# Code Smells for Machine Learning Applications

**ABSTRACT:**

Machine learning is super popular, but there's a problem – we don't have clear rules for writing good code for it. This paper says that even though machine learning code is just a small part of a bigger system, it's crucial for how everything works. So, the paper suggests 22 common mistakes (called "code smells") in machine learning code and how to fix them. These mistakes were found by looking at research papers, online discussions, and actual code. The goal is to help data scientists and developers create and take care of good-quality machine learning code.

KEYWORDS

Code Smell: It's like a funny odor in code that suggests something might be wrong. Just like a bad smell warns you of something off, code smells signal potential issues in the way code is written.

Anti-pattern: This is like a common mistake or a bad habit in writing code. It's a known way of doing things that often leads to problems. Think of it as a "don't do this" warning in coding.

Machine Learning: This is when computers learn from examples and experiences, getting better at tasks over time. It's like teaching a computer to recognize patterns or make decisions without being explicitly programmed for each step.

Code Quality: This is about how well-written and organized the code is. Good code is clear, easy to understand, and less likely to have errors. It's like having a neat and organized room where you can find things easily.

Technical Debt: Imagine you're in a hurry and take a shortcut while building something. Later, you'll need to go back and fix or improve what you rushed through. That's technical debt – it's the extra work you create for yourself by not doing things properly the first time.

## 1.INTRODUCTION:

Machine learning is super popular, but there's a problem – we're not paying enough attention to the quality of the code that makes it work. The teams working on machine learning are made up of experts from different fields, and they often don't know much about good software practices. Even though there's a lot of interest in improving the models and data, people haven't been focusing on making sure the code itself is good.

Now, having bad code in machine learning is a big deal. It can lead to serious problems, and unlike regular software, finding these issues is harder and takes a lot of time. So, the paper is saying that we need to do more to make sure the code for machine learning is top-notch.

One way to make code better is by getting rid of "code smells" and "anti-patterns." Code smells are like bad signs in the code, indicating potential problems. The paper gives an example of a "pitfall" in Python code, showing a slow way of doing something and a faster, better way. Fixing these issues is important not just for making the code run faster but also for preventing future problems. The goal is to make sure machine learning code is easy to understand, maintain, and doesn't cause issues down the road. To improve the quality of code in machine learning applications and make development easier, the paper conducted a study to find common issues in machine learning code. They asked, "What are the usual problems in machine learning code?"

The key contributions of the paper are:

1. A list of common issues (code smells) specific to machine learning code.
2. A dataset from various sources like papers, online discussions, code changes on GitHub, and posts on Stack Overflow, providing practical insights into machine learning code quality.

## 2. RELATED WORK

The paper talks about "code smells," which are common mistakes in coding that can cause problems in the long run. These mistakes have been widely studied, and it's known that they can decrease code quality and make it more prone to errors.

The authors specifically focus on machine learning code and mention that the usual code smell studies don't pay much attention to this area. They explain that machine learning projects have their own challenges, and even existing tools might struggle to analyze them correctly.

While there are some studies on bugs and refactoring in machine learning, the paper stands out in a few ways:

1. It gives practical advice in the form of "code smells" to improve code and prevent issues.
2. It looks specifically at code-level problems.
3. It not only focuses on bugs but also considers issues related to performance, reproducibility, and maintainability.
4. It expands its focus beyond deep learning, including other machine learning tasks using popular libraries like Scikit-Learn, Pandas, NumPy, and SciPy.

The paper also distinguishes itself from other studies that look at pitfalls in data science projects or provide tests and monitoring needs for machine learning. Instead, this paper creates a catalog of machine learning-specific code smells to guide developers toward better coding practices by getting rid of these issues.

## 3.METHODOLOGY:

The process of gathering machine learning-specific code smells involves examining academic literature, grey literature, community-based coding Q&A platforms like Stack Overflow, and public software repositories like GitHub. The approach includes mining research papers, grey literature,

utilizing existing bug datasets, and conducting Stack Overflow mining. The collected code smells are cross-referenced with recommendations from official documentation of machine learning libraries. Finally, two authors validate the catalog of identified code smells.

### 3.1 Paper Mining

Google Scholar Search:

The team used Google Scholar to find papers on code smells in machine learning projects. They combined machine learning and code quality-related keywords like "Artificial Intelligence," "Machine Learning," "Technical Debt," and "Code Smell" in their search queries.

1.Query Analysis and Saturation:

After analyzing the initial results, the team reached a saturation point for each query by consulting the first five pages (50 results) for each query. This resulted in a total of 1750 papers.

2. Selection based on title and abstract:

The team excluded papers related to Machine Learning for Software Engineering (ML4SE) and selected papers based on titles and abstracts. This step resulted in 33 papers after excluding duplicates.

3.Snowballing:

The team used forward and backward snowballing, browsing the references and citations of the 33 papers. They selected papers based on title and abstract, avoiding duplicates, and added nine more papers.

4. Full-text reading and selecting the ones with potential machine learning-specific code smells:

The team read the full text of the 42 selected papers and identified six final papers contributing to the machine learning-specific code smell catalog.

In summary, the team systematically searched Google Scholar, refined their selection based on title and abstract, used snowballing for additional papers, and finally, carefully read the full text to identify papers contributing to the machine learning code smell catalog.

### 3.2 Grey Literature Mining

In this study, grey literature, which includes online content like blog posts from experienced practitioners, is considered a valuable source for machine learning-specific coding advice. To collect this

information, the study employs a Google search engine strategy using queries similar to those used for research literature. A back-cutting strategy is applied after the fifth page of results for each query, resulting in 1750 entries.

Additionally, a new set of queries, combining popular machine learning-related Python libraries (TensorFlow, PyTorch, Scikit-Learn, Pandas, NumPy, and SciPy) with code quality-related keywords, is used to search for grey literature. Saturation is achieved by analyzing the first result page, resulting in 420 entries. In total, 2170 entries are collected for grey literature mining.

To filter relevant entries, a three-step process is implemented: 1) reading the title, 2) reading the first summary, and 3) reading the entire article. Not all entries contain actionable coding advice, as some provide general information without specific code-level pitfalls.

Ultimately, eight cornerstone blog posts are identified as contributors to the code smell catalog, listed in Section A of the Appendix as (1), (2), (3), (4), (5), (6), (7), and (8).

### 3.3 Reusing Existing Bug Datasets

The researchers are using a dataset from a previous study by Zhang et al. to find code smells in TensorFlow applications. Zhang et al. had looked at bugs in TensorFlow applications, studied their patterns using 88 Stack Overflow posts and 87 GitHub commits, and shared a replication package of these bugs called the "TensorFlow Bugs" package. Now, the current researchers are reusing this package to identify common issues that might apply to other projects, and they want to document these as code smells.

### 3.4 Complementary Stack Overflow Mining

1) Library Selection:
2) Keyword Selection:
3) Applying Search Terms:

Purpose:

After using existing bug datasets, the team applied a similar study method to other machine learning libraries, focusing on Stack Overflow posts and excluding GitHub commits.

Reasoning:

GitHub and Stack Overflow were chosen for their similar issue patterns, as noted in previous studies [8] and confirmed in the TensorFlow Bugs replication package [22].

Search Criteria:

The team used specific keywords related to software quality (e.g., error, bug), filtered out terms related to installation and building, and selected posts based on the number of answers.

Post Selection Process:

Using the keyword "Error" resulted in the maximum number of posts, so the team ranked and selected the top posts with this keyword, as well as the top posts for other keywords. Duplicated posts were then removed.

Final Dataset:

The process yielded a dataset of 87 GitHub commits and 491 Stack Overflow posts for PyTorch, Scikit-learn, Pandas, NumPy, and SciPy, along with contributions from the TensorFlow Bugs replication package [22].

In summary, the team focused on collecting information from Stack Overflow to complement their dataset, ensuring a comprehensive set of posts related to machine learning code issues.

**3.5 Validation**

In the validation process, the first author gathers all identified code smells from the empirical study, which includes information from papers, grey literature, GitHub, and Stack Overflow. These findings are then discussed with the second author through meetings. The discussions cover an introduction to each code smell, analysis of examples where the issue was found, and the collection of additional evidence. The team looks for references in academic and grey literature that support each identified code smell. In total, 31 code smells were initially collected, but after the validation process, 9 of them were excluded.

**4.RESULTS**

4.1Unnecessary Iteration:

Context:

Loops can be slow and lengthy in code.

Vectorized solutions (operations on entire sets of data) can replace loops.

Problem:

Iterating manually over rows in Pandas and using loops for slicing in TensorFlow can be slow.

Solution:

For data-intensive machine learning applications, use vectorized solutions instead of iterating over individual values. In Pandas, prefer built-in vectorized methods like join and group by. In TensorFlow, use the tf.reduce_sum() API for faster reduction operations rather than combining slicing operations and loops.

4.2NaN Equivalence Comparison Misused

Context:

Comparing NaN (Not a Number) behaves differently from comparing None.

Problem:

While None == None evaluates to True, np.nan == np.nan (in NumPy) evaluates to False.

In Pandas, treating None like np.nan for simplicity and performance reasons leads to DataFrame element comparisons with np.nan always returning False.

This difference might lead to unintentional behaviors in the code if developers are not aware.

Solution:

Developers need to be cautious when using NaN comparison.

4.3 Chain Indexing

Context:

In Pandas, df["one"]["two"] and df.loc[:,("one","two")] produce the same result.

df["one"]["two"] is referred to as chain indexing.

Problem:

Chain indexing, like df["one"]["two"], may lead to performance issues and error-prone code.

For example, using df["one"]["two"] is seen as two separate events by Pandas, potentially causing slower performance.

Assigning to the result of chain indexing can have unpredictable outcomes, as Pandas doesn't guarantee whether it returns a view or a copy, leading to potential assignment failures.

Solution:

Developers using Pandas should avoid chain indexing to prevent performance issues and unpredictable results.

In simple terms, when working with Pandas, it's advisable for developers to avoid using chain indexing like df["one"]["two"]. Instead, using methods like df.loc[:,("one","two")] can be more efficient and less error-prone.

4.4 Columns and DataType Not Explicitly Set

Context:

In Pandas, when importing data into a DataFrame, all columns are selected by default.

The data type for each column is determined based on default dtype conversion.

Problem:

If columns are not explicitly selected, developers may not know what to expect in the downstream data schema.

If data types are not set explicitly, unexpected inputs may silently proceed to the next step, potentially causing errors later.

This issue applies to various data importing scenarios.

Solution:

It is recommended to explicitly set columns and data types during data processing to ensure clarity and prevent potential errors.

4.5 Empty Column Misinitialization

Context:

Developers may need to create a new empty column in a Pandas DataFrame.

Problem:

If zeros or empty strings are used to initialize a new empty column, methods like .isnull() or .notnull() may not work as expected.

This issue may also arise in other data structures or libraries.

Solution:

Use the NaN value (e.g., np.nan) when creating a new empty column in a DataFrame.

Avoid using "filler values" such as zeros or empty strings.

4.6 Merge API Parameter Not Explicitly Set

Context:

The df.merge() API is used for merging two DataFrames in Pandas.

Problem:

Although using the default parameters may produce the same result, explicitly specifying on, how, and validate parameters enhances readability.

The on parameter indicates which columns to join on, how describes the join method (e.g., outer, inner), and validate checks if the merge is of a specified type.

Not specifying these parameters might lead to unexpected results, especially if assumptions about unique merge keys are incorrect.

Merging operations are often computationally and memory-intensive, so it's preferable to perform the merge in one stroke for performance.

Solution:

Developers should explicitly specify parameters (on, how, validate) for merge operations to improve readability and avoid potential issues.

4.7 In-Place APIs Misused

Context:

Data structures can be manipulated either by making changes to a copy of the structure or by modifying the existing structure directly (in-place).

Problem:

Some methods default to in-place changes, while others return a copy.

If a developer assumes an in-place approach but doesn't assign the returned value to a variable, the operation won't affect the final outcome.

For instance, in Pandas, using df.dropna() without assigning the result to a variable or setting the in-place parameter won't update the original DataFrame.

Misuse of in-place operations can lead to unintended outcomes.

Solution:

Developers should check whether the result of the operation is assigned to a variable or if the in-place parameter is set in the API.

It's a misconception that in-place operations in Pandas save memory; a copy of the data is still created.

In PyTorch, in-place operations save GPU memory but risk overwriting values needed for gradient computation.

4.8 Dataframe Conversion API Misused

Context:

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

Problem:

df.values() has an inconsistency problem, making it unclear whether the returned value is the actual array, a transformation of it, or one of the Pandas custom arrays.

Despite being noted as a warning in the documentation, using df.values() doesn't log a warning or error during code compilation.

Solution:

When converting a DataFrame to a NumPy array, it's better to use df.to_numpy() instead of df.values().

4.9 Matrix Multiplication API Misused

Context:

When performing multiplication on two-dimensional matrices in NumPy, both np.matmul() and np.dot() yield the same result.

Problem: In mathematics, the dot product result is expected to be a scalar, not a vector, but np.dot() returns a new matrix for two-dimensional matrix multiplication.

This inconsistency with mathematical semantics can lead to misuse, with developers using np.dot() in scenarios where it's not appropriate, such as two-dimensional multiplication.

Solution: When multiplying two-dimensional matrices, it's preferable to use np.matmul() over np.dot() for clearer semantic consistency with mathematical expectations.

4.10 No Scaling before Scaling-Sensitive Operation

Context:

Feature scaling is a method used to bring features from different value ranges to the same range.

Problem:

Certain operations, like Principal Component Analysis (PCA), Support Vector Machine (SVM), Stochastic Gradient Descent (SGD), Multi-layer Perceptron classifier, and L1 and L2 regularization, are sensitive to feature scaling.

Neglecting feature scaling before such operations can lead to incorrect results. For instance, without scaling, a variable on a larger scale might dominate PCA, producing inaccurate principal components.

Solution:

To prevent issues, it's important to check whether feature scaling is applied before performing operations that are sensitive to scaling.

### 4.11 Hyperparameter Not Explicitly Set

Context:

Hyperparameters are parameters set before the learning process to control the behavior of the training algorithm.

Problem:

Default hyperparameters in machine learning library APIs might not be optimal for specific data or problems, potentially leading to suboptimal results.

Default hyperparameter values may change in new library versions, affecting model behavior.

Lack of explicit hyperparameter setting hinders replicating the model in different programming languages.

Solution:

Set hyperparameters explicitly and tune them for better results and reproducibility.

### 4.12 Memory Not Freed

Context:

Machine learning training is memory-intensive, and system memory is limited.

Problem:

Running out of memory during training can lead to training failure.

Solution:

Use provided APIs to free up memory in deep learning libraries.

In TensorFlow, use clear_session() in a loop if creating models repeatedly.

In PyTorch, use .detach() to free tensors from the graph when possible to prevent unnecessary memory usage.

4.13 Deterministic Algorithm Option Not Used

Context:

Using deterministic algorithms can improve reproducibility in machine learning.

Problem:

Non-deterministic algorithms may not produce repeatable results, making debugging inconvenient.

Solution:

Set deterministic algorithm options to True during development for reproducibility.In PyTorch, use torch.use_deterministic_algorithms(True) when debugging. Avoid using this option in the deployment stage for better performance.

4.14 Randomness Uncontrolled

Context:

Randomness is involved in certain scenarios during machine learning development, like algorithms with inherent randomness or dataset splitting in cross-validation.

Problem:

Without setting a random seed explicitly, results become irreproducible, making debugging challenging and replication of studies difficult.

Solution:

Set a global random seed during development for reproducibility. In Scikit-Learn, PyTorch, Numpy, and similar libraries, explicitly set a random seed to ensure consistent results. For DataLoader in PyTorch, set a random seed to ensure consistent data splitting and loading.

4.15 Missing the Mask of Invalid Value

Context:

In deep learning, variable values change during training, potentially leading to invalid values for certain operations.

Problem:

Invalid values, especially near zero, in operations like tf.log() can cause errors that are hard to debug and may not directly point to the problematic code line.

Solution:

Add a mask for potential invalid values to prevent errors. For example, use tf.clip_by_value() to wrap the argument for tf.log() to avoid it turning to zero. By adding a mask, like tf.clip_by_value(x, 1e-10, 1.0), where values below 1e-10 are replaced, potential errors can be avoided.

4.16 Broadcasting Feature Not Used

Context Deep learning libraries like PyTorch and TensorFlow sup□ports the element-wise broadcasting operation. Problem Without broadcasting, tiling a tensor first to match an□other tensor consumes more memory due to the creation and stor□age 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

Context:

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

Problem:

If an array is initialized using tf.constant() and the developer tries to assign a new value to it in a loop to make it grow, it can lead to errors.

Using the low-level tf.while_loop() API to fix this error is inefficient, as it involves building many intermediate tensors.

Solution:

Utilize tf.TensorArray() for growing arrays in loops, especially in TensorFlow 2.

Developers should leverage new data types provided by libraries for more efficient solutions.

4.18 Training / Evaluation Mode Improper Toggling

Context:

In deep learning code using PyTorch, calling .eval() switches to evaluation mode, deactivating the Dropout layer.

Problem:

Forgetting to toggle back to training mode after the inference step can lead to the Dropout layer not being utilized in some data training, affecting the training result.

Similar issues may arise in TensorFlow.

Solution:

Developers should call the training mode at the appropriate place in the code.

Ensure that there's no oversight in switching back to the training mode after the inference step to maintain the intended behavior of the Dropout layer.

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 for□ward the input into the network in PyTorch. Problem In PyTorch, self.net() and self.net.forward() are not identi□cal. 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

Context:

In PyTorch, the sequence optimizer.zero_grad(), loss_fn.backward(), optimizer.step() is crucial for gradient descent optimization.

Problem:

Forgetting to use optimizer.zero_grad() before loss_fn.backward() can lead to accumulated gradients from previous iterations, causing gradient explosion and training failure.

Solution:

Developers should use optimizer.zero_grad(), loss_fn.backward(), optimizer.step() in that order.

Ensure that optimizer.zero_grad() is applied before loss_fn.backward() to clear gradients before the backpropagation step.

4.21 Data Leakage

Context:

Data leakage occurs when information from the prediction result is present in the training data.

Problem:

Data leakage leads to overly optimistic experimental outcomes and poor real-world performance, affecting the reliability of machine learning models.

Solution:

Leaky predictors and leaky validation strategies contribute to data leakage.

Leaky predictors involve modifying or generating features after the goal value is known, detectable only at the data level.

Leaky validation involves mixing training and validation data, detectable at the code level.

Best practice in Scikit-Learn is to use the Pipeline() API to prevent data leakage.

4.22 Threshold-Dependent Validation

Context:

Machine learning model performance is assessed using metrics, which can be threshold-dependent (e.g., F-measure) or threshold-independent (e.g., Area Under the Curve - AUC).

Problem:

Choosing a specific threshold for threshold-dependent metrics is challenging and may result in less interpretable results.

Solution:

Use threshold-independent metrics for model evaluation, as they are more robust and less dependent on specific threshold values.

## 5.DISCUSSIONS AND IMPLICATIONS

The code smell catalog resulting from the empirical study includes 22 code smells categorized into four stages of the machine learning pipeline: Data Cleaning, Feature Engineering, Model Training, and Model Evaluation. These code smells can impact application code by making it error-prone, less efficient, less reproducible, causing memory issues, less readable, and less robust. Among the identified smells, 16 are generic and occur irrespective of the library used, while 6 are specific to the library's API design.

This catalog is valuable for understanding common flaws in machine learning application development, offering insights into recurring code issues from diverse sources. It is particularly helpful for data scientists without a strong software engineering background, providing guidelines for developing machine learning applications.

The study notes that machine learning libraries regularly evolve with new versions, and some code smells identified in the "TensorFlow Bugs" replication package have already been deprecated in TensorFlow version 2. The researchers anticipate that new API-specific code smells will emerge with future versions and library features. They emphasize the importance of a code smell catalog in fostering continuous collaboration between practitioners and academics.

Acknowledging the fast-paced evolution of AI frameworks, the study identifies three code smells as potentially temporary: Dataframe Conversion API Misused, Matrix Multiplication API Misused, and Gradients Not Cleared before Backward Propagation. While other smells are expected to persist, these temporary ones may become obsolete in a few years. Despite their potential transience, these smells are deemed important for flagging to help practitioners prevent downstream issues.

### 5.1 Implications to Data Scientists and Machine Learning Application Developers

The catalog of code smells has implications for data scientists and machine learning developers. It includes various smells from different sources and stages of development, each having different effects. For example, the "Unnecessary Iteration" smell indicates inefficient code often found in data cleaning stages, while "Hyperparameter Not Explicitly Set" signals potentially irreproducible code in the model training stage. Developers can use the catalog to check for these issues in their code.

Certain code smells are mentioned multiple times across academic and grey literature, providing practitioners with an indication of their relevance. Developers can use the references associated with these smells to learn more about them.

This catalog is particularly beneficial for machine learning developers, especially those with limited software engineering experience. It helps build awareness of common pitfalls and best practices, aiming to prevent errors in code. The assumption is that being aware of these code smells can expedite development, leading to higher-quality software in production. Future work will investigate whether addressing these code smells results in more accurate training, improved hyperparameter optimization, clearer code, and reduced maintenance efforts.

### 5.2 Implication to Machine Learning Library Developers

The catalog of code smells has implications for developers of machine learning libraries. Some of the identified smells stem from APIs requiring specific usage patterns that may not be intuitive for users. For instance, addressing the "Dataframe Conversion API Misused" smell could involve deprecating one API method and replacing it with another for clarity. The results highlight the crucial role of library API design in preventing potential issues in projects.

While some identified smells are already mentioned in library documentation, developers still create code that does not follow these recommendations. This suggests that developers might face challenges in strictly adhering to documentation, potentially due to fast development cycles or lack of experience with the library. The study suggests that merely indicating warnings in documentation may not be enough. Library developers should actively engage in community forums, like Stack Overflow, to help address non-obvious issues.

Lastly, the study emphasizes the importance of library maintainers reaching out to and collaborating with existing projects that support the development of machine learning software, such as static code analysis tools, testing tools, quality auditors, and experiment trackers. Library developers' insights are crucial in developing coding tools that align with best practices.

### 5.3 Implication to Code Analysis Tool Developers

For developers creating code analysis tools, this study suggests that such tools play a crucial role in promoting best practices and alerting application developers to potential pitfalls that might not be addressed solely through better API design.

The research serves as a foundation for future work on automated tools designed to identify and address these undesirable code patterns. These tools aim to reduce developers' effort in discovering and eliminating code smells, providing valuable support for ensuring code quality. Automated tools are seen as advantageous, especially given human tendencies to overlook certain aspects, making it essential to have technology explicitly checking whether best practices are followed.

Furthermore, the study notes that some code smells are context-dependent. In line with previous work on context-aware code analysis tools for machine learning applications, the automated tools can adapt to different contexts. For instance, they can have distinct configuration settings for development and deployment stages based on the specific requirements of each stage in the machine learning pipeline.

### 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.

## 5THREATS TO VALIDITY

In our study, there are potential threats to the validity of the results. The manual inspection of code smells by the first author may introduce bias due to varying understandings of machine learning code. To address this, the second author reviews all instances of code smells, and the two authors discuss their findings.

In the literature survey, initial keyword selection in the search query may miss relevant entries. To mitigate this, we refine keywords iteratively based on retrieved content and use forward and backward snowballing to complement the search.

The quality of results in the grey literature search depends on the Google search engine's relevance sorting algorithm, which is beyond our control. However, we believe this has minimal impact on the study's results.

When mining Stack Overflow and GitHub, we inspect a specific number of entries, and the generalizability of our results is unclear. To cover common mistakes, we use the "highest voted" criteria on Stack Overflow, acknowledging that less-voted instances may also contain issues.

The study focuses on six Python machine learning libraries, potentially limiting coverage of other frameworks. This is addressed by selecting the most popular frameworks.

We recognize that there may be more warnings in library documentation that could be considered code smells. However, we only consider warnings that have been observed to lead to real code issues in other sources, such as Stack Overflow.

## 7 CONCLUSION AND FUTURE WORK

In this research, the authors studied and identified specific issues, called "code smells," in the code of machine learning applications. They gathered information from various sources, including research papers, online discussions, and code repositories, to compile a list of 22 code smells related to machine learning. The goal is to raise awareness about these issues and improve the overall quality of machine learning code.

For future work, the authors plan to conduct a large-scale validation of their findings by interviewing machine learning practitioners and analyzing code changes in GitHub repositories. They also aim to develop a tool that can automatically detect these code smells in machine learning code to encourage best practices. Lastly, they want to explore how common these code smells are in real-world machine learning applications and understand the benefits of using a catalog of machine learning-specific code smells.