Code Smells for Machine Learning Applications

Haiyin Zhang
haiyin.zhang@ing.com
AI for Fintech Research, ING
Amsterdam, Netherlands

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

Arie van Deursen
Arie.vanDeursen@tudelft.nl
Delft University of Technology
Delft, Netherlands

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. In particular, code smells have rarely been studied in this domain. 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 avoid issues in the long run. This paper 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 code smell catalog helps data scientists and developers produce and maintain high-quality machine learning application code.

KEYWORDS

Code Smell, Anti-pattern, Machine Learning, Code Quality, Technical Debt

1 INTRODUCTION

Despite the large increase in the popularity of machine learning applications [3], there are several concerns regarding the quality control and the inevitable technical debt growing in these systems [16]. 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 [12, 17]. Yet, previous research shows that practitioners are eager to learn more about engineering best practices for their machine learning applications [5].

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 [10] and data quality [7]. However, the quality assurance of machine learning code has not been highlighted [12]. Recent work studied the code quality for machine learning applications in a general way, finding some code quality issues such as duplicated code [20] and violations of traditional naming convention [17]. 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 [20]. 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 [22]. 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 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 [11].

Listing 1: Coding Pitfall Example from [4]

```
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 paper 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 RELATED WORK

Code smells are common poor code design choices that negatively affect the systems and violate the best practice or original design vision [11]. Martin Fowler introduced a general code smell list in his book [13]. 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 [14, 18, 21].

The prevalence of traditional code smells in machine learning projects was studied in van Oort et al.'s paper [20]. 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 [19]. 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 [22]. 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 [8]. They adopted some of the root causes of deep learning bugs from [22] 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 [6]. 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 [15]. 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 [1]. 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.

3 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 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.

3.1 Paper Mining

Our methodology for paper mining is described as follows, and shown as Figure 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 3. After analyzing papers from the initial result set, we reach a level of saturation for each query after
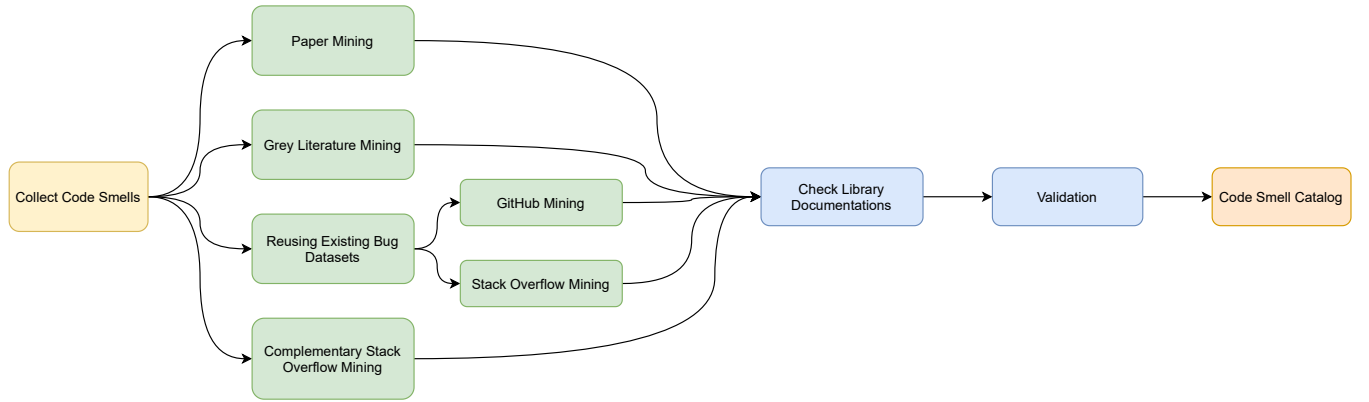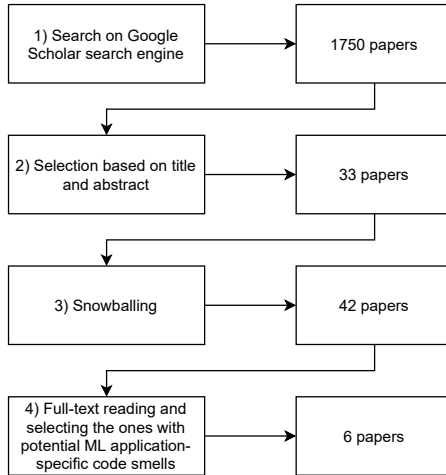
Figure 1: Methodology

Figure 2: Paper Selection Process

Figure 3: Search Query for Literature and Grey Literature

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 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 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: [1, 4, 6, 8, 15, 22].

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 3). We also apply a back-cutting

Table 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

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 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 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 in the Appendix: (1), (2), (3), (4), (5), (6), (7), (8).

3.3 Reusing Existing Bug Datasets

We reuse the dataset provided in the work by Zhang et al. [22] 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.

3.4 Complementary 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 [8] and verified in the TensorFlow Bugs replication package [22].
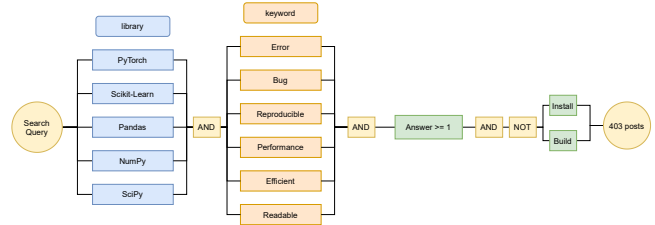
Figure 4: Search Query for Stack Overflow Mining
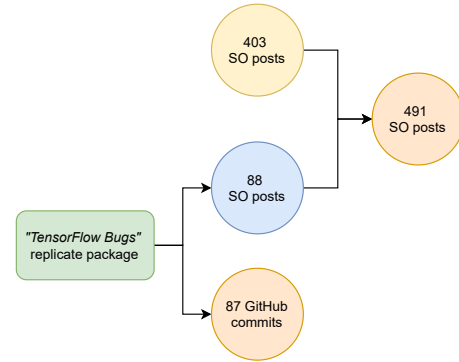
Figure 5: Total Number of Stack Overflow Posts and GitHub Commits

1) Library Selection: We use five libraries: PyTorch, Scikit-Learn, Pandas, NumPy, and SciPy, excluding TensorFlow from the six libraries mentioned in Section 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 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 [2]: 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 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* [22], we have 87 GitHub commits and 491 Stack Overflow posts in our dataset, as presented in Figure 5.

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.

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, where n refers to the nth element in the list.

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 [4]. 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.

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* [4]. 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.

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.

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.

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.

Table 2: Code Smell Catalog

Code Smell	Pipeline Stage	Effect	Type	Literature	Grey Literature	GitHub Commits	SO Posts
Unnecessary Iteration	Data Cleaning	Efficiency	Generic	[4]	(6)(14)	(13)	
NaN Equivalence Comparison Misused	Data Cleaning	Error-prone	Generic	[4]			
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	[4]		(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	[4][1][6]			
Memory Not Freed	Model Training	Memory Issue	Generic	[6]	(5)(19)		(20)
Deterministic Algorithm Option Not Used	Model Training	Reproducibility	Generic	[1]	(9)		
Randomness Uncontrolled	Model Training & Model Evaluation	Reproducibility	Generic	[1]	(1)(5)(9)		(26)
Missing the Mask of Invalid Value	Model Training	Error-prone	Generic	[22][6]			(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	[4]	(8)		(27)
Threshold-Dependent Validation	Model Evaluation	Robustness	Generic	[15]			

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.

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 *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.

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 [4]. 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).

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()*.

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).

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.

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 [4]. 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.

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.

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.

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.

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 [22]. 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.

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.

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.

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.

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()*.

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()*.

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 [4].

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.

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 [15].

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

5 DISCUSSIONS AND IMPLICATIONS

The code smell catalog summarized from the empirical study is presented in Table 2. 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.

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.

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 the recommendations. For example, the effect of index chaining (cf. Section 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.

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 [9]. 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.

5.4 Implication to Students

As mentioned by [4], 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.

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).

7 CONCLUSION 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.

REFERENCES

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

[2] International Organization for Standardization/International Electrotechnical Commission et al. 2001. ISO/IEC 9126–Software Engineering–Product Quality.

[3] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. 2020. The State of the ML-universe. *Proceedings of the 17th International Conference on Mining Software Repositories* (2020). https://doi.org/10.1145/3379597.3387473

[4] MPA Haakman. 2020. Studying the Machine Learning Lifecycle and Improving Code Quality of Machine Learning Applications. (2020).

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

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

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

[8] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. 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*. 510–520.

[9] Jai Kannan, Scott Barnett, Andrew Simmons, Luís Cruz, and Akash Agarwal. 2022. MLSmellHound: A Context-Aware Code Analysis Tool. In *2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*.

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

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

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

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

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

[15] Gopi Krishnan Rajbahadur, Gustavo Ansaldi Oliva, Ahmed E Hassan, and Juergen Dingel. 2019. 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)*. IEEE, 12–22.

[16] D. Sculley, Gary Holt, D. Golovin, Eugene Davydov, Todd Phillips, D. Ebner, Vinay Chaudhary, M. Young, J. Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *NIPS*.

[17] Andrew J Simmons, Scott Barnett, Jessica Rivera-Villicana, Akshat Bajaj, and Rajesh Vasa. 2020. A large-scale comparative analysis of Coding Standard conformance in Open-Source Data Science projects. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.

[18] Dag I.K. Sjøberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dybå. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1144–1156. https://doi.org/10.1109/TSE.2012.89

[19] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 238–250.

[20] Bart van Oort, Luís Cruz, Maurício Aniche, and Arie van Deursen. 2021. The Prevalence of Code Smells in Machine Learning projects. In *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*. 1–8. https://doi.org/10.1109/WAIN52551.2021.00011

[21] Aiko Yamashita and Leon Moonen. 2013. To what extent can maintenance problems be predicted by code smell detection? – An empirical study. *Information and Software Technology* 55, 12 (2013), 2223–2242. https://doi.org/10.1016/j.infsof.2013.08.002

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

A 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/wh

at-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