinter at GitHub: `https://github.com/SERG-Delft/dslinter`

[2]dslinter at PyPI: `https://pypi.org/project/dslinter/`

[3]Replication Package: `https://github.com/Hynn01/dslinter-experiments`

quality checks are rare. Our study fills this gap by developing a static code analysis tool for machine learning applications.

3.2.2 Static code analysis tools

The static code analysis tool is one solution to help ensure code quality.

Zakas et al. created ESlint to help developers find and fix potential problems in JavaScript code [32]. It is built into most editors nowadays and can be incorporated into the continuous integration pipeline [4]. Tómasdóttir et al. studied the adoption of JavaScript and found that developers usually adopt linter for several reasons: code consistency maintenance, errors prevention, discussion time reduction, complex code elimination, and code review automation [28].

Popa et al. developed Pylint for detecting the violation of PEP8 coding rules [27]. Pylint uses Abstract Syntax Tree analysis to find warning signs and errors that can be relayed to the user as well as return an overall grade reflecting on the software quality level [5]. It allows developers to easily extend its core functionality and augment it with new rules and algorithms.

Besides Pylint, developers can choose Flake8 and Black for static code analysis for Python projects. However, while these tools are highly customisable, none of these offers any machine learning-specific insights. Even though static analysis tools are widely used, [23] reports that their adoption is low in machine learning teams. According to their investigation, partitioners complain about excessive false positives, partly because the rules are unrelated to the machine learning context.

The detection of a few machine learning coding issues has been automated in the Pylint extension `dslinter`, a prototype which looks for code smells in machine learning project source code [7]. However, the number of smells detected by `dslinter 1.0.0` is limited and many common smells, as derived from popular developer practices, still exist that are not identified with this tool. Due to these limitations, we are motivated to expand the smell set of `dslinter` and increase its power, making it even more useful to developers.

3.2.3 Code smell prevalence studies

Some researchers have already researched code smells in a specific context.

Muse et al. studied the prevalence of SQL code smells in data-intensive systems [18]. 150 open-source projects were collected and examined to find SQL code smells. Their study showed a high diffusion of code smells in data-intensive software.

Bavota et al. researched the prevalence of code smells specific to tests such as unit tests [1]. They performed an exploratory study on 27 software systems and a controlled experiment involving 61 participants with different experience levels. Their study finds that 86% of JUnit tests exhibit at least one smell, which indicates the test smells are pervasive. Their study also verified that the test smell could strongly harm the program's comprehension and maintenance.

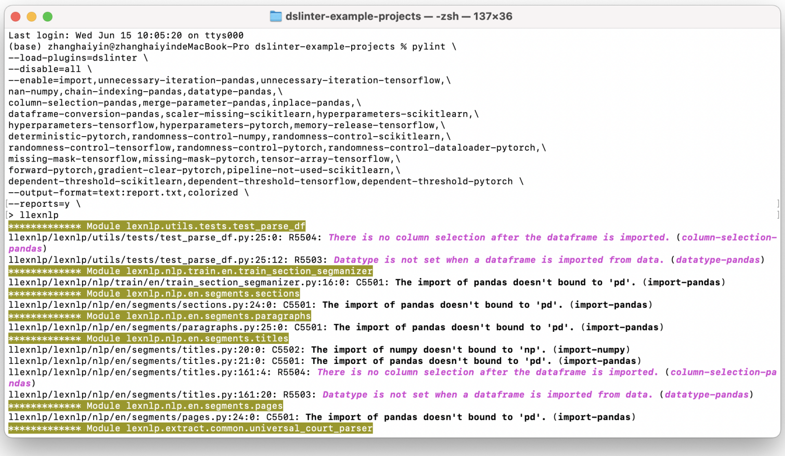
[4]ESlint Documentation: `https://eslint.org/`

[5]Pylint Documentation: `http://pylint.pycqa.org/en/latest/intro.html`

Currently, there are rarely studies applying static analysis tools to research machine learning code quality. Van Oort et al. applied Pylint to analyze 74 machine learning projects, implying that Pylint is unable to analyze whether the ML library APIs are correctly used, which reduces the power of Continuous Integration (CI) for the machine learning systems [29].

In Chapter 2 we studied the code smells in both papers and grey literature, and identified 22 machine learning-specific code smells. The prevalence of these machine learning-specific code smells in public codebases has not yet been studied. Our new version of `dslinter` not only helps developers ensure machine learning code quality but can also be used to research the prevalence of machine learning-specific code smell.

3.3 Dslinter



```
Last login: Wed Jun 15 10:05:20 on ttys000
(base) zhanghaiyin@zhanghaiyindeMacBook-Pro dslinter-example-projects % pylint \
--load-plugins=dslinter \
--disable=all \
--enable=import,unnecessary-iteration-pandas,unnecessary-iteration-tensorflow,\
nan-numpy,chain-indexing-pandas,datatype-pandas,\
column-selection-pandas,merge-parameter-pandas,inplace-pandas,\
dataframe-conversion-pandas,scaler-missing-scikitlearn,hyperparameters-scikitlearn,\
hyperparameters-tensorflow,hyperparameters-pytorch,memory-release-tensorflow,\
deterministic-pytorch,randomness-control-numpy,randomness-control-scikitlearn,\
randomness-control-tensorflow,randomness-control-pytorch,randomness-control-dataloader-pytorch,\
missing-mask-tensorflow,missing-mask-pytorch,tensor-array-tensorflow,\
forward-pytorch,gradient-clear-pytorch,pipeline-not-used-scikitlearn,\
dependent-threshold-scikitlearn,dependent-threshold-tensorflow,dependent-threshold-pytorch \
--output-format=text:report.txt,colorized \
--reports=y \
> llexnlp
************* Module lexnlp.utils.tests.test_parse_df
llexnlp/lexnlp/utils/tests/test_parse_df.py:25:0: R5504: There is no column selection after the dataframe is imported. (column-selection-
pandas)
llexnlp/lexnlp/utils/tests/test_parse_df.py:25:12: R5503: Datatype is not set when a dataframe is imported from data. (datatype-pandas)
************* Module lexnlp.nlp.train.en.train_section_segmanizer
llexnlp/lexnlp/nlp/train/en/train_section_segmanizer.py:16:0: C5501: The import of pandas doesn't bound to 'pd'. (import-pandas)
************* Module lexnlp.nlp.en.segments.sections
llexnlp/lexnlp/nlp/en/segments/sections.py:24:0: C5501: The import of pandas doesn't bound to 'pd'. (import-pandas)
************* Module lexnlp.nlp.en.segments.paragraphs
llexnlp/lexnlp/nlp/en/segments/paragraphs.py:25:0: C5501: The import of pandas doesn't bound to 'pd'. (import-pandas)
************* Module lexnlp.nlp.en.segments.titles
llexnlp/lexnlp/nlp/en/segments/titles.py:20:0: C5502: The import of numpy doesn't bound to 'np'. (import-numpy)
llexnlp/lexnlp/nlp/en/segments/titles.py:21:0: C5501: The import of pandas doesn't bound to 'pd'. (import-pandas)
llexnlp/lexnlp/nlp/en/segments/titles.py:161:4: R5504: There is no column selection after the dataframe is imported. (column-selection-pa
ndas)
llexnlp/lexnlp/nlp/en/segments/titles.py:161:20: R5503: Datatype is not set when a dataframe is imported from data. (datatype-pandas)
************* Module lexnlp.nlp.en.segments.pages
llexnlp/lexnlp/nlp/en/segments/pages.py:24:0: C5501: The import of pandas doesn't bound to 'pd'. (import-pandas)
************* Module lexnlp.extract.common.universal_court_parser
```

Figure 3.1: Dslinter

We developed a static analysis tool – coined as `dslinter` – that implements 20 code smells, based on our study presented in Chapter 2. The tool is developed under an open source license and is available here: `https://github.com/SERG-Delft/dslinter`. It can also be easily installed using the pip package manager: `https://pypi.org/project/dslinter/`. There are 36 checkers in total for different libraries. In 36 checkers, 9, 3, 9, 8, and 5 examine the API usage for Pandas, NumPy, PyTorch, TensorFlow, and Scikit-learn libraries, respectively. Each checker's name is a combination of smell and library. Below, we pinpoint each code smell checker supported by `dslinter`.

3.3.1 Checkers

- **Import Checkers** (`import-pandas`, `import-numpy`, `import-pyplot`, `import-sklearn`, `import-tensorflow`, `import-pytorch`): Check whether data science modules are imported using the correct naming conventions.

- **Unnecessary Iteration Checkers** (`unnecessary-iteration-pandas`, `dataframe-iteration-modification-pandas`, `unnecessary-iteration-tensorflow`): Vectorized solutions are preferred over iterators. These checkers check whether iterations are used while vectorized APIs can be used. If this is the case, the rule is violated.

- **Nan Equality Checker** (`nan-numpy`): The NaN equivalence comparison is different to None comparison. The result of `NaN == NaN` is False. This checker checks whether a value is compared with np.nan. If so, the rule is violated.

- **Chain Indexing Checker** (`chain-indexing-pandas`): Chain indexing is considered bad practice in Pandas code and should be avoided. If chain indexing is used on a Pandas DataFrame, the rule is violated.

- **DataType Checker** (`datatype-pandas`): DataType should be set when a DataFrame is imported from data to ensure the data formats are imported as expected. If the DataType is not set when importing, the rule is violated.

- **Column Selection Checker** (`column-selection-pandas`): Column should be selected after the DataFrame is imported for better elaborating what to be expected in the downstream.

- **Merge Parameter Checker** (`merge-parameter-pandas`): Parameters ‘how’, ‘on’ and ‘validate’ should be set for merge operations to ensure the correct usage of merging.

- **In-Place Checker** (`inplace-pandas`): Operations on DataFrames return new DataFrames, and they should be assigned to a variable. Otherwise, the result is lost, and the rule is violated.

- **DataFrame Conversion Checker** (`dataframe-conversion-pandas`): For DataFrame conversion in Pandas code, use `.to_numpy()` instead of `.values` for better consistency.

- **Scaler Missing Checker** (`scaler-missing-scikitlearn`): Check whether the scaler is used before every scaling-sensitive operation in scikit-learn codes. Scaling-sensitive operations include Principal Component Analysis (PCA), Support Vector Machine (SVM), Stochastic Gradient Descent (SGD), Multi-layer Perceptron classifier and L1 and L2 regularization.

- **Hyperparameter Checkers** (`hyperparameters-scikitlearn`, `hyperparameters-tensorflow`, `hyperparameters-pytorch`): For machine learning algorithms, some important hyperparameters should be set for better reproducibility and less error-prone code.

- **Memory Release Checker** (`memory-release-tensorflow`): The checker checks whether the memory is released in time in the training process. If a neural network is created in the loop, and no memory clear operation is used, the rule is violated.
- **Deterministic Algorithm Usage Checker** (`deterministic-pytorch`): If `use_deterministic_algorithm` is not used in a PyTorch program, the rule is violated.
- **Randomness Control Checkers** (`randomness-control-numpy`, `randomness-control-scikitlearn`, `randomness-control-tensorflow`, `randomness-control-pytorch`, `randomness-control-dataloader-pytorch`): The checkers check whether a random seed is set in the machine learning program to preserve reproducibility. If not, the rule is violated.
- **Mask Missing Checkers** (`missing-mask-tensorflow`, `missing-mask-pytorch`): If log function is used in the code, check whether the argument value is valid, i.e., not equal to zero.
- **Tensor Array Checker** (`tensor-array-tensorflow`): The checker checks whether `tf.TensorArray()` is used for growing array in the loop in TensorFlow program. If not, the rule is violated.
- **Net Forward Checker** (`forward-pytorch`): It is recommended to use `self.net()` rather than `self.net.forward()` in PyTorch code. If `self.net.forward()` is used in the code, the rule is violated.
- **Gradient Clear Checker** (`gradient-clear-pytorch`): The `loss_fn.backward()` and `optimizer.step()` should be used together with `optimizer.zero_grad()` in PyTorch code. If the `optimizer.zero_grad()` is missing in the code, the rule is violated.
- **Pipeline Not Used Checker** (`pipeline-not-used-scikitlearn`): Scikit-learn estimators and preprocessor should be used together inside Pipelines to prevent data leakage.
- **Dependent Threshold Checkers** (`dependent-threshold-scikitlearn`, `dependent-threshold-tensorflow`, `dependent-threshold-pytorch`): If threshold-dependent evaluation(e.g., F-Score) is used in the code, check whether threshold-indenpendent evaluation(e.g., AUC) metrics is also used in the code. If not, the rule is violated.

3.3.2 Implementation

The `dslinter` is a Pylint plugin, which works by traversing the abstract syntax tree (AST) and checking if the rules are violated. The Pylint and its dependencies take care of building the AST and then the `dslinter` checks the rules on the node it visits. Whenever the `dslinter` finds a violation, it exports the violation to the output. Other Pylint features can still be enabled when using this plugin.

Example 1 – Chain Indexing Checker

Listing 3.1: Checker Example 1

```
import pandas as pd
df = pd.DataFrame([[1,2,3],[4,5,6]])
col = 1
x = 0
- df[col][x] = 42
+ df.loc[x, col] = 42
```

The Chain Indexing Checker shown in listing 3.1 works by checking all the subscript nodes and counting the indexing number attached to the subscript node. If the subscript node is a DataFrame and the indexing number is no less than two, the rule is violated, and a message is raised. Whether the subscript node is a DataFrame is checked by a type inference module. The type inference module makes use of the `mypy` package to statically infer the type of the object in the Python code.

Example 2 – Scaler Missing Checker

Listing 3.2: Checker Example 2

```
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC
+ from sklearn.pipeline import make_pipeline
+ from sklearn.preprocessing import StandardScaler

# Make a train/test split using 30% test size
RANDOM_STATE = 42
features, target = load_wine(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    features, target, test_size=0.30, random_state=RANDOM_STATE
)

# Fit to data and predict using pipelined GNB and PCA
- clf = SVC()
+ clf = make_pipeline(StandardScaler(), SVC())
clf.fit(X_train, y_train)
pred_test = clf.predict(X_test)
ac = accuracy_score(y_test, pred_test)
```

The Scaler Missing Checker shown in listing 3.2 checks whether a scaler is used before the scaling-sensitive operations. The scaler in the Scikit-learn library are `"PCA"`, `"KernelPCA"`, `"SparsePCA"`, `"IncrementalPCA"`, `"LinearSVC"`, `"LinearSVR"`, `"NuSVC"`, `"NuSVR"`, `"OneClassSVM"`, `"SVC"`, `"SVR"`, `"SGDClassifier"`, `"SGDOneClassSVM"`, `"SGDRegressor"`, `"MLPClassifier"` and `"MLPRegressor"`. The scaling-sensitive operations are `"RobustScaler"`, `"StandardScaler"`, `"MaxAbsScaler"` and `"MinMaxScaler"`. The learning functions are `"fit"`, `"fit_transform"` and `"transform"`.

Since the scaling-sensitive operations can be used with or without Pipeline API in the Scikit-learn library, the violation detection has two parts of checks, respectively. If the Pipeline API is used in the code, and there is a scaling-sensitive function used as an argument of the Pipeline call, check whether a scaler is also in the arguments. If not, the rule is violated and a warning message is raised. If the Pipeline API is not used in the code, and a learning function is used on a variable representing a scaling-sensitive operation, check the argument used in the learning function. If the argument has never been processed by a scaler, the rule is violated and a warning message is raised.

3.3.3 Improvement Iterations

During development, to test the code smell checkers, we ran the `dslinter` against an existing dataset of Python notebooks and another dataset of regular Python projects. We then collected 20 code instances detected by our tool: 10 from notebooks and another 10 from regular projects. These 20 smells are manually analysed to detect whether there are any false positives and improvements that need to be made in the tool. These datasets, Notebook Dataset 1 and Project Dataset 1, are further explained below in Section 3.4.1.

3.4 Methodology

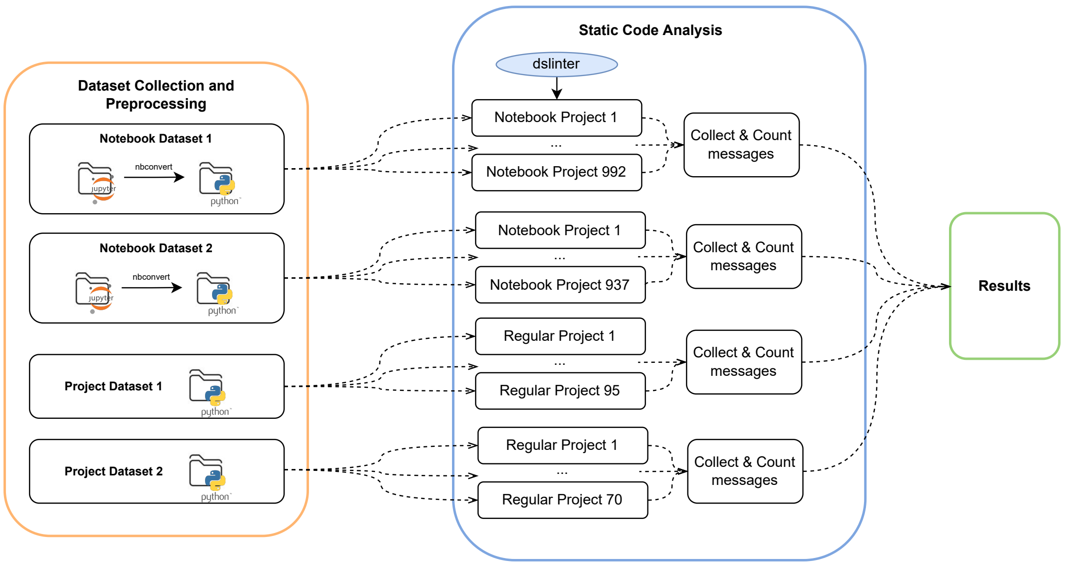


Figure 3.2: The Whole Process

This section describes the methodology for conducting the empirical study on the prevalence of machine learning-specific code smells. The whole process contains dataset collection and preprocessing as well as static code analysis, which is outlined in Figure 3.2.

3.4.1 Dataset Collection and Preprocessing

We use four datasets in our study, including two notebook sets and two regular project sets. These two types of datasets are different by nature. The notebooks typically have small code blocks and a single file, while the regular projects always have more files and complex entanglement. We analyse these two groups separately because we anticipate that these differences might cause an impact on the overall quality of the code, which can be reflected by a different distribution of code smell instances.

Table 3.1: Characteristics of the Datasets

	Notebook Dataset 1	Notebook Dataset 2	Project Dataset 1	Project Dataset 2
Number of projects	992	927	95	70
Number of Python files	992	927	5,075	2,866
Lines of Python code	552,488	340,717	849,014	466,924
Average code line number for each project	557	368	8,937	6,670
Average code line number for each Python file	557	368	167	163

- **Notebook Dataset 1**: A list of 1100 machine learning kernels was collected by Haakman et al. [7] from Kaggle on May 1st, 2020. The kernel refers to a solution in Kaggle, which is a combination of code, data and analysis. Typically there is a notebook in a kernel. The kernels in the list are selected based on 'hotness', meaning that they are rated highly and have been consistently popular on the platform for a long time. They are collected using the Kaggle API[6] and the notebooks are converted to Python scripts using `nbconvert` [7]. We wrapped each kernel in a folder. In the list, 93 kernels are not able to access on May 8th, 2022 and 15 kernels do not contain notebooks or Python scripts. Instead, they contain R scripts or other kinds of files. We delete the folders without notebooks or Python files and call the rest of the folders notebook projects. 992 notebook projects are retrieved at the end. This set has 552,488 source code lines of Python code. Each notebook project has 557 source code lines on average.
- **Notebook Dataset 2**: We collected a list of 1000 kernels on May 8th, 2022. These kernels are also from Kaggle and ranked by 'hotness', and Notebook Dataset 2 does not overlap with Notebook Dataset1. Since 3 kernels cannot be accessed and 58 kernels do not contain notebooks or Python scripts, 939 notebook projects are retrieved from this list. During the static code analysis process, the analysis tool mentioned in section 3.4.2 failed to parse the output for 12 projects. Therefore, 927 notebook projects are retrieved and analysed at the end. This set has source code lines of 340,717 Python code. Each script has 368 source code lines on average.

[6]Kaggle API: `https://github.com/Kaggle/kaggle-api`
[7]nbconvert: `https://github.com/jupyter/nbconvert`

- **Project Dataset 1**: We reused the first 100 projects in the Boa Data science corpus created by Biswas et al. [2] as the Project Dataset 1. The corpus was created to enable mining software repository research, such as identifying bug patterns or suggesting best practices for developing Data Science applications, which matches our study. The projects in the corpus are all written in Python and developed for Data Science tasks. Two projects cannot be retrieved, and the analysis tool fails to parse the output for three projects, so there are 95 projects left. This set contains 849,014 source code lines, so each project has 8,947 code lines on average.

- **Project Dataset 2**: Oort et al. [29] collected a set of 74 ML projects and analyzed them with Pylint to research the prevalence of code smells in ML projects. The 74 projects come from academic papers, (student) reproductions, Kaggle competitions, and industrial projects, which are also suitable for our research. Therefore, we reuse the dataset as the Project Dataset 2. The analysis tool fails to parse the output for four projects, so there are 70 projects left. This set contains 466,974 source code lines in total, with 6,670 source code lines for each project on average.

In total, we gathered 2,084 projects, 9,860 Python files, and 2,209,143 lines of Python code. The average code line number for each project is 1060, and the average code line number for each Python file is 224.

Table 3.2: Top 5 Popular Libraries & The Libraries in Dslinter

Top 5 Popular Libraries		The Libraries in Dslinter	
Library	Number	Library	Number
NumPy	1777	NumPy	1777
Pandas	1751	Pandas	1751
Matplotlib	1394	Scikit-learn	1146
Scikit-learn	1146	TensorFlow	291
Seaborn	930	PyTorch	157

To analyze whether the scope of the `dslinter` fits the projects collected in our datasets, we did a preliminary assessment of the top libraries used by the projects in the dataset. We then compare these top libraries with the libraries supported by `dslinter`. The top five data science libraries used in the datasets are NumPy, Pandas, Matplotlib, Scikit-learn, and Seaborn. The five libraries `dslinter` focus on are NumPy, Pandas, Scikit-learn, TensorFlow and PyTorch. So we cover all the most popular libraries except for the visualization libraries – i.e., matplotlib and seaborn, which are beyond our study scope.

3.4.2 Static Code Analysis

We run `dslinter 2.0.9` on the four datasets mentioned above to verify the effectiveness and accuracy of the checkers. "Run dslinter" means running Pylint with the `dslinter` plugin enabled and other checkers disabled. For randomness control checkers, we use different settings for notebooks and regular projects. Since the random seed only needs to be set once and globally in the program, the checkers, by default, only check whether the entrance file of the project has the random seed set to avoid excessive false positives. The checkers check whether the file is an entrance file by checking whether `"if __name__ == '__main__:'"` is used. For notebooks, one file usually stands for one project, while they normally do not use `"if __name__ == '__main__:'"` in the script. Therefore, we disable the entrance file check in the randomness control checkers for notebooks by using `no_main_module_check` option.

We reuse and adapt the replication package from [29] to run `dslinter`. Previous study [9] simply treated all notebooks as a big project and used the Pylint command line in the console to perform the evaluation. We did not follow the method in [9]. Instead, we use the analysis tool from [29] to conduct the experiment. This is done for two reasons: 1) The projects can be distributedly run to save time. 2) The report generated by the analysis tool shows not only the number of violations for each checker but also the number for each notebook or regular project. In this way, we can gain more insight into the code smell difference between various notebooks or regular projects. To adapt the tool to our tasks, we made some modifications to the code: 1) We replaced the Pylint command with the `dslinter` command. 2) The dependency installation part of the code is removed because collecting the result from the static analysis tool does not need the dependency to be installed, while this is the most time-consuming part and causes some exceptions. 3) The tool originally required a URL of the projects to download the code. However, the notebooks do not belong to any git repository and thus do not have URLs. We modified the code to enable it to run from folders, solving the problem of running the tool on notebooks. The code for our experiment can be found at `https://gitlab.com/Hynn01/python-ml-analysis`, and the code for extra analysis and the visualization of the results can be found at `https://github.com/Hynn01/dslinter-experiments`. In total, our experiments on Notebook Dataset 1, Notebook Dataset 2, Project Dataset 1 and Project Dataset 2 took 65, 32, 19, and 10 minutes respectively.

We collect the report generated from the analysis tool and analyse the results. The general results can be used to answer RQ1. To address RQ2, we manually inspect up to 10 instances respectively in notebook dataset 2 and project dataset 2 and then combined the result to calculate a true positive rate. Applying the true positive rate, we estimate the number of true positives of each checker. Moreover, the result for regular projects and notebooks are compared to answer RQ3. Combining the insights taken from the results, we deduce patterns to answer the research questions and raise discussions.

3.5 Results

Applying our methodology, we run the `dslinter` on four datasets and gather the results. In this section, we present our results and answer the research questions.

3.5.1 RQ1: How prevalent are the machine learning-specific code smells?

The checkers in the `dslinter` found 14,854 violations in total in all datasets. In Table 3.3, we show the number of violations for each checker on Notebook Dataset 1, Notebook Dataset 2, Project Dataset 1, Project Dataset 2, respectively. Following the total number of violations for each checker, a bar is plotted accordingly to visualize the number. The checkers in the table follow the order of code smells we showed in the catalog in Chapter 2.

Table 3.3: The Number of Violations for Each Checker

Checker Name	Notebook Dataset 1	Notebook Dataset 2	Project Dataset 1	Project Dataset 2	Total
import-pandas	4	0	32	0	36
import-numpy	7	6	256	19	288
import-pyplot	2	0	1	0	3
import-sklearn	9	19	1	3	32
import-tensorflow	0	4	4	0	8
import-pytorch	2	0	0	3	5
unnecessary-iteration-pandas	8	15	3	0	26
dataframe-iteration-modification-pandas	0	0	0	0	0
unnecessary-iteration-tensorflow	0	0	0	0	0
nan-numpy	7	0	0	0	7
chain-indexing-pandas	73	22	0	8	103
datatype-pandas	2169	1586	47	766	4568
column-selection-pandas	1907	1354	23	463	3747
merge-parameter-pandas	6	3	1	0	10
inplace-pandas	21	20	0	0	41
dataframe-conversion-pandas	351	191	14	214	770
scaler-missing-scikitlearn	99	66	4	24	193
hyperparameters-scikitlearn	392	420	9	11	832
hyperparameters-tensorflow	19	16	1	0	36
hyperparameters-pytorch	4	7	6	12	29
memory-release-tensorflow	0	0	0	0	0
deterministic-pytorch	51	66	15	229	361
randomness-control-numpy	523	499	248	169	1439
randomness-control-scikitlearn	106	91	6	8	211
randomness-control-tensorflow	83	100	443	42	668
randomness-control-pytorch	32	51	12	207	302
randomness-control-dataloader-pytorch	86	83	25	428	622
missing-mask-tensorflow	1	0	84	7	92
missing-mask-pytorch	3	0	5	49	57
tensor-array-tensorflow	0	0	0	0	0
forward-pytorch	3	8	16	33	60
gradient-clear-pytorch	0	3	1	0	4
pipeline-not-used-scikitlearn	159	135	0	0	294
dependent-threshold-scikitlearn	7	3	0	0	10
dependent-threshold-tensorflow	0	0	0	0	0
dependent-threshold-pytorch	0	0	0	0	0

As shown in Table 3.3 and Figure 3.3, the two most frequently violated rules are `datatype-pandas` and `column-selection-pandas`, which are related to the `Columns and DataType Not Explicitly Set` smell. They have 4,568 and 3,747 occurrences, respectively. The number of violations for them is much higher than for other rules.

Five of the top ten most violated rules correspond to `Randomness Uncontrolled` smell. The `randomness-control-numpy`, `randomness-control-tensorflow`, `randomness-control-dataloader-pytorch`, `deterministic-pytorch` and `randomness-control-pytorch` rules have 1439, 668, 662, 302, 361 violations, respectively. It demonstrates that the random seed is commonly not specified in machine learning projects.

The `hyperparamters-scikitlearn` rule has the fourth highest number of violations with 832 violations. This rule is related to the `Hyperparameter not Explicitly Set` smell.

The `data-conversion-pandas` rule has the fifth highest number of violations, which has 770 violations. This rule is derived from the official Pandas documentation and relates to the `DataFrame Conversion API Misused` smell.

Finally, the `pipeline-not-used-scikitlearn` has the tenth highest number of violations, amounting to 294. This rule relates to the `Data Leakage` smell. The documentation for the Scikit-learn library recommends using the Pipeline API to prevent inconsistent preprocessing and data leakage. However, 294 code instances do not follow this practice in the codebases.

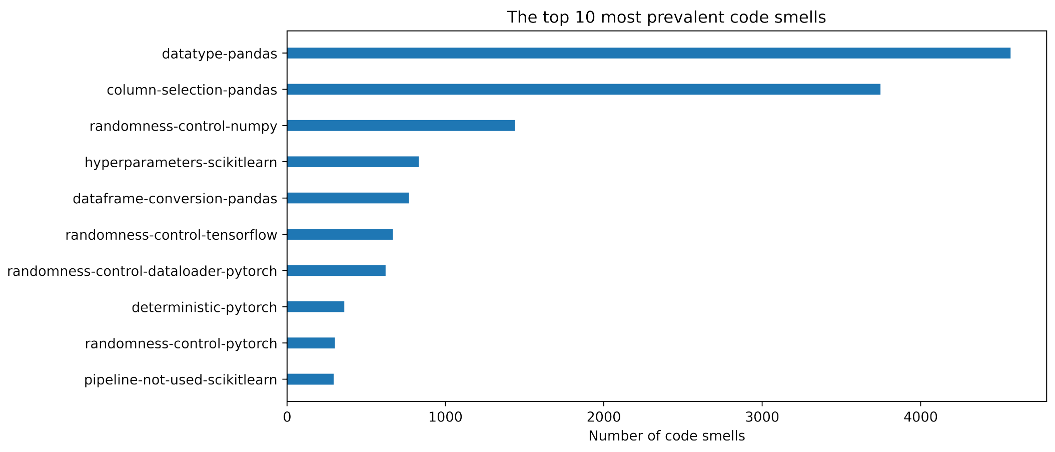


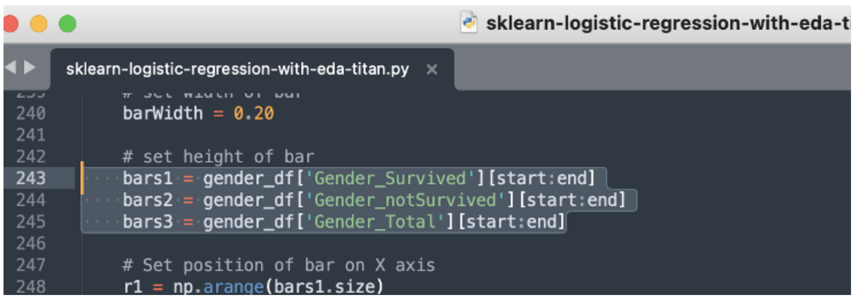
Figure 3.3: The Top 10 Most Frequently Violated Rules

3.5.2 RQ2: How accurate is the machine learning-specific code smell detection tool dslinter?

An overview of the true positives rate of the tool is provided in Table 3.4. It shows the checker name, the number of instances in Notebook Dataset 2 and Project Dataset 2, and the estimated true positives calculated based on the true positives rate. The true positive rates are calculated based on manually inspecting 20 randomly selected instances: 10 from Notebook Dataset 2 and 10 from Project Dataset 2. If the instance number is fewer than 10 in the Notebook Dataset 2 or Project Dataset 2, we inspect all the instances in that dataset. A true positive is a code instance with a refactoring opportunity or a potential bug.

Out of 36 checkers, 24 checkers have the 100% true positives rate. The `gradient-clear-pytorch` checker has the lowest true positive rate – 0%. We randomly select three examples of the true positives to show in Figures 3.4, 3.5, 3.6 and three examples of false positives in Figures 3.7, 3.8, 3.9.

Examples of True Positives


sklearn-logistic-regression-with-eda-t
sklearn-logistic-regression-with-eda-titan.py

```
    barWidth = 0.20

    # set height of bar
    bars1 = gender_df['Gender_Survived'][start:end]
    bars2 = gender_df['Gender_notSurvived'][start:end]
    bars3 = gender_df['Gender_Total'][start:end]

    # Set position of bar on X axis
    r1 = np.arange(bars1.size)
```

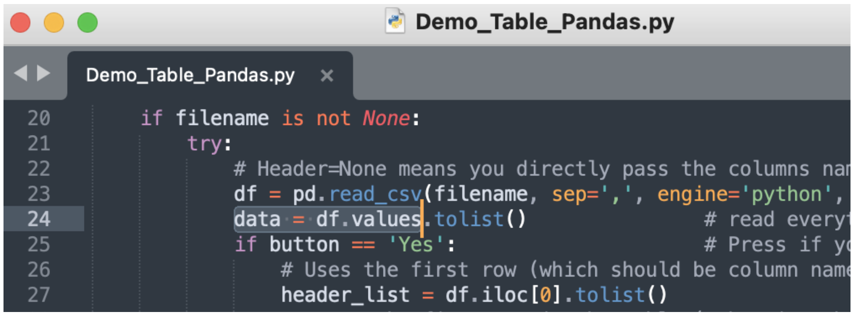
Figure 3.4: True Positive 1 - Chain Indexing Smell

Table 3.4: True Positive Number of Each Checker

Checker Name	Notebook Dataset 2 + Project Dataset 2	Estimated True Positives	TP Rate
import-pandas	0	0	100%
import-numpy	25	25	100%
import-pyplot	0	0	100%
import-sklearn	22	22	100%
import-tensorflow	4	4	100%
import-pytorch	3	3	100%
unnecessary-iteration-pandas	15	9	60%
dataframe-iteration-modification-pandas	0	0	100%
unnecessary-iteration-tensorflow	0	0	100%
nan-numpy	0	0	100%
chain-indexing-pandas	30	30	100%
datatype-pandas	2352	2352	100%
column-selection-pandas	1817	1817	100%
merge-parameter-pandas	3	3	100%
inplace-pandas	20	8	40%
dataframe-conversion-pandas	405	385	95%
scaler-missing-scikitlearn	90	86	95%
hyperparameters-scikitlearn	431	431	100%
hyperparameters-tensorflow	16	10	60%
hyperparameters-pytorch	19	10	53%
memory-release-tensorflow	0	0	100%
deterministic-pytorch	295	280	95%
randomness-control-numpy	668	568	85%
randomness-control-scikitlearn	99	99	100%
randomness-control-tensorflow	142	121	85%
randomness-control-pytorch	258	181	70%
randomness-control-dataloader-pytorch	511	511	100%
missing-mask-tensorflow	7	6	86%
missing-mask-pytorch	49	49	100%
tensor-array-tensorflow	0	0	100%
forward-pytorch	41	33	80%
gradient-clear-pytorch	3	0	0%
pipeline-not-used-scikitlearn	135	135	100%
dependent-threshold-scikitlearn	3	3	100%
dependent-threshold-tensorflow	0	0	100%
dependent-threshold-pytorch	0	0	100%

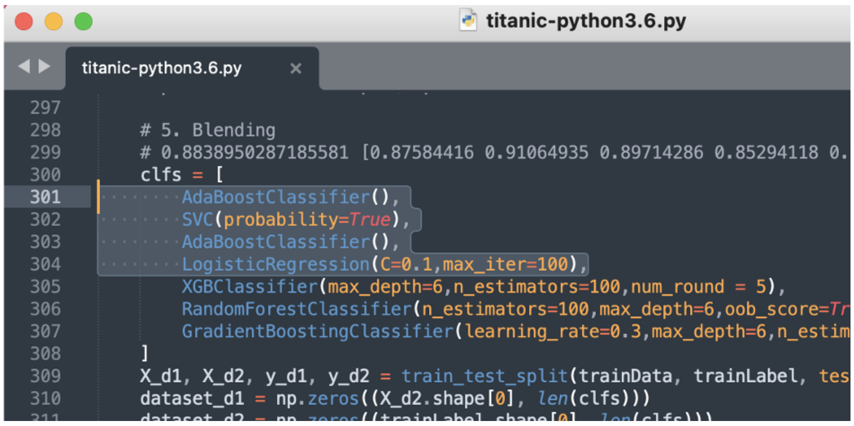
Three true positives of the `Chain Indexing` smell detected by the `chain-indexing-pandas` checker are shown in Figure 3.4. In the example, the code `gender_df['Gender_Survived'][start:end]` can be refactored to `gender_df.loc('Gender_Survived', start:end)`, which is more efficient and less error-prone.

A true positive of the `DataFrame Conversion API Misused` smell detected by the `dataframe-conversion-pandas` checker is presented in Figure 3.5. As recommended in the Pandas official documentation, the code `df.values` can be refactored to `df.to_numpy()`. Since the returned value of `.values` is not always consistent, (i.e., it could be the actual array, some transformation of it, or one of the Pandas custom arrays) and the semantic



```
    if filename is not None:
        try:
            # Header=None means you directly pass the columns nar
            df = pd.read_csv(filename, sep=',', engine='python',
            data = df.values.tolist()                  # read every
            if button == 'Yes':                        # Press if yo
                # Uses the first row (which should be column name
                header_list = df.iloc[0].tolist()
```

Figure 3.5: True Positive 2 - DataFrame Conversion API Misused Smell



```

    # 5. Blending
    # 0.8838950287185581 [0.87584416 0.91064935 0.89714286 0.85294118 0.
    clfs = [
        AdaBoostClassifier(),
        SVC(probability=True),
        AdaBoostClassifier(),
        LogisticRegression(C=0.1,max_iter=100),
        XGBClassifier(max_depth=6,n_estimators=100,num_round = 5),
        RandomForestClassifier(n_estimators=100,max_depth=6,oob_score=Tr
        GradientBoostingClassifier(learning_rate=0.3,max_depth=6,n_estim
    ]
    X_d1, X_d2, y_d1, y_d2 = train_test_split(trainData, trainLabel, tes
    dataset_d1 = np.zeros((X_d2.shape[0], len(clfs)))
    dataset_d2 = np.zeros((trainLabel.shape[0], len(clfs)))
```

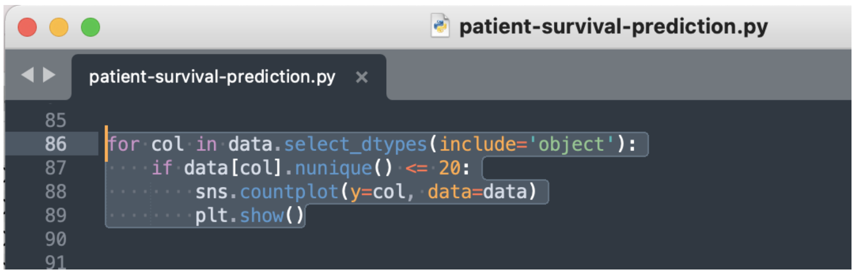
Figure 3.6: True Positive 3 - Hyperparameter Not Explicitly Set Smell

of `.values` is ambiguous, using `.to_numpy()` would guarantee an array and clarify the code.

A true positive of the `Hyperparameter Not Explicitly Set` smell detected by the `hyperparameters-scikitlearn` checker is displayed in Figure 3.6. The learning rate is an important hyperparameter for AdaBoostClassifier [20]. Nevertheless, it is not set in the code. This might harm the reproducibility and maintainability of the software solution. We recommend to refactor the code `AdaBoostClassifier()` to `AdaBoostClassifier(batch_size=xxx)`.

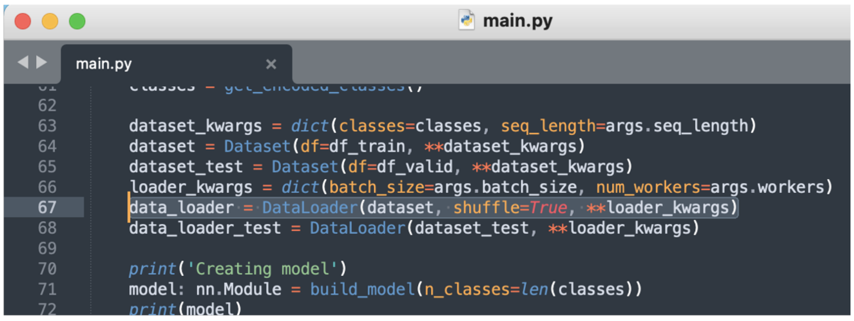
Examples of False Positives

A false positive of the `Unnecessary Iteration` smell detected by the `unnecessary-iteration-pandas` checker is depicted in Figure 3.7. There is indeed an iteration over a DataFrame in the code. However, since the operation inside the iteration is plotting, the code cannot be refactored. A possible solution for avoiding this kind of false positive is to check whether a possible-to-vectorize operation is used in the iteration. This might require extra manual effort to collect all the possible-to-vectorize functions. Adding this constraint


patient-survival-prediction.py
patient-survival-prediction.py

```python
for col in data.select_dtypes(include='object'):
    if data[col].nunique() <= 20:
        sns.countplot(y=col, data=data)
        plt.show()
```

Figure 3.7: False Positive 1 - Unnecessary Iteration Smell



```python

    dataset_kwargs = dict(classes=classes, seq_length=args.seq_length)
    dataset = Dataset(df=df_train, **dataset_kwargs)
    dataset_test = Dataset(df=df_valid, **dataset_kwargs)
    loader_kwargs = dict(batch_size=args.batch_size, num_workers=args.workers)
    data_loader = DataLoader(dataset, shuffle=True, **loader_kwargs)
    data_loader_test = DataLoader(dataset_test, **loader_kwargs)

    print('Creating model')
    model: nn.Module = build_model(n_classes=len(classes))
    print(model)
```

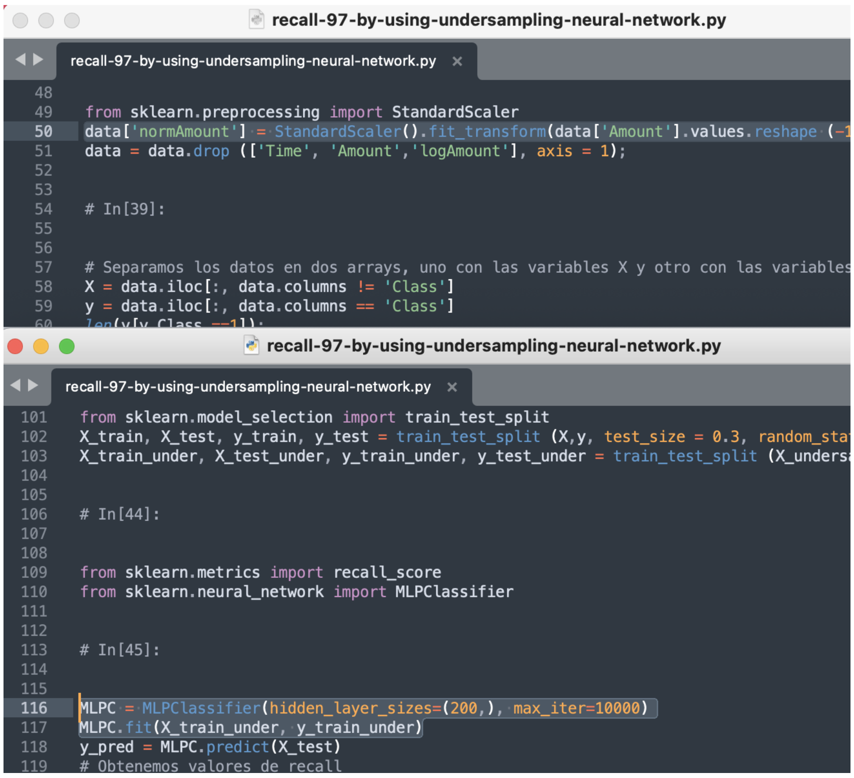
Figure 3.8: False Positive 2 - Hyperparameter Not Explicitly Set Smell

will reduce false positives, while at the same time, it might increase the false negatives if the possible-to-vectorize function list cannot cover all the functions.

A false positive of the `Hyperparameter Not Explicitly Set` smell detected by the `hyperparameters-pytorch` checker is exhibited in Figure 3.8. In this code example, the Dataloader takes an argument `**loader_kwargs`, which predefines the `batch_size` before. Nevertheless, the checker did not take this into account and caused a false positive. There are two possible ways to reduce this false positive: 1) Whenever the checker sees an argument list (e.g., `**kwargs`), it assumes that the hyperparameters are already predefined and skip the check. This might cause some false negatives. 2) Trace the argument list variable and see what is assigned to the variable. This increases the complexity of the checker and elongates the time required to check the violation.

A false positive for the `No Scaling before Scale-Sensitive Operation` smell detected by the `scaler-missing-scikitlearn` checker is presented in Figure 3.9. The data was initially scaled but underwent numerous other data processing steps and was assigned to another variable. In the end, the checker believed that the scaling-sensitive function's input data had never been processed by a scaler, resulting in a false positive.

‘


recall-97-by-using-undersampling-neural-network.py
recall-97-by-using-undersampling-neural-network.py
recall-97-by-using-undersampling-neural-network.py
recall-97-by-using-undersampling-neural-network.py

```

from sklearn.preprocessing import StandardScaler
data['normAmount'] = StandardScaler().fit_transform(data['Amount'].values.reshape (-1
data = data.drop (['Time', 'Amount','logAmount'], axis = 1);


# In[39]:


# Separamos los datos en dos arrays, uno con las variables X y otro con las variables
X = data.iloc[:, data.columns != 'Class']
y = data.iloc[:, data.columns == 'Class']
len(y[y.Class ==1]);
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split (X,y, test_size = 0.3, random_sta
X_train_under, X_test_under, y_train_under, y_test_under = train_test_split (X_unders


# In[44]:


from sklearn.metrics import recall_score
from sklearn.neural_network import MLPClassifier


# In[45]:


MLPC = MLPClassifier(hidden_layer_sizes=(200,), max_iter=10000)
MLPC.fit(X_train_under, y_train_under)
y_pred = MLPC.predict(X_test)
# Obtenemos valores de recall
```

Figure 3.9: False Positive 3 - No Scaling before Scaling-Sensitive Operation Smell

3.5.3 RQ3: Is the prevalence of code smells different in Python notebooks and regular Python projects?

An overview of the comparison between the notebooks and the projects is presented in Figure 3.10. The bars show the number of rule violations in notebooks and projects per 100,000 lines. As shown in the figure, in most cases, notebooks contain more code smells than projects. However, there are also a few cases where projects have more violations than notebooks, which includes the following rules: `"import-pandas"`, `"import-numpy"`, `"hyperparameters-pytorch"`, `"deterministic-pytorch"`, `"randomness-control-tensorflow"`, `"randomness-control-pytorch"`, `"randomness-control-dataloader-pytorch"`, `"missing-mask-tensorflow"`, `"missing-mask-pytorch"` and `"forward-pytorch"`.

We plotted the top 10 most prevalent code smells in notebooks and regular projects, respectively. As we can see from the figures, six rules are the same and four are different. Six common rules are `datatype-pandas`, `column-selection-pandas`, `randomness-control-numpy`, `dataframe-conversion-pandas`, `randomness-control-tensorflow`, and `randomness-control-dataloader-pytorch`. The rest four code smells in notebooks are `"hyperparameters-scikitlearn"`, `"pipeline-not-used-scikitlearn"`, `"randomness-control-scikitlearn"` and `"scaler-missing-scikitlearn"`. The rest four code smells in projects are `"import-`

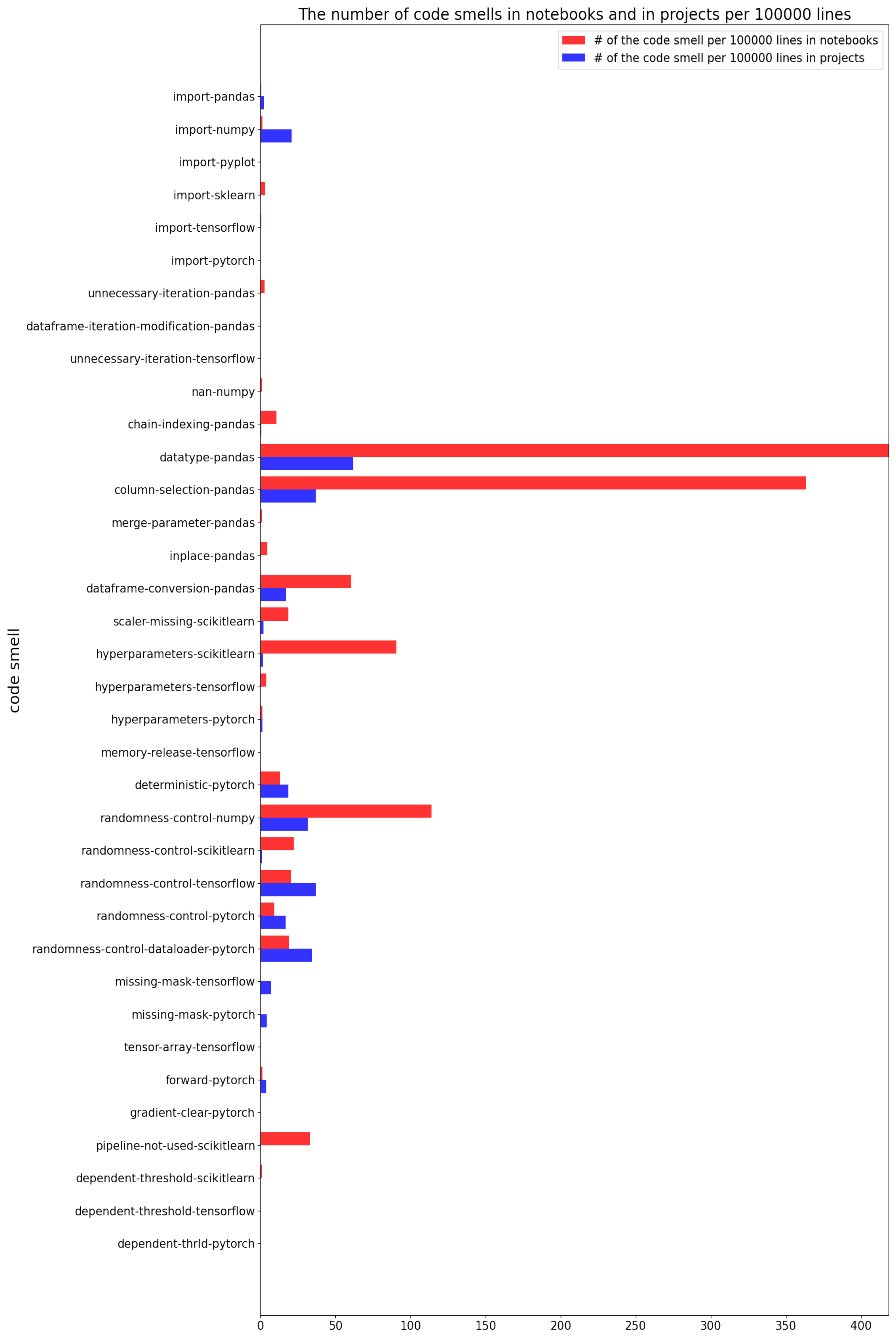


Figure 3.10: The number of code smells in notebooks and in projects per 100000 lines

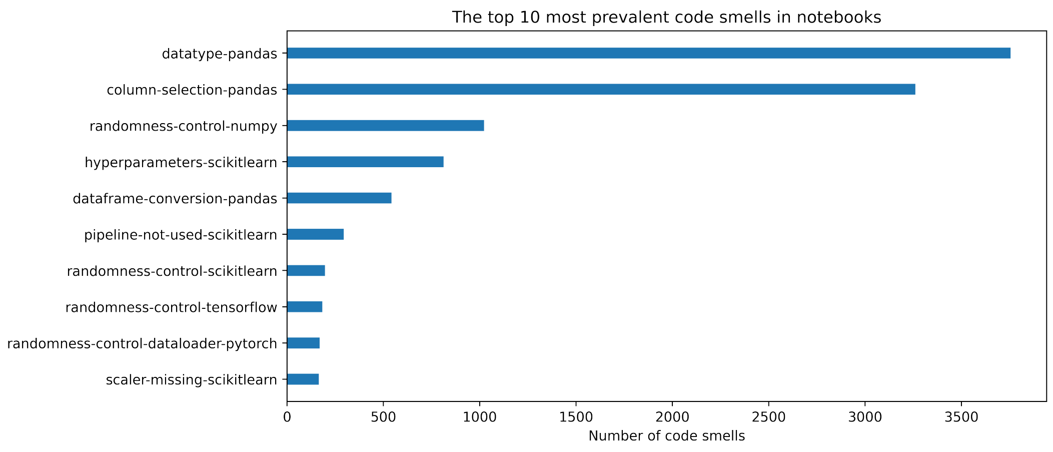


Figure 3.11: The top 10 most prevalent code smells in notebooks

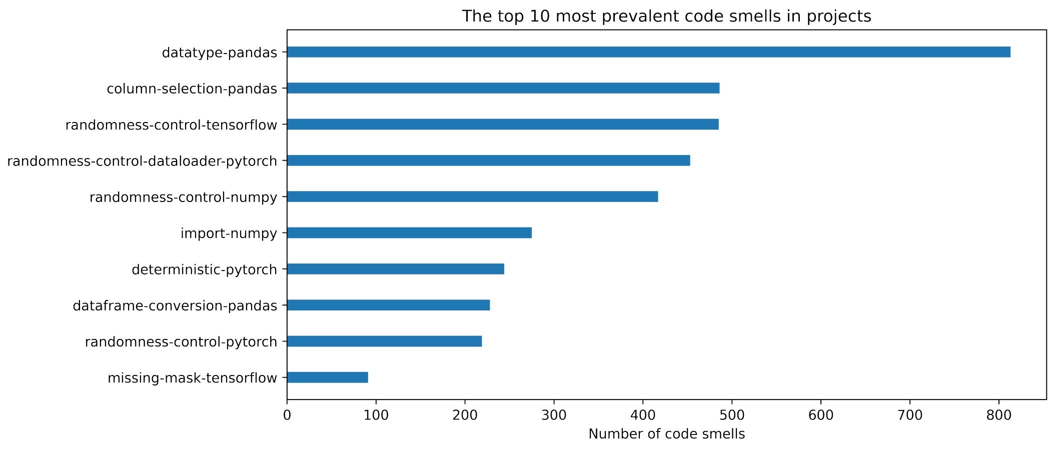


Figure 3.12: The top 10 most prevalent code smells in projects

`numpy"`, `"deterministic-pytorch"`, `"randomness-control-pytorch"` and `"missing-mask-tensorflow"`.

Additionally, we plotted the distribution of code smell counts within a single notebook or project. The majority of notebooks or projects contain fewer than 10 code smells. The maximum number of code smells a notebook can have is approximately 70, and the maximum number of code smells for a project is approximately 700.

Most notebooks contain two distinct types of code smells, whereas most projects do not have any code smells. 19% of notebooks do not contain any code smell, while 28% of projects do not contain any code smell.

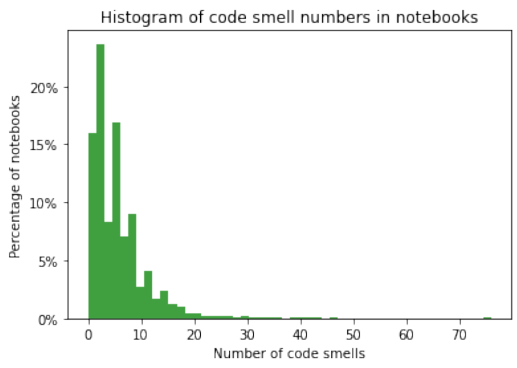


Figure 3.13: Histogram of code smell numbers in notebooks

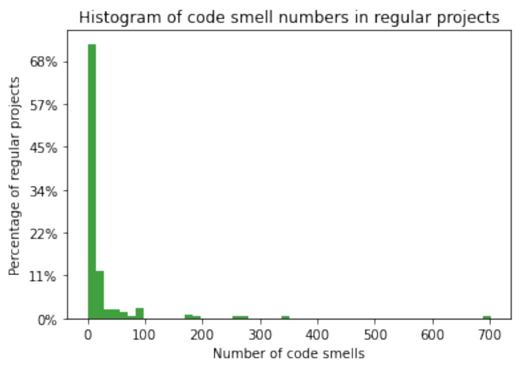


Figure 3.14: Histogram of code smell numbers in projects

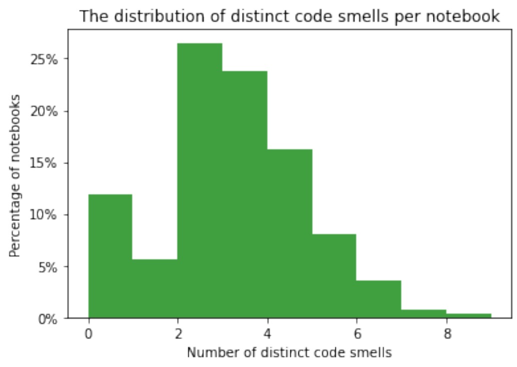


Figure 3.15: The distribution of distinct code smell number per notebook

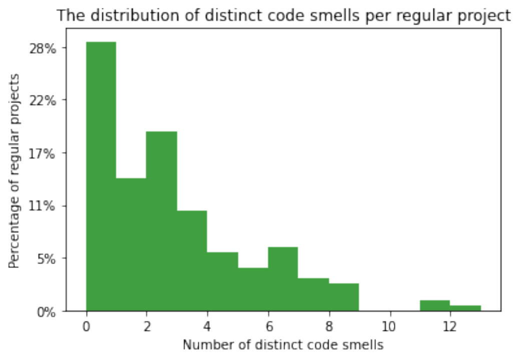


Figure 3.16: The distribution of distinct code smell number per regular project

3.6 Discussions and Implications

In this section, we discussed our results and implications. We discuss the results in three aspects: the prevalence of code smells, the challenges of building and putting the static analysis tool into use, and the code smell prevalence difference between notebooks and regular projects.

3.6.1 Discussions about the prevalence of code smells

As shown in the result for RQ1, the two most frequently violated rules are `datatype-pandas` and `column-selection-pandas`. After manually inspecting a few instances, we discovered some interesting patterns. For instance, the developers typically use `df.head()` to print out the DataFrame, rather than selecting columns to imply what data to expect downstream. Also, developers typically import data without explicitly setting the DataType. However, these smells were mentioned in an unofficial Pandas style guide. This demonstrates that while some developers may perceive this smell as a problem, others do not. It indicates that these two checkers should be disabled by default, but enabled by developers who believe this may cause issues. The configuration of a static analysis tool is crucial.

Five of the top ten most violated rules correspond to `Randomness Uncontrolled` smell, which demonstrates that the random seed is commonly not specified in machine learning projects. One could argue that a portion of the instances are false positives because randomness does not need to be controlled in some scripts. However, we believe that many machine learning algorithms have a random seed involved, and thus it is preferable to set the random seed. The result also indicates that the developers do not care much about reproducibility.

The `hyperparamters-scikitlearn` rule has the fourth highest number of violations. On the one hand, this is partially due to the widespread use of the Scikit-learn library and its learning class. On the other hand, it suggests that some developers are unaware that explicitly setting hyperparameters is the best practice, while some developers and researchers have different opinions about the essential hyperparameters.

The `data-conversion-pandas` rule and the `pipeline-not-used-scikitlearn` rule has the fifth and tenth highest number of violations, respectively. These two rules are specified in the official library documentation. Nevertheless, they still have a large number of violations. This indicates that the developer is unaware of best practices, resulting in the accumulation of technical debt.

- A portion of the rules are derived from unofficial documentation, which a substantial number of code instances do not adhere to.
- Reproducibility of results is not sufficiently valued.
- Many developers do not explicitly set the hyperparameters or hold divergent views regarding the most crucial hyperparameters.
- A significant proportion of violations involve API usage that deviates from the official documentation recommendation, which threatens the reliability of software.

One may argue that these code smells are widespread because the APIs are widely used. Many instances in the codebases follow the best practice as well. Therefore, we conduct a further analysis experiment on all the datasets. We make changes to the dslinter code on the "experiement" branch [8]. For instance, `scaler-missing-scikitlearn` checker checks whether the variable for a scaler-sensitive learning function was not processed by a scaler before. We make a `scaler-missing-scikitlearn-correct` checker by checking whether the variable was used by a scaler before. The new checkers are also run by the analysis tool.

There are 4568 code instances that violate the `datatype-pandas` rule and 108 that adhere to it, whereas there are 3747 instances of code that violate the `column-selection-pandas` rule and 155 that follow it. This result is consistent with the understanding gained from the number of violations and the manual inspection of the violations, indicating that developers do not typically code in this manner. For the top 5 most prevalent smells, the number of instances that break the rules is far more than that follow the best practices.

Among the checker rules that we did further analysis on, only `scaler-missing-scikitlearn` checker has more instances that follow the best practice. It demonstrates that a scaler is typi-

[8]The "experiement" branch: `https://github.com/SERG-Delft/dslinter/tree/experiment`

Table 3.5: The number of code instances that break the rule and follow the rule in four datasets

Four Datasets	Number of instances that break the rule	Number of instances that follow the rule
datatype-pandas	4568	108
column-selection-pandas	3747	155
randomness-control-numpy	1439	270
hyperparameters-scikitlearn	832	84
dataframe-conversion-pandas	770	108
scaler-missing-scikitlearn	193	203

cally placed before a scaling-sensitive operation. If the developer does not comply, it would be reasonable to issue a warning.

A scaler is typically placed before a scaling-sensitive operation. It would be reasonable to issue a warning if the developer does not comply with the rule.

Another interesting finding by looking at the raw data is that some projects have specific styles and have a high number of violations on a specific rule. For instance, in the `chemistry chainer` in Project Dataset 1, there are 164 violations of the `import-numpy` rule. While most of the developers use `import numpy as np`, the developers in this project use `import numpy` all the time and skip the alias. For the project that has their specific style, we agree that the developers can customize the configuration to meet their tastes. On the other hand, if the developer is unaware of the common practices before, our tool can help them to have early detection and prevent the technical debt from accumulating. We believe that following the common practice will make it easier for outside contributors to participate in public projects.

For the project that has their specific style, we agree that the developers can customize the configuration to meet their tastes. On the other hand, if the developer is unaware of the common practices before, our tool can help them to have early detection and prevent the technical debt from accumulating.

3.6.2 Discussions about the challenges of building and applying static analysis tool

First of all, the results of RQ1 indicate that the rules of selecting the column after importing the DataFrame and setting the DataType when importing data are not popular among developers. These smells were derived from an unofficial Pandas style guide. This suggests that rules derived from grey literature are opinion-based and need to be validated before putting the static analysis tool into use.

Rules derived from grey literature are opinion-based and need to be validated before putting the static analysis tool into use.

After collecting some smells from various sources, one difficulty in developing a static analysis tool is mapping the smell to the checker rules. Some smells are, by default, at a very low abstraction level. For example, the `DataFrame Conversion` smell is specific to Pandas and links to a specific API `.values`. The checker directly verifies whether `.values` is used. However, there are some smells at a very high abstraction level. Originally `pipeline-not-used` checker was named `data-leakage` checker and the rule is to check whether pipeline API is used around the machine learning class. However, not using Pipeline API does not directly point to the data leakage problem. Using it as an indicator for whether data is leaked will cause an excessive number of false positives.

Moreover, the code can be heterogeneous and thus the implementation of the checker must cover more cases during the development iterations. For instance, the example code for `Gradients Not Cleared before Backward Propagation` shows as Listing 3.3 in the source. The `gradient-clear-pytorch` checker checks whether `.zero_grad()` is used in the same abstract syntax tree layer if `.backward()` and `.step()` are used, according to the code example. However, in the real life code, the `.zero_grad()` is used in the for loop instead of in the same layer, as shown in 3.4.

Listing 3.3: Gradients Not Cleared before Backward Propagation Example from the source

```
output = model(input)
optimizer.zero_grad()
loss_fn.backward()
optimizer.step()
```

Listing 3.4: Gradients Not Cleared before Backward Propagation Example from a notebook

```
for i,data in enumerate(trainLoader):
    ...
    optimizer1.zero_grad()
    ...

totalLoss1 = running_loss/(i+1)
totalLoss1.backward()
optimizer1.step()
```

> Mapping the smell to the checker rules is one of the challenges associated with developing a static analysis tool.

Another obstacle to implementing the code smell checkers is that some smells require context in the execution environment. For example, to check the `Matrix Multiplication API Misused` smell, the array dimension of the input array must be known. Therefore, it is not possible to develop a static analysis solution to detect whether this type of API is misused.

Some smells require context in the execution environment, which cannot be detected by a static analysis tool.

Still, two-thirds of checkers reach 100% true positive rate shows that our static analysis tool is effective, and it is useful to use a static analysis tool to detect the machine learning library API usage violations. We recommend applying our tool in the machine learning application to improve the code quality. In addition, the result shows the general programming language linter can be extended for specific use cases. If there is some common knowledge in the company or organization, they can write and apply their own rules. Some smells are dropped in our previous study for not having enough evidence. For example, a large learning rate could cause a potential problem in the code. However, since we could not find a recommended threshold for checking the learning rate violation, we dropped this smell. If this kind of common knowledge is established in an organization, they can self-define the rule and conduct the check.

Our static analysis tool is generally effective, and we recommend applying our tool in the machine learning application to improve the code quality. In addition, the tool can be easily extended for specific checks in companies or organizations.

3.6.3 Discussions about the code smell prevalence difference between notebooks and regular projects

Six rules are common to both notebooks and regular projects, indicating that some rules are easily broken in both notebooks and projects. Regardless of notebooks and projects, `datatype-pandas` and `column-selection-pandas` are the two rules with the most violations. The `Randomness Uncontrolled` and `DataFrame Conversion API Misused` are also popular smells regardless of projects or notebooks. The remaining four most prevalent code smells in notebooks are all associated with Scikit-learn libraries, while those in projects are associated with NumPy, PyTorch, and TensorFlow. This may be attributable to the widespread use of the Scikit-learn library in notebooks. We analyze library usage in notebooks and projects separately. Table 3.6 compares the percentage of libraries utilized in regular projects and notebooks. The percentage of using Scikit-learn in notebooks is much higher than in regular projects.

Table 3.6: The percentage of libraries in notebooks & The percentage of libraries in regular projects

Library	Notebooks	Regular Projects
NumPy	87.6%	58.2%
Pandas	89.7%	18.2%
Scikit-learn	57.7%	23.6%
TensorFlow	13.1%	24.2%
PyTorch	6.1%	23.6%

Some rules are easily broken in both notebooks and projects, while some rules are not due to the different characteristics of notebooks and projects.

The percentage of regular projects with zero smells is higher than that of notebooks, indicating that the code quality is generally higher in regular projects. This could result from the peer review process and code quality control measures established in the projects. However, there can be more instances of code smells and more types of code smell in a single regular project, because projects typically contain more code, and the code is diverse. The highest number of code violations in one single regular project is 702. This also indicates that it is easier for regular projects to accumulate technical debt.

The code quality is generally higher in regular projects. However, there can be more instances of code smells and more types of code smell in a regular project. This is because projects typically contain more code, and the code is diverse.

One observation from investigating the raw data is that the “interview” project has 89 violations and 11 different kinds of code smell. It suggests that the individuals who use the online tutorials may learn some suboptimal API usages. The poor quality of online tutorials might be one of the reasons for the prevalence of code smells.

One of the reasons for the prevalence of code smells might be the poor quality of the online tutorials.

3.7 Thread to Validity

A few factors act as threats to the validity of our study which we have tried as much as possible to minimise.

Firstly, the datasets might not be able to represent the machine learning codebases that are put into production in the industry, because they are all collected from the public repositories. The project from the industry might have a higher code quality standard. However, we argue that we collected both notebooks and regular projects, and some of the projects are from the public repositories of big companies. We tried our best to represent the Python projects in the real world by collecting four datasets.

Secondly, we only selected 20 instances to calculate the true positive rate of the `dslinter`. We can not guarantee that the true positives and false positives distribution in the 20 instances is identical to the distribution in all instances. Nonetheless, we attempt to compensate for this by selecting the 20 instances randomly.

Moreover, the criterion for a true positive is that the code instance has a potential refactoring opportunity or bug. However, to what extent can it affect the software and whether the project developer perceives it as a refactoring opportunity still needs to be verified.

Lastly, in this research, we only study the code smell prevalence in Python. There are probably some common machine learning coding faults in other programming languages as well. Further studies still need to be conducted to verify the prevalence of machine learning-specific code smells in different programming languages.

3.8 Conclusions and Future Work

In this chapter, we extended a static analysis tool `dslinter` to automatedly detect the machine learning-specific code smells. By using the tool and running the tool on four datasets, we further investigate the prevalence of code smells for machine learning applications. The results show that many best practices stated in the official documentation are violated, while developers have a different coding preference from the unofficial documentation. In addition, the awareness to preserve reproducibility still needs to arise. Our tool helps machine learning practitioners be aware of the possible faults and best practices, and aims to improve the machine learning code quality and software reliability.

Future work will include further research into the tool's validity. Pull request of the refactoring code can be created to see if the developers agree with the refactoring. Intelligent refactoring recommendation tools can be created to recommend refactoring solutions. In the study, we also felt that it is not easy to use linters on jupyter notebooks. Therefore, how to better integrate existing code quality solutions into notebooks can be investigated. In addition, the Pylint-Pycharm graphics interface has not supported the plugin yet, which calls for further effort. Last but not least, the rule of the tool can still be finer tuned for better accuracy.

Chapter 4

Conclusions and Future Work

This chapter provides an overview of the project's conclusions and research directions for future work.

4.1 Conclusions

In this thesis, we identified several code smells specific to machine learning applications and developed a static analysis tool `dslinter` to improve the machine learning application code quality.

In Chapter 2, we conducted an empirical study to collect the code smells specific for machine learning applications. The code smells were collected by us from various sources, including 1750 papers, 2170 grey literature entries, the existing bugs datasets including 88 Stack Overflow posts and 87 GitHub commits and 403 complementary Stack Overflow posts. We analyzed the pitfalls mentioned in the posts and decided whether to take it as a code smell. 22 code smells were collected in the end, including general and API-specific smells. The code smells are also classified by different pipeline stages and their effects. We want to raise the discussion about machine learning-specific code smells and help improve code quality in the machine learning community in this way.

In Chapter 3, we developed a static analysis tool `dslinter` to automatedly detect the machine learning-specific code smells. By using the tool and running the tool on four datasets, we further investigate the prevalence of code smells for machine learning applications. The results show that many best practices stated in the official documentation are violated, while developers have a different coding preference from the unofficial documentation. In addition, the awareness to preserve reproducibility still needs to arise. Our research helps machine learning practitioners be aware of the possible faults and best practices, and aims to improve the machine learning code quality and software reliability.

4.2 Future work

Both studies performed in this thesis call for future research directions.

Regarding the machine learning-specific code smells, machine learning practitioners can be interviewed to improve the code smell catalog. Our study mostly investigates the API usage of the machine learning libraries, but further research to look inside the machine learning library implementations will also be interesting. In addition, code smells in different languages can be studied to increase the generalizability of the machine learning-specific code smells. Furthermore, the correlation between code smells and data smells can be studied.

For `dslinter`, the checker rules can still be finer tuned for better accuracy. Future work will also include further research into the tool's validity. Pull requests of the refactoring code can be created to see if the developers agree with the refactoring opportunities. In addition, intelligent refactoring recommendation tools can be created to recommend the refactor solution. Moreover, in the study, we also felt that applying linter to jupyter notebooks is not easy. Therefore, how to better integrate existing code quality solutions into notebooks can be investigated. The Pylint-Pycharm graphics interface has not yet supported the plugin, which also calls for further effort. Finally, ESlint is more popular than Pylint was felt in the study. It would be interesting to study the reason behind it and help promote the machine learning code quality assurance tools.

Bibliography

[1] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[2] Sumon Biswas, Md Johirul Islam, Yijia Huang, and Hridesh 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)*, pages 577–581. IEEE, 2019.

[3] Markus Borg. Agility in software 2.0–notebook interfaces and mlops with buttresses and rebars. In *International Conference on Lean and Agile Software Development*, pages 3–16. Springer, 2022.

[4] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. The ml test score: A rubric for ml production readiness and technical debt reduction. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1123–1132. IEEE, 2017.

[5] International Organization for Standardization/International Electrotechnical Commission et al. Iso/iec 9126–software engineering–product quality, 2001.

[6] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. The state of the ml-universe. *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020. doi: 10.1145/3379597.3387473.

[7] Mark Haakman. Master's thesis, TU Delft, 2020. URL `https://repository.tudelft.nl/islandora/object/uuid:38ff4e9a-222a-4987-998c-ac9d87880907/datastream/OBJ/download`.

[8] Mark Haakman, Luís Cruz, Hennie Huijgens, and Arie van Deursen. Ai lifecycle models need to be revised. *Empirical Software Engineering*, 26(5):1–29, 2021.

[9] MPA Haakman. Studying the machine learning lifecycle and improving code quality of machine learning applications. 2020.

[10] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1110–1121, 2020.

[11] Nick Hynes, D Sculley, and Michael Terry. The data linter: Lightweight, automated sanity checking for ml data sets. In *NIPS MLSys Workshop*, 2017.

[12] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 510–520, 2019.

[13] Jai Kannan, Scott Barnett, Andrew Simmons, Luís Cruz, and Akash Agarwal. Mlsmellhound: A context-aware code analysis tool. In *2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2022.

[14] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[15] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, 2020.

[16] Valentina Lenarduzzi, Francesco Lomio, Sergio Moreschini, Davide Taibi, and Damian Andrew Tamburri. Software quality for ai: Where we are now? In *International Conference on Software Quality*, pages 43–53. Springer, 2021.

[17] Kent Beck Martin Fowler. *efactoring: Improving the Design of Existing Code*. 2018.

[18] Biruk Asmare Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. On the prevalence, impact, and evolution of sql code smells in data-intensive systems. In *Proceedings of the 17th international conference on mining software repositories*, pages 327–338, 2020.

[19] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, 2017. doi: 10.1007/s10664-017-9535-z.

[20] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. Tunability: importance of hyperparameters of machine learning algorithms. *The Journal of Machine Learning Research*, 20(1):1934–1965, 2019.

[21] Gopi Krishnan Rajbahadur, Gustavo Ansaldi Oliva, Ahmed E Hassan, and Juergen Dingel. Pitfalls analyzer: Quality control for model-driven data science pipelines. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 12–22. IEEE, 2019.

[22] D. Sculley, Gary Holt, D. Golovin, Eugene Davydov, Todd Phillips, D. Ebner, Vinay Chaudhary, M. Young, J. Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *NIPS*, 2015.

[23] Alex Serban, Koen van der Blom, Holger Hoos, and Joost Visser. 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)*, pages 1–12, 2020.

[24] Andrew J Simmons, Scott Barnett, Jessica Rivera-Villicana, Akshat Bajaj, and Rajesh 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)*, pages 1–11, 2020.

[25] Dag I.K. Sjøberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013. doi: 10.1109/TSE.2012.89.

[26] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. An empirical study of refactorings and technical debt in machine learning systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 238–250. IEEE, 2021.

[27] Sylvain Thénault et al. Pylint. *Code analysis for Python*, 2001.

[28] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering*, 46(8):863–891, 2018.

[29] Bart van Oort, Luís Cruz, Maurício Aniche, and Arie van Deursen. 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, 2021. doi: 10.1109/WAIN52551.2021.00011.

[30] Bart van Oort, Luís Cruz, Babak Loni, and Arie van Deursen. "project smells" – experiences in analysing the software quality of ml projects with mllint. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering In Practice (ICSE-SEIP)*, 2022.

[31] Aiko Yamashita and Leon Moonen. To what extent can maintenance problems be predicted by code smell detection? – an empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2013.08.002. URL `https://www.sciencedirect.com/science/article/pii/S0950584913001614`.

[32] Nicholas C Zakas. Eslint.

[33] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 129–140, 2018.

Appendix A

Grey Literature References

A.1 Grey Literature References

(1) Christian Haller. My Machine Learning Model Is Perfect. URL: https://towardsdatascience.com/my-machine-learning-model-is-perfect-9a7928e0f604

(2) Cheng-Tao Chu. Machine Learning Done Wrong. URL: https://ml.posthaven.com/machine-learning-done-wrong

(3) What are common mistakes when working with neural networks? URL: https://www.kaggle.com/general/196487

(4) Top 10 Coding Mistakes Made by Data Scientists. URL: https://www.kdnuggets.com/2019/04/top-10-coding-mistakes-data-scientists.html

(5) Igor Susmelj, Lucas Vandroux, Daniel Bourke (2022). A PyTorch Tools, best practices & Styleguide. URL: https://github.com/IgorSusmelj/pytorch-styleguide

(6) EffectiveTensorflow. URL: https://github.com/vahidk/EffectiveTensorflow

(7) Josh Levy-Kramer (2021). Pandas Style Guide. URL: https://github.com/joshlk/pandas_style_guide

(8) Scikit-Learn Documentation. URL: https://scikit-learn.org/stable/common_pitfalls.html

(9) PyTorch Documentation. Reproducibility. URL: https://pytorch.org/docs/stable/notes/randomness.html

(10) Alexandra Deis. In-place Operations in PyTorch. URL: https://towardsdatascience.com/in-place-operations-in-pytorch-f91d493e970e

(11) GitHub Commit. URL: https://github.com/bamos/dcgan-completion.tensorflow/commit/e8b930501dffe01db423b6ca1c65d3ac54f27223

(12) Samual Sam (2018). Inplace operator in Python. URL: https://www.tutorialspoint.com/inplace-operator-in-python

(13) Github Commit – Tensor Flow. URL: https://github.com/tensorflow/models/commit/90f63a1e1653

(14) Pandas Documentation. Essential basic functionality – Iteration. URL: https://pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#iteration

(15) Vectorization, Part 2: Why and What? URL: https://www.quantifisolutions.com/vectorization-part-2-why-and-what/

(16) Scikit-Learn Documentation. URL: https://scikit-learn.org/stable/modules/preprocessing.html

(17) Stack Overflow. GridSearchCV extremely slow on small dataset in scikit-learn. URL: https://stackoverflow.com/questions/17455302/gridsearchcv-extremely-slow-on-small-dataset-in-scikit-learn/23813876#23813876

(18) Feature scaling. URL: https://en.wikipedia.org/wiki/Feature_scaling

(19) TensorFlow Documentation. Backend: clear_session. URL: https://www.tensorflow.org/api_docs/python/tf/keras/backend/clear_session

(20) Stack Overflow. Tensorflow OOM on GPU. URL: https://stackoverflow.com/questions/42495930/tensorflow-oom-on-gpu

(21) Stack Overflow. Tensorflow NaN bug? URL: https://stackoverflow.com/questions/33712178/tensorflow-nan-bug

(22) Stack Overflow. TensorFlow's ReluGrad claims input is not finite. URL: https://stackoverflow.com/questions/33699174/tensorflows-relugrad-claims-input-is-not-finite

(23) Stack Overflow. Tensorflow - Convolutionary Net: Grayscale vs Black/White training. URL: https://stackoverflow.com/questions/39487825/tensorflow-convolutionary-net-grayscale-vs-black-white-training

(24) Stack Overflow. Implement MLP in tensorflow. URL: https://stackoverflow.com/questions/35078027/implement-mlp-in-tensorflow

(25) Weight Initialization Techniques in Neural Networks. URL: https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78

(26) Stack Overflow. Best practices for generating a random seeds to seed Pytorch? URL: https://stackoverflow.com/questions/57416925/best-practices-for-generating-a-random-seeds-to-seed-pytorch

(27) Stack Overflow. Keras Regression using Scikit Learn StandardScaler with Pipeline and without Pipeline. URL: https://stackoverflow.com/questions/43816718/keras-regression-using-scikit-learn-standardscaler-with-pipeline-and-without-pip/43816833#43816833

(28) Ask a Data Scientist: Data Leakage. URL: https://insidebigdata.com/2014/11/26/ask-data-scientist-data-leakage/

(29) Data Leakage. URL: https://www.kaggle.com/alexisbcook/data-leakage

(30) Pandas Documentation. URL: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#indexing-view-versus-copy

(31) Stack Overflow. Extrapolate values in Pandas DataFrame. URL: https://stackoverflow.com/questions/22491628/extrapolate-values-in-pandas-dataframe/35959909#35959909

(32) Stack Overflow. Why does one use of iloc() give a SettingWithCopyWarning, but the other doesn't? URL: https://stackoverflow.com/questions/53806570/why-does-one-use-of-iloc-give-a-settingwithcopywarning-but-the-other-doesnt/53807453#53807453

(33) Stack Overflow. Convert pandas dataframe to NumPy array. URL: https://stackoverflow.com/questions/13187778/convert-pandas-dataframe-to-numpy-array/54508052#54508052

(34) Stack Overflow. Does np.dot automatically transpose vectors? URL: https://stackoverflow.com/questions/54160155/does-np-dot-automatically-transpose-vectors/54161169#54161169

(35) Linear Algebra (numpy.dot). NumPy Documentation. URL: https://numpy.org/doc/stable/reference/generated/numpy.dot.html#numpy.dot

(36) Yuval Greenfield. Most Common Neural Net PyTorch Mistakes. URL: https://medium.com/missinglink-deep-learning-platform/most-common-neural-net-pytorch-mistakes-456560ada037

(37) Stack Overflow. Is this a right way to train and test the model using Pytorch? URL: https://stackoverflow.com/questions/67066452/is-this-a-right-way-to-train-and-test-the-model-using-pytorch/67067242#67067242

(38) Why does detach reduce the allocated memory? URL: `https://discuss.pytorch.org/t/why-does-detach-reduce-the-allocated-memory/43836`

(39) Dot product. Wikipedia. URL: `https://en.wikipedia.org/wiki/Dot_product`

(40) Stack Overflow. What is the rationale for all comparisons returning false for IEEE754 NaN values? URL: `https://stackoverflow.com/questions/1565164/what-is-the-rationale-for-all-comparisons-returning-false-for-ieee754-nan-values`

(41) Broadcasting the good and the ugly URL: `https://effectivemachinelearning.com/PyTorch/3._Broadcasting_the_good_and_the_ugly`