A Catalog of Data Smells for Coding Tasks

A Catalog of Data Smells for Coding Tasks

ANTONIO VITALE, Automatica e Informatica, Politecnico di Torino, Torino, Italy and Department of Biosciences and Territory, University of Molise, Pesche, Italy
ROCCO OLIVETO and SIMONE SCALABRINO, Department of Biosciences and Territory, University of Molise, Pesche, Italy

Large Language Models (LLMs) are increasingly becoming fundamental in supporting software developers in coding tasks. The massive datasets used for training LLMs are often collected automatically, leading to the introduction of data smells. Previous work addressed this issue by using quality filters to handle some specific smells. Still, the literature lacks a systematic catalog of the data smells for coding tasks currently known. This article presents a Systematic Literature Review (SLR) focused on articles that introduce LLMs for coding tasks. We first extracted the quality filters adopted for training and testing such LLMs, inferred the root problem behind their adoption (data smells for coding tasks), and defined a taxonomy of such smells. Our results highlight discrepancies in the adoption of quality filters between pre-training and fine-tuning stages and across different coding tasks, shedding light on areas for improvement in LLM-based software development support.

CCS Concepts: • **Software and its engineering → Software verification and validation**; *Extra-functional properties*;

Additional Key Words and Phrases: LLMs for coding tasks; data smells; data quality; Systematic Literature Review

ACM Reference format:
Antonio Vitale, Rocco Oliveto, and Simone Scalabrino. 2025. A Catalog of Data Smells for Coding Tasks. *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 113 (April 2025), 32 pages.
https://doi.org/10.1145/3707457

1 Introduction

Programming languages are powerful and flexible tools that allow developers to express the same instructions in plenty of different ways. In practice, however, programs written by developers present repetitive patterns that can be captured and predicted by statistical language models [23]. **Large Language Models (LLMs)** are an evolution of statistical language models that recently

This publication is part of the project PNRR-NGEU which has received funding from the MUR–DM 118/2023. This work has been partially supported by the European Union-NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 "QualAI: Continuous Quality Improvement of AI-Based Systems," Grant no. 2022B3BP5S, CUP: H53D23003510006.
Authors' Contact Information: Antonio Vitale (corresponding author), Automatica e Informatica, Politecnico di Torino, Torino, Italy and Department of Biosciences and Territory, University of Molise, Pesche, Italy; e-mail: antonio.vitale@polito.it, a.vitale8@studenti.unimol.it; Rocco Oliveto, Department of Biosciences and Territory, University of Molise, Pesche, Italy; e-mail: rocco.oliveto@unimol.it; Simone Scalabrino, Department of Biosciences and Territory, University of Molise, Pesche, Italy; e-mail: simone.scalabrino@unimol.it.

ACM 1557-7392/2025/4-ART113
https://doi.org/10.1145/3707457

```
// Code instance
@Override
public Object eGet(int featureID, boolean resolve, boolean coreType) {
    switch (featureID) {
        case SimpleExpressionsPackage.COMPARISON__LEFT:
            return getLeft();
        case SimpleExpressionsPackage.COMPARISON__OPERATOR:
            return getOperator();
        case SimpleExpressionsPackage.COMPARISON__RIGHT:
            return getRight();
    }
    return super.eGet(featureID, resolve, coreType);
}

// Comment
<!-- begin-user-doc -->
<!-- end-user-doc -->
@generated
```

Fig. 1. Low quality instance from CodeSearchNet [26].

emerged as a powerful tool to capture, predict, and use such patterns to support software developers in several ways. Specifically, the use of LLMs is becoming increasingly popular for tackling coding tasks [36], i.e., software engineering-related tasks in which the source code is involved. An example of such tasks is *bug fixing*: Given a buggy version of a code snippet (e.g., a method), the task consists of modifying it to remove the issue.

Modern LLMs have been enabled by the introduction of the Transformer architecture [48], which is based on Deep Neural Networks with millions/billions of parameters. Tuning all such parameters requires a huge amount of training data. One of the biggest advantages of Transformers is that they allow for the adoption of *transfer learning*. Their training is divided into steps: The *pre-training* allows to teach the language to the model independently from the task at hand, while a subsequent *fine-tuning* step allows to specialize the model on a downstream task.

Given the size of the data required to both pre-train and fine-tune such models, the datasets adopted are rarely curated and generally collected in a fully automated way. For example, CodeSearchNet [26] is a dataset that contains 6 million functions, with and without documentation, collected from open source code of six different programming languages (i.e., Go, Java, JavaScript, PHP, Python, and Ruby). CodeSearchNet has been adopted to pre-train models very popular in the SE community, such as T5 [36], CodeT5 [51], CodeBERT [19], and CodeT5+ [50]. Let us consider the example in Figure 1. We can observe a pair code-comment present in the CodeSearchNet dataset, in which the comment is clearly inappropriate both to document the code and to represent the ground truth for a code-summarization task.

Previous work provided evidence that shows that datasets used for both pre-training and fine-tuning LLMs contain problematic instances like the previously mentioned one [45, 124]. Training a model on similar instances might be detrimental for two reasons. First, it might introduce issues in the model. Second, in the best scenario, it might be useless and, thus, result in a waste of resources (time and energy). Enhancing data quality by removing and/or updating such instances leads to improved model effectiveness [124]. For this reason, most previous studies that adopted LLMs used several strategies (*quality filters*) to clean up the datasets they adopted. While the use of some quality filters is well-established in the literature (e.g., removal of duplicates), researchers are not always aware of all the less popular options, which are not necessarily less important.

Besides, the quality filters adopted by researchers are only specific solutions to broader problems—what Foidl et al. defined as *data smells* [20]. While catalogs of data smells have been defined in previous work [20, 42], they are generic and they mostly focused on tabular data. To the best of our knowledge, no previous work tried to define a catalog of data smells for coding tasks in the context of LLMs.

In this work, we aim to fill this gap by relying on the literature in the Software Engineering research field. We present a **Systematic Literature Review (SLR)** in which we analyze articles in

which researchers used LLMs to tackle coding tasks. We specifically focused on studies in which researchers used at least a quality filter to improve the quality of the dataset(s) they adopted.

After performing a query on the most relevant digital libraries, we collected a set of ~11k papers, which became 81 after applying filters based on our inclusion and exclusion criteria. Then, through snowballing, we enlarged such a set with 26 additional papers, collecting a total of 107 papers. We manually extracted all the quality filters used in them and derived the root issue that they aimed at solving—i.e., data smells for coding tasks.

The main contribution of our work is a taxonomy of 71 data smells for coding tasks, with the solutions (i.e., quality filters) researchers previously adopted for them. We also performed a more in-depth analysis on the diffusion of the adoption of quality filters in both pre-training and fine-tuning, and in specific downstream coding tasks. We observed that some quality filters (e.g., the ones related to code and comments quality) are more often adopted for fine-tuning than for pre-training, and that they are currently ignored for some tasks (e.g., automated program repair).

2 Background and Related Work

In this section, we present the concept of *data smell* and the recent studies on this topic.

2.1 Data Smells

Data smells have been first introduced by Foidl et al. [20] as data quality issues that may lead to future problems. The authors introduced a catalog of 36 data smells and divided them into three categories: (i) Believability Smells (e.g., the presence of dummy values to represent missing values), (ii) Understandability Smells (e.g., encoding an integer as a string), and (iii) Consistency Smells (e.g., use on inconsistent abbreviations). Recupito et al. [42] extended such a catalog by adding 12 data smells with three additional categories: (i) Redundant Value Smells (e.g., features having a linear relationship), (ii) Distribution Smells (e.g., values different from the distribution), and (iii) Miscellaneous Smells (e.g., features that are likely to introduce bias and unfairness).

Existing catalogs only consider *task-agnostic* data smells, i.e., problems that are not dependent on the specific downstream task on which a machine learning model is trained. In other words, such problems only represent the intersection of the data smells that regards classes of tasks or specific tasks. In this work, we differentiate from Foidl et al. and Recupito et al. by focusing on *task-dependent* data smells. Specifically, we focus on data smells of code-related tasks.

2.2 Transformer-Based Models for Code-Related Tasks

Deep learning methodologies have caused a significant paradigm shift in various research fields. The introduction of the Transformer architecture [48] revolutionized the research and practice in NLP and Software Engineering. Such an architecture overcomes the limits of the RNN architecture through the self-attention mechanism. On the backbone of Transformers, models leveraging *transfer learning* have been introduced [18, 39]. Such a mechanism consists of (i) a *pre-training* phase, in which the model learns through a self-supervised task and (ii) a *fine-tuning* phase, in which the acquired knowledge is leveraged to boost performance on a downstream task through additional training on a task-specific dataset. Pre-Trained Transformer models such as BERT [18], T5 [40], GPT-3 [14], and GPT-4 [9] have achieved state-of-the-art results in many NLP tasks. In addition, models such as CodeBERT [19], GraphCodeBERT [21], CodeT5 [51], and Code Llama [119] have allowed researchers to define state-of-the-art approaches for addressing code-related tasks. An example of code-related task previously addressed in the literature is *code summarization*. Such a task aims consists in generating a natural language description (output) of the given code snippet (input). Generally, the training set is composed of instances in the format $\langle code, description \rangle$, which is fed to

the model that learns in a supervised fashion. Another example of a task is *code completion*, which consists of completing a partial coding solution written by the developer. Given an incomplete code snippet (input), the model generates the missing code (output). Similar to other tasks, training for code completion typically relies on a supervised approach, where the model learns from instances in the format ⟨*incomplete code*, *missing code*⟩ pairs, enabling it to predict (i) the next token, (ii) the next line, or (iii) the next block.

3 SLR Planning

The main *goal* of our SLR is to define the issues that can affect the quality and effectiveness of Transformer-based models for code-related tasks. We specifically focus on Transformer-based models (and, thus, indirectly on LLMs) since they allow to achieve state-of-the-art results in basically all code-related tasks [36, 51, 66, 110, 119, 124, 129].

Our study is steered by the following **Research Questions (RQs)**:

- *RQ1*: *What are the known data smells for code-related tasks and the strategies adopted to remove them?*
- *RQ2*: *How frequent is the use of strategies to remove data smells between the two training steps of Transformers (i.e., pre-training and fine-tuning)?*
- *RQ3*: *How frequent is the use of strategies to remove data smells among different coding task?*

We followed the guidelines provided by Kitchenam et al. [30].

3.1 Study Collection

As a first step, we needed to collect a list of primary studies of interest. To do this, we relied on the following digital libraries: ACM Digital Library [1], IEEE Xplore Digital Library [4], Springer Link Online Library [8], and Scopus [7]. We decided not to use Google Scholar to exclude gray literature, as also done in previous work [16].

Query. To define the query, we initially considered the query used in a recent SLR with a similar aim, i.e., the one by Tufano et al. [130] and the one by Hou et al. [25]. We used the same keywords adopted in the former (i.e., "pre-training" and its variations and "transfer learning"), but we also included "fine-tuning" and its variations: We do this because we aim to select papers that mention at least one of the two steps that characterize Transformer-based models. We used, instead, the same starting year used in the latter (which includes papers published after 2017) because the Transformer architecture was introduced in that year [48]. Finally, similarly to the SLR by Tufano et al. [130], we focused on venues with names related to software engineering (roughly identified, at this stage, through the keywords "software," "program," and "code"). The final query we used to select the first set of studies is the following:

full text CONTAINS (*"pretrain"* OR *"pretrained"* OR *"pretraining"* OR *"pre-train"* OR *"pre-trained"* OR *"pre-training"* OR *"finetune"* OR *"finetuned"* OR *"finetuning"* OR *"fine-tune"* OR *"fine-tuned"* OR *"fine-tuning"* OR *"transfer learning"*) AND
publication date IS FROM 01.01.2017 TO 31.07.2024 AND
publication venue CONTAINS
(*"software"* OR *"program"* OR *"code"*)

3.2 Inclusion and Exclusion Criteria

We defined a set of inclusion and exclusion criteria for our SLR. Such criteria are summarized in Table 2.

Table 1. Papers Returned from Digital Libraries Queries

Source	Papers
ACM Digital Library	2,095
IEEE Xplore Digital Library	3,280
Springer Link Online Library	1,826
Scopus	4,029
Total with duplicates	11,230
Total with only SE papers and without duplicates	2,822

Table 2. Inclusion and Exclusion Criteria

Inclusion Criteria	
IC1	The paper must be published at SE conferences or journals ranked A-A*.
IC2	The paper must present an approach based on Transformer.
IC3	The paper must present approaches to automate a code-related task.
IC4	The paper must describe a methodology for data pre-processing for at least one between pre-training and fine-tuning.
IC5	The paper must precisely describe the pre-processing techniques used.
Exclusion Criteria	
EC1	The paper is made of less than 8 pages.
EC2	The paper is not written in English.
EC3	The paper has not been peer-reviewed.
EC4	The PDF of the paper is not available.

To ensure a high quality set of primary studies, we considered only the ones published in top SE conferences[1] and journals[2], i.e., the ones with CORE ranking A*-A as of 2023[3] (IC1) [53, 130]. Note that the topic under study has been explored also in other research fields (e.g., Artificial Intelligence and Natural Language Processing). However, we only consider SE venues because our focus is on the SE field since we want to focus on code-related datasets. In addition, we decided to exclude papers that have not been subject to a full peer-review process yet (EC1) and the ones shorter than 8 pages (EC3) to collect only high-standard, detailed results that documented extensively their findings [16, 130]. Our main objective is to collect papers in which the authors trained models for tackling *code-related tasks* (IC3) and applied any kind of pre-processing or quality filter to an originally defined or ready-made dataset (IC4, IC5).

3.3 Filtering and Snowballing

Table 1 reports the results obtained from the queries to the digital libraries. We obtained a total of 11,230 results. We excluded duplicates (e.g., present on both IEEE Xplore and ACM DL) and papers not published in the SE venues we selected, leading us to 2,822 papers. Given the large number of papers left, we decided to apply an additional filter before delving into the full-text read. One of the

[1]We considered the following conferences: ASE, EASE, ESEC/FSE, ESEM, ICPC, ICSA, ICSE, ICSME, ICST, ISSRE, ISSTA, MSR, SANER, and SEAMS.
[2]We considered the following journals: EMSE, IST, JSS, TSE, and TOSEM.
[3]http://portal.core.edu.au/

Table 3. Number of Papers Extracted for Primary Studies and Snowballing

Initial Search		Snowballing	
Venue	**Papers**	**Venue**	**Papers**
ICSE	17	arXiv	7
TOSEM	14	ICLR	4
ESEC/FSE	13	EMNLP	3
TSE	9	ACL	2
ASE	9	MAPS	2
ICPC	8	NeurIPS	2
MSR	3	AST	1
JSS	2	NAACL	1
ICSME	2	ICML	1
SANER	2	KDD	1
EASE	1	LREC	1
EMSE	1	TMLR	1
Total	81	Total	26

authors inspected the papers starting from the title and abstract and excluded the ones clearly out of the scope of our SLR. Note that we adopted a conservative approach, i.e., we did not exclude papers unless we were certain. This filter left us with 486 papers. Then, two of the authors inspected the 486 remaining papers by reading the full-text and decided to include them or not based on the inclusion/exclusion criteria. In the end, we selected 81 primary studies as a base for our SLR.

After having read the selected papers, we extracted the references and performed snowballing. In this step, we decided to relax some inclusion and exclusion criteria. First, we ignored IC1 for such a step, i.e., we included also papers not published in top SE venues. We did this because we noticed that some relevant contributions have been published in venues related, for example, to AI and NLP (e.g., ICLR and NeurIPS). Note that we used IC1 in our original search anyway because most of the papers from non-SE venues matching our query were not relevant for our study and, thus, analyzing all of them would have required an enormous manual effort with minimal gain. We also relaxed EC3 and included primary studies published on arXiv. We did this for an analogous reason: Some highly relevant contributions (e.g., Codex [62] and Code Llama [119]) have not been published in peer-reviewed venues. Still, their quality is supported by the citations they received from the paper in which we found them. This resulted in the inclusion of 26 papers. Thus, in total, we considered 107 studies. We report their distribution in terms of venues in Table 3. We show the number of papers by publication year for the 81 studies and for the 26 snowballing papers we analyzed in Figure 2. We did not report the number of occurrences for 2017, 2018, and 2019 since we found no papers from those years.

3.4 Data Extraction

To answer RQ1, we looked for candidate *quality filters*. Specifically, we searched for (i) pre-processing techniques used to clean up the dataset before training the model, and (ii) possible empirical evidence presented in the papers (e.g., studies explicitly aimed at validating the effectiveness of a given filter). Two of the authors independently extracted such information from the primary studies and assigned one or more tags representing the quality filters adopted by each of them. We performed

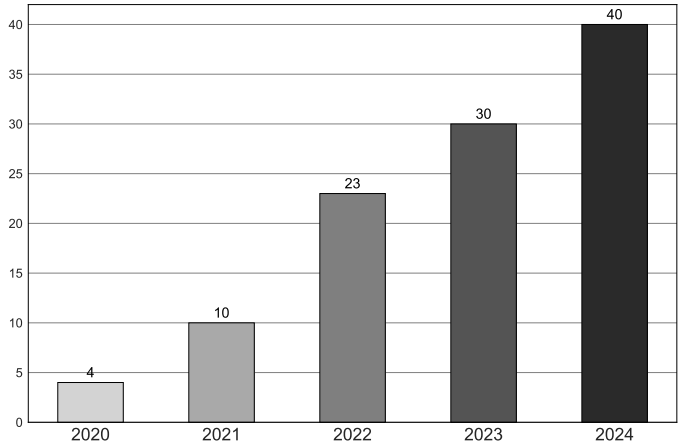

Fig. 2. Number of papers from SLR by year.

a card sorting activity aimed at merging duplicates (i.e., merging similar tags representing the same quality filter). After this step, it could happen that some of the tags assigned by the two authors to a given paper differed (i.e., there were disagreements). When this happened, the authors discussed each disagreement (i.e., tag that one author assigned and the other did not assign) to a given paper by reading once again the paper, aiming at reaching a consensus. There has been no case in which we did not reach a consensus. After such a preliminary step, we qualitatively analyzed the list of quality filters and derived the underlying issue that the authors aimed to solve for each of them. Two of the authors independently derived an underlying data smell for each quality filter identified in the previous step. For example, for the quality filter "duplicates removal," we derived the underlying data smell "presence of exact code clones." Again, we performed a card sorting for merging duplicate data smells and the two authors discussed disagreements aiming at reaching consensus. This happened less frequently than in the initial tagging activity since this step was more straightforward. In the end, we sorted the collected data and defined a taxonomy of data smells.

To answer RQ2, we first identified the datasets on which the quality filters detected in RQ1 have been applied and how each of them has been used in the paper (i.e., as *pre-training* and/or *fine-tuning* quality filter). If Transformer-based models were trained from scratch directly for the target task, we labeled them as "fine-tuning" since no pre-training had been performed in those cases. Two of the authors independently carried out such an analysis and discussed possible disagreements. We used an analogous process to answer RQ3: This time, we focused on extracting the coding task at hand. Given the straightforward nature of such analyses, we registered no disagreements. For both RQs, we plot the prevalence of the adoption of quality filters in terms of phase (RQ2) and task (RQ3). We also report the same data divided by smell type.

We made our replication package publicly available [49]. It includes the list of the selected papers and the labeling we performed.

4 Results

In this section, we report the results and the answers to our RQs.

4.1 RQ1: Data Smells for Code-Related Tasks

In our qualitative analysis, we identified 106 quality filters applied by the paper authors. Starting from them, we extracted 71 data smells for code-related tasks. Note that the number of quality

filters differs from the number of papers analyzed since each study can contain more than a single quality filter, or more papers can share the same one. As a result, we constructed our taxonomy of data smells, which we depict in Figure 3. Such a taxonomy is composed of nine root categories. For each category, we report in the top-right box the number of papers in which the data smell appears. Note that the sum of such numbers is greater than the number of primary studies considered since they can adopt more than a quality filter. It is worth noting that the number of occurrences does not indicate the importance of the data smell, but rather its popularity in the research community.

We also annotate data smells with the ☑ and ✔ symbols to indicate that it has been empirically demonstrated that their presence decreases the effectiveness of the model. The former symbol indicates stronger evidence (e.g., the data smell has been tested alone), while the latter indicates weaker evidence (e.g., the data smell has been tested in conjunction with other data smells). We report in Appendix A the complete mapping between the root categories of our taxonomy and the papers from which they have been extracted.

In the following, we discuss each category.

4.1.1 Limited Informativeness. This category holds the most number of occurrences (89) in the papers we analyzed and it indicates different forms of noise and shortcomings.

Unsurprisingly, the most addressed sub-category is *Noise Tokens* [57, 67, 69, 71, 83, 92, 98, 100, 104, 106, 107, 122, 124, 127–130, 132, 142, 149, 156, 160]. Such a data smell category represents cases in which non-ASCII characters, HTML/XML tags, URLs, file paths, literals, dates, code references, commit hash tokens, and **Personally Identifiable Information (PII)** tokens appear in the instances. Such information is irrelevant to the model and makes the dataset less uniform since their presence may lead the model to focus less on the bigger picture and more on small details. In addition, URLs, file paths, and PII tokens may be long, too specific to the domain context (i.e., project, developer), and result in security and privacy issues [15]. These tokens are found in code, comments, and natural language snippets.

The *Non-Informative Tokens* sub-category [64, 86, 127, 142, 150] occurs when instances contain tokens that do not contribute to model learning. Lack of code structure tokens represents the scarcity of information on the hierarchical structure of code that is not learned by the model. This does not allow the model to learn the code structure information that is crucial for programming languages such as Python (e.g., code indentation). Annotation tokens and notional tokens often receive less attention from models, indicating they contribute minimally to the model's learning. This disproportionate attention results in additional computational costs, highlighting the potential for efficiency improvements by pruning these less impactful tokens [150].

Irrelevant Code [62, 66–68, 71, 83, 90, 92, 94, 97, 106, 111, 112, 124, 130, 135, 138, 151, 156, 159] occurs when trivial, unhelpful, or even problematic code appears in the dataset. First, this is the case of *boilerplate code* (e.g., default getter and setter methods in Java), IDE autogenerated-blocks (i.e., naive summaries), overridden methods (e.g., highly repetitive), obsolete code (e.g., outdated APIs), and deprecated code (i.e., set for upgrade or removal). Out-of-scope code can be harmful as well. For example, out-of-scope code includes the presence of test code in datasets that aim at containing examples of production code and vice versa. Finally, superfluous code occurs when few statements are sufficient to provide useful information to the model, albeit being task-dependent.

We found a lower number of *Inconsistent Conventions* [63, 65, 66, 68, 90, 122, 130, 142, 146, 151, 153] in studies. Such a sub-category of smells occurs when code conventions are not uniform throughout the dataset (e.g., some instances use camel case and others snake case for identifiers). The conflicting signals resulting from using inconsistent conventions may divert the learning of the code logic.

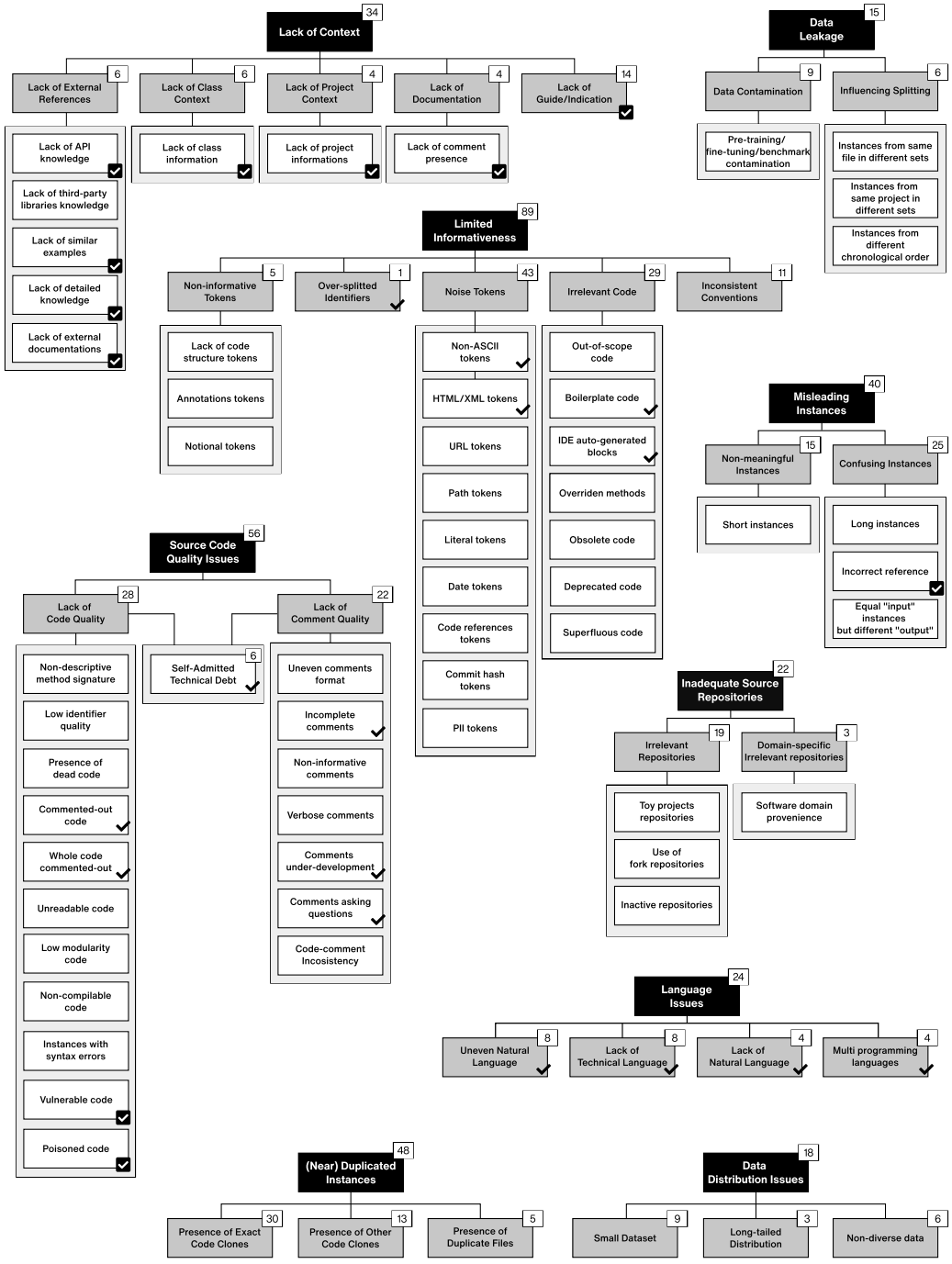

Fig. 3. Data smells of code-related datasets.

Finally, the *Over-Splitted Identifiers* [124] sub-category refers to the splitting operation of instance components in camel case or in snake case. This may misrepresent the original meaning of such a component.

Procedures to Address Limited Informativeness. One of the most common operations that aim to solve such a data smell category is the removal of elements that can infect the instance.

When dealing with *Noise Tokens* such as non-ASCII characters, the common pre-processing operation is related to the removal of such elements [67, 69, 83, 104, 106, 107, 128, 130, 149].

Instead, regarding URLs, paths, literals, dates, code references, commit hash, and PII tokens, a common procedure is to identify and substitute them with a default one that abstracts their real content (i.e., `https://www.d... -> <URL_TOKEN>`, `/home/user/... -> <PATH_TOKEN>`) [57, 71, 98, 100, 104, 122, 127, 129, 130, 132, 142, 149, 156, 160]. Different from the others, PII tokens are harder to detect. For this reason, specific detection models are trained and used to detect them, followed by an abstraction operation with special tokens (e.g., `<NAME>`, `<E-MAIL>`, `<KEY>`, `<PASSWORD>`) [92].

Regarding *Irrelevant Code*, the common approach is to completely remove instances that represent test methods [66–68, 90, 97, 106, 111, 130, 156], boilerplate code [66, 68, 90, 92, 94, 124, 159], overridden methods [90], and obsolete and deprecated code [112]. IDE auto-generated blocks are removed, leaving the code part of the instance intact (if any) [62, 83, 92, 111, 124, 135, 138, 151, 156]. On the other hand, superfluous code has not been explicitly addressed. However, prior study [71] empirically shows that not all statements of a method are needed to train an effective model.

Instances that belong to the *Non-Informative Tokens* sub-category in the form of annotations and notional tokens are removed too. Instead, when there is a lack of structure tokens, these are added where needed to let the model know the instance structure. For example, when dealing with Python code, tokens such as `<INDENT>` and `<DEDENT>` are used [64, 86, 127].

The datasets affected by *Inconsistent Conventions* are generally treated in such a way that all instances follow the same convention. An example may be the change of identifier formatting from snake case to camel case (e.g., `ordered_books -> orderedBooks`) or the same code format applied to all instances (e.g., format to follow pep8 conventions for Python code) [63, 65, 66, 68, 90, 122, 130, 142, 146, 151, 153].

Instances affected by *Over-Splitted Identifiers* are simply replaced with the original ones [124]. For example, if an instance contains in a comment `j Text Field` it is replaced with the original complete identifier `jTextField`.

4.1.2 Data Leakage. We found only 15 occurrences related to *Data Leakage* [76, 77, 82, 87, 92, 94, 97, 102, 103, 123, 141, 148, 156, 158, 159]. This category of smells includes issues regarding (partial) overlaps among datasets. The *Pre-Training/Fine-Tuning/Benchmark Contamination* sub-category is strictly related to the transfer-learning paradigm and to a fair evaluation. Pre-trained models have already been subject to training on a dataset P. Fine-tuning them on a dataset F that shares instances with P (i.e., $F \cap P \neq \emptyset$), or evaluating them on a benchmark (e.g., test set) B with similar instances result in an unfair evaluation of the model's effectiveness. As a result, the measured accuracy might be unnaturally higher, and the resulting model might be poorly generalizable [77, 82, 87, 92, 94, 97, 103, 141, 158].

The *Influencing Splitting* sub-category regards the dataset splitting. When collecting a dataset for code-related tasks, the usual procedure is to mine source code repositories and their histories to extract code snippets (e.g., methods) from files. Therefore, the dataset might contain several code snippets from the same file, and even several versions of them in time. If the split into training, validation, and test sets is done in a completely random way, it is likely that different sets (e.g., both training and test sets) contain code snippets from the same file or even related to the exact same snippet at different times. Thus, similarly to the previous category, this might result to an

unfair evaluation of the model's effectiveness and poor generalizability [123, 148, 159]. However, it is worth noting that if the main scope is to fine-tune a personalized model (i.e., a model fine-tuned for a specific project) the *Influencing Splitting* sub-category may not be harmful [54].

Procedures to Address Data Leakage. When dealing with the *Pre-Training/Fine-Tuning/Benchmark Contamination* sub-category, a typical procedure consists of checking for overlapping instances between the pre-training and the fine-tuning (or benchmark) datasets and making sure that they appear only in one of them [77, 82, 87, 92, 94, 97, 103, 141, 158]. This can happen in different ways. The decontamination process can be applied by removing the exact overlap between pre-training dataset instances and fine-tuning or benchmark instances. On the other hand, a more specific analysis can be made by removing instances that are not exactly equal. For example, instances that share at least a 10-gram (10 consecutive tokens) are removed [82]. Additionally, another procedure consists in using embedding models to identify and remove code snippets with high semantic similarity [141].

Instead, as for the *Influencing Splitting* sub-category [123, 148, 159], the procedure is to make sure that all instances that originate from the same file are present only in one of the three sets and not scattered in them [123]. If there are several code snippets m_i that belong to the same source file f_k, the splitting procedure must ensure that all the m_i belong to the same set (either training, validation, or test). An analogous procedure is applied at the project granularity level. The code snippets m_i that belong to the same project p_j are moved as a group across the three sets (and never scattered among them) [148]. Finally, a different approach is adopted when the dataset contains different revisions in the history of the project of the source code at hand. In this case, instances are scattered in chronological order among the three sets. If there are several revisions r_i, with $i \in [0, N]$, based on the splitting ratios (r_t for the training, r_v for the validation, and r_T for the test sets) the earliest revisions ($r_{\{0,\ldots,Nr_t\}}$) will be placed in the training set, while the latest ones will be placed in the validation ($r_{\{Nr_t,\ldots,(r_t+r_v)\}}$) and test ($r_{\{N(r_t+r_v),\ldots,N\}}$) sets [148].

4.1.3 Lack of Context. Thirty-four occurrences from the studies we considered included quality filters aimed at enhancing the context of the code in the instances we considered. Thus, the lack of context (e.g., class, method, or project information) is considered a possible cause of lower model's effectiveness. Most of the previous works [36, 47, 66, 68, 104, 105, 110, 124, 129, 130] operate at method-level granularity. It means that the training instances are lines of code representing a whole method. However, method-level information might be insufficient for some tasks.

Let us consider the *Code Review* task in the *code-to-code* format with method-level granularity as a running example to illustrate the different types of lack of context. In such a task, given a code snippet (method), the objective is to generate the revised version of the code [129]. In other words, the dataset is composed of pairs $\langle m_b, m_a \rangle \in P$, where m_b is the code to change, m_a is the code after the edit, and P is the project to which the pairs $\langle m_b, m_a \rangle$ belong. The changes required to transform m_b to m_a may require plenty of contextual information.

First, the model might need to know what other methods/variables are available in the same class, and what other classes/methods are available in the same project. If it does not, the model needs to make up method and class names based on the general knowledge it acquired to make some edits. Classes and methods made up by the model might be missing and, even if they do not, they might have different behaviors from the ones assumed by the model. For example, a method might return null in certain conditions while the model assumes it does not. If this happens, the dataset is affected by *Lack of Class Context* [73, 94, 99, 117, 118, 156]—the model does not know how the class behaves—and *Lack of Project Context* [72, 82, 99, 122]—the model is not aware of the other classes/methods available in the project.

The model might not be aware of how the method it aims to change is supposed to work in the first place. This generally happens when no documentation for the method is provided to the model. If there is no documentation, the edit suggested by the model might change its contract (e.g., the method returns `null` while it is not supposed to do it). When this happens, the dataset is affected by *Lack of Documentation* [89, 99, 131, 156]. Besides, the model might need to know which external libraries are adopted: If the project uses `org.apache.commons.csv`[4] for CSV file processing, and the model is unaware of its functionalities, it is likely to generate suboptimal or incorrect code. If such information is lacking, the dataset suffers from *Lack of External References* [100, 116, 121, 152, 155, 160].

Finally, there might be plenty of valid edits that could be done with the input code. Consider, for example, a method that lacks a null check (possible bug) and that is poorly readable. Given such a method, the model might not know what edit it is supposed to do (adding the check or improving the readability). The ground truth (i.e., the actual edit in the dataset) might contain a partial change (e.g., only the null check). In this case, the model would be trained to blindly perform some kinds of operations that are valid in a specific context (the developer was not focusing on improving code readability) without knowing what this context is. Some guidance (e.g., natural language instructions, such as the ones included in the commit message) might help the model understand what properties of the code it is supposed to change. When this happens, the dataset is affected from *Lack of Guide/Indication* [58, 59, 64, 78, 95, 106, 125, 126, 128, 134, 141, 147, 160].

Procedures to Address Lack of Context. To address the *Lack of Class Context* smell, many procedures have been used. One of these is to take into account more lines before and after that code taken into account [94, 117, 118] letting the model exploit the additional information they provide. A more complex approach is to extract the class context information (i.e., class methods) and encode it into vector representations to feed specific model components (e.g., a new encoder) [99]. This, however, requires tweaking the standard model architecture.

A similar procedure is adopted to address the *Lack of Project Context* smell [99]. Additionally, another procedure involves the use of *ad-hoc* tools (e.g., static code analysis tools) that, based on the generated project graph structure, can find cross-file context that is jointly learned with the actual instance (which needs project context) [72]. In the case of extra-class information (i.e., usage of other classes), another process is the simplification of members [122]. For example, reducing it in the following way: `task.assets.completeBtn -> completeBtn`. Such a simplification aims to decrease the degree of dependency on external contexts. Albeit the previous procedures are fit for the fine-tuning dataset, *Lack of Project Context* can be addressed also for the pre-training one. The procedure involves a topological sort of project file dependencies in a way that all dependent files are placed in a specific order, ensuring that the context of each file relies on the files that appear earlier in the sequence [82].

As for the *Lack of External References* smell, different procedures are adopted. One of them (*Lack of API Knowledge*) consists in encoding the API documentation into a vector representation for a specific component of the model (i.e., one of the encoders) [121]. Another one (*Lack of Third-Party Libraries Knowledge*) leverages an information retrieval approach to gather `import` statements, directly informing the model about the libraries in use, enhancing its accuracy [100]. Alternatively, RAG-based solutions [33] are very common. One of them (*Lack of Similar Examples*) aims to augment the context by retrieving similar examples from proxy datasets (i.e., instances that neither appear in the training set nor in the test set) [116]. Similarly, historical instances stored in a local data archive can be retrieved through *ad-hoc* models (e.g., *k*NN-LLM) to augment each instance [160]. Furthermore, other sources of external references can be retrieved, such as

[4] https://github.com/apache/commons-csv

external documentation (*Lack of External Documentations*) [152], and expert information needed for the task at hand (*Lack of Detailed Information*) such as descriptions and examples of the missing knowledge [155].

As for the *Lack of Documentation* smell [89, 99, 131, 156], the papers we analyzed do not propose any concrete solution. However, one of the primary studies we considered [131] provides empirical evidence showing that the presence of comments documenting the code is beneficial for code-related tasks.

Finally, to tackle the *Lack of Guide/Indication* sub-category, the common approach is to augment the code instance with additional information, such as the commit message that documents the change from which the code instance derives [59] and the code review message [95]. In more specific scenarios (e.g., Automated Program Repair), feedback indications are also useful, such as the error type to fix [58, 64]. Another common procedure consists in dividing complex tasks into simpler ones. For example, a model can be explicitly trained to first localize where to apply the change and then, conditioned by such localization (which serves as an indication), perform it [78, 106, 125, 134]. In addition, extracting important statements from the code through *ad-hoc* models can be beneficial [126]. Last, but not least, training a model with generated Chain-of-Thought [52] instances by teacher models (e.g., GPT-4 [9]) allows the model to learn how to provide itself indications while generating code [141].

4.1.4 Source Code Quality Issues. Low-quality code and documentation are known to be detrimental to software maintenance and evolution in general. We found many occurrences (56) reporting that the *Source Code Quality Issues* is a problem also when training and testing models for coding tasks. We categorized the specific smells into categories, *Lack of Code Quality* and *Lack of Comment Quality*, which refer to the presence of quality issues regarding the source code or the documentation, respectively.

As for the *Lack of Code Quality* [56, 67, 69, 70, 79, 82, 84, 91, 96, 101, 123, 124, 130, 136, 142, 143, 145, 149, 150, 153, 159], we identified both *functional* and *non-functional* issues that could affect the source code in the datasets. Instances of code that are not compilable, that contain syntax errors, vulnerable code, and poisoned code (functional issues) may mislead the model not only for those specific instances but also for others. Indeed, the model is partially trained to make those mistakes. A similar effect is given by code instances that contain dead code and commented-out code. Non-descriptive method signatures and low-quality identifiers, as well as unreadable and non-modular code, (non-functional issues), are more subtle problems. Let us consider an instance with the method name `handle()`: It is unclear what the method handles. Thus, the model likely learns less about what the method should achieve. Note that these smells are highly related to the ones regarding the *Lack of Context*. Some class-related information might make the meaning of a generic method name like `handle` clear. As for the identifiers, instances that contain variables named `x` instead of `length`, `qr` instead of `query`, `flag` instead of `isValid` obscure the real meaning and usefulness of such identifiers, reducing the understandability of the code for the model.

The *Lack of Comment Quality* [57, 65, 100, 104, 106, 115, 124, 130, 135, 156, 159] also might be detrimental. Similarly to what happens with human developers, empty comments, short comments that do not contribute to informativeness, too long comments that contain unnecessary details, and inconsistent comments, make the code understanding step more problematic.

One smell is a sign of both poor comment and code quality: *Self-Admitted Technical Debt* [104, 124, 130, 132, 149, 156]. The presence of *Self-Admitted Technical Debt* (i.e., comments reporting `TODO`, `FIXME`, and more complex ones [13]) entails low-quality code (there is technical debt) and comments (the comment reports the technical debt instead of providing interesting information about the code).

Procedures to Address Source Code Quality Issues. Instances that contain syntax errors and that are not compilable are generally removed [67, 69, 82, 84, 96, 123, 130, 145, 159]. The same is true for snippets containing commented-out code [124, 149]. Regarding method name and identifier quality, no approaches have been devised to solve such an issue; however, empirical evidence shows that such smells can negatively impact the model's performance, leading to reduced understanding, lower accuracy, and potential security vulnerabilities [56, 79, 91, 101, 136, 142, 150, 153, 159].

Regarding unreadable code and low modularity code, such smells are addressed by removing the affected instances; however, no specific procedures are documented in the studies we analyzed [82]. Dead code, vulnerable code, and poisoned code have been empirically shown as detrimental [70, 91, 143]. Again, the procedure to address them consists in detecting such instances and removing them. Such detection can be made through activation clustering [44], spectral signature analysis [41, 46], and ONION [38].

Similar strategies are adopted for low-quality comments. One of the strategies consists in removing the whole instances containing comments that are not perceived as quality-worthy [57, 65, 100, 104, 106, 115, 124, 130, 135, 141, 156, 159]. However, it is worth noting that many syntactic checks can be performed with CAT [124] that allows to detect most of the smells we identified. To address code-comment inconsistency, a procedure consists of detecting the affected instances using tools such as DocChecker and removing them [17]. Another procedure consists of removing only the comments, for example when they are too short, too long, or contain special symbols or tags [57, 124].

Similarly to other smells in this category, *Self-Admitted Technical Debt* is tackled by detecting through string-matching possible candidates (i.e., `TODO`, `FIX-ME`) and removing them [104, 124, 130, 132, 149].

4.1.5 Data Distribution Issues. A dataset is affected by *Data Distribution Issues* when the distribution of some code properties (e.g., snippet length) is strongly skewed (*Long-Tailed Distribution*), when the dataset is not big enough (*Small Dataset*), or when dataset suffers from poor diversity (*Non-Diverse Data*). Eighteen occurrences report quality filters or empirical evidence regarding data distribution issues. Actually, data distribution issues are not specific to Transformer-based models for coding tasks: Any machine learning model benefits from a balanced distribution of the data. For example, classifiers benefit from a balanced class distribution, and class imbalance affects the model accuracy on minority classes [22, 27, 28].

The *Long-Tailed Distribution* smell [115, 157, 161] refers to an analogous kind of imbalance: A few classes (in this case, code or change properties) are highly represented, while most of the classes are lowly represented. When trained on a *Long-Tailed Distribution* dataset, the model will perform better on the few common instances (i.e., high-tail) and poorly on infrequent ones (i.e., low-tail). In this context, however, it is very hard to formalize which code properties need to be balanced. The main issue lies in the many properties of the source code that could be measured. Let us consider, again, the *Code Review* task we already mentioned in Section 4.1.3. Let $E = \{e_1, e_2, \ldots, e_n\}$ be the types of edits found in the dataset. Let us assume that the edit e_i (e.g., rename a variable) is one of the most frequent ones, while the edit e_j (e.g., edit a comment) occurs a few times. A model trained with such a dataset will probably generate a higher number and more successful predictions when changes related to e_i than to e_j. This smell affects the generalizability and the robustness of the model. The *Small Dataset* sub-category [60, 80, 93, 107, 109, 114, 120, 144, 147] is self-explanatory: Having a dataset with too few instances naturally reduces the learning capabilities of the model [32]. A small dataset also reduces the exposure of the model to diversity. The *Non-Diverse Data* refers to the lack of variation in the dataset instances, which can significantly impact the model's ability to generalize across multiple scenarios. This smell closely relates to *Long-Tailed Distribution*,

but differently from it, consists of only similar-context instances. Let us take as an example the Code Readability Improvement task, where the improvements the model can perform are multiple (e.g., renaming variables, adding comments, or refactoring complex logic flows). A dataset made for the most part by renaming operations will expose the model to few improvement edits, letting the latter be able to improve code readability only by performing a specific operation (e.g., renaming variables). If this happens, the dataset is affected by *Non-Diverse Data*.

Procedures to Address Data Distribution Issues. A solution to data distribution issues classically adopted for machine learning is to adopt data augmentation techniques. For example, let us consider the Automated Program Repair task: The dataset is composed of pairs ⟨*broken code*, *repaired code*⟩. Whether there is a low quantity of such instances (*Small Dataset*), a possible way to fix the smell is to collect a series of methods assumed to be correct and mutate them to introduce issues [93, 144]. Furthermore, data augmentation techniques can also leverage models to generate synthetic data that can be aggregated to the few already collected to improve the size of the dataset [80, 114, 120, 147]. Another useful operation that aims to solve the *Small Dataset* issue is to leverage a dataset for a code-related task that is similar to the task at hand and perform a phase of supervised pre-training with such a dataset before fine-tuning the model for the target code task [107]. Let us consider, again, the Automated Program Repair task. A possible dataset for the supervised pre-training is the dataset used for the Code Review task, which contains generic code changes, not just the ones aimed at fixing bugs. A supervised pre-training helps the model to gain more knowledge related to code changes and then to leverage the learned features to increase its learning for bug-fixing task.

As for the *Long-Tailed Distribution* [115, 157, 161], a possible procedure consists of forcing the model to focus on rarer instances. This can be achieved by implementing methods like Focal Loss [34], which modifies the loss function to give more weight to rare examples, encouraging the model to learn more from under-represented distributions [157].

Regarding the *Non-Diverse Data* [96, 103, 111, 113, 119, 137], the common procedure is to directly rely on instances conditionally generated by teacher models (e.g., GPT-4 [9]) following specific guidelines. The procedure heavily relies on knowledge distillation process [24] which involves transferring knowledge from a large, pre-trained teacher model to a smaller student model. Given the nature of teacher models to comprehend natural language, it is possible to condition the instances generated by the teacher model with the aim of collecting instances diverse from each other. Examples of such procedures are self-instruct [111, 113, 119], OSS-instruct [137], and Code-Eval-Instruct [103].

4.1.6 (Near) Duplicated Instances. Smells from this category occur when duplicated or near-duplicated instances appear in the dataset. We found 48 occurrences documenting such issues.

First, the *Presence of Exact Code Clones* smell [57, 58, 66, 67, 75, 81, 86–88, 100, 104–110, 115, 118, 121–125, 129, 130, 132, 139, 155, 158] refers to the presence of type-1 clones. If identical instances are present in the dataset, it is possible that, after splitting it into training, validation, and test sets, the clones are spread in the three sets, with the natural consequent issues related to hyper-parameter tuning and model testing. The problem remains even if the duplicates are not spread among the three sets. If they are part of the training set, this redundancy reduces the dataset diversity, which might lead to overfitting [11]. As a result, the model is less generalizable.

Presence of Other Code Clones [56, 66, 68, 82, 92, 93, 101, 102, 119, 135, 137, 146, 156] refers to renamed/parametrized, near-miss, and semantic clones (types 2, 3, and 4) [10, 43]. This smell might result in the *Long-Tailed Distribution* smell because the cloned instances may bias the model towards the highly represented classes. The consequences are analogous to those of *Long-Tailed Distribution*.

Additionally, this smell might inflate the performance results at test time, since the model is trained on similar instances to those present in the test set.

Presence of Duplicate Files [58, 65, 69, 128, 139] is related to the data collection phase. When retrieving data from open source repositories it is very likely to find the same contents in different files (e.g., files from non-tagged fork repositories, or mirrored repositories). The consequences are analogous to those of *Presence of Exact Code Clones*.

Procedures for Addressing (Near) Duplicated Instances. To address both *Presence of Exact Code Clones* and *Presence of Other Code Clones*, the most commonly used methodology consists in simply removing the duplicates. As for the former, a simple string match can be adopted. The latter, instead, requires specialized tools, like the one proposed by Allamanis [11], which also detects and removes non-exact code clones. Similarly, another procedure consists in removing those instances that share a high similarity degree measured with metrics such as BLEU [37].

As for *Presence of File Duplicates* [58, 65, 69, 128, 139], different depth-degree manners exist to address such a smell. The straightforward manner consists of performing simple string matching checks of the file contents [69], while a more detailed analysis consists of comparing the Abstract Syntax Trees [58] or removing files with the same hash [65, 128, 139].

4.1.7 Language Issues. The *Language Issues* category contains smells related to what languages are used (both technical, e.g., the source code, and natural, e.g., the comments) and how they are used. We found 24 occurrences related to such issues.

The *Uneven Natural Language* sub-category [57, 92, 124, 129, 130, 138, 149, 156] refers to the presence of tokens or entire instances in different natural languages from the ones on which the model has been pre-trained. Since many of the models are pre-trained on English [18, 40], uneven language often means that non-English tokens or sentences are used. Even though programming languages are made of English keywords, developers can use any language for identifiers and comments. Since the code-related datasets are mined from public repositories, it is very likely to have tokens in languages other than English. This smell is implicitly related to *Long-Tailed Distribution*, when the imbalanced distribution at hand is the one of the natural languages used.

The *Lack of Technical Language* smell [55, 77, 82, 112, 129, 140, 146, 154] appears when technical language is lacking in the pre-training stage. This is especially important for code-related tasks that include natural language (i.e., code review, code generation) or need domain-specific knowledge (e.g., API libraries). If the model has not learned specific technical terms during pre-training, or it has not been exposed to technical documentation, it may not be able to fully replicate their use with the fine-tuning stage alone. For example, let us consider the Code Review task, in which, besides the source code to modify, the model is fed with a natural language command indicating the action to perform. If the model is given the textual command "*Remove ternary operator to increase code readability*" and has not been sufficiently exposed to the concept of "ternary operator," it will likely not be able to fulfill the request.

The *Lack of Natural Language* smell [58, 74, 119, 128] appears when natural language is lacking in the pre-training stage. This is important for both code-related tasks that include natural language and for tasks that do not. Natural language allows the model to better understand and *transfer* the semantics of variables and method names to source code [128]. During code writing, developers name variables and method names based on the semantic meaning of the natural language vocabulary. A model which is directly pre-trained on code may encounter major difficulties in managing the real meaning the developers primarily assigned to it, likely to be conditioned only by learned patterns.

The *Multi-Programming Language* sub-category [56, 61, 127, 133] has its benefits and drawbacks. When the primary objective is to enhance performance in a low-resource programming language

(e.g., Ruby), utilizing a dataset comprising various programming languages can be advantageous. This is because code structures across different languages often exhibit similarities, with consistent patterns in identifiers and method names [56]. However, the beneficial pattern of shared structure and identifiers across languages does not uniformly apply to all code-related tasks. Incorporating code snippets from multiple programming languages can be a smell. It might lead to sub-optimal outcomes for tasks where specific language features are predominant [133].

Procedures to Address Language Issues. To address the *Uneven Language* sub-category, the most common operation is to remove part of the instance or the whole instance containing non-English terms. The former can be done for isolated cases, e.g., if non-English tokens are used in a few comments, only those comments can be removed without removing the whole instance. Removing the whole instance is preferred when the code is documented in a non-English language in which method names and variables are based on a non-English language. Different procedures are applied to detect non-English text. One procedure consists in verifying if the text can be encoded in ASCII [149], or directly excluding instances that contain non-Latin characters [130]. More precise procedures involve the usage of external libraries aimed at detecting languages [92, 139] such as "langid" [5], "cld3" [6], and "fasttext" [2, 29].

Instead, to address the *Lack of Technical Language* sub-category, a typical procedure consists of collecting instances from platforms that are likely to contain pairs of natural language of technical nature with the related code (i.e., related to code scenarios), such as StackOverflow questions and comments [77, 82, 129], StackExchange [55], issues from an issue tracker, and commit messages [82]. Additionally, on top of the previous sources of technical language, other sources rely on the information related to API such as names, and documentation. Such information is easily extracted by parsing the official API documentation [140, 146].

Regarding the *Lack of Natural Language* sub-category, the procedure involves the collection of natural language instances such as those extracted from books, Wikipedia, and news articles [128]. It has been shown that relying on models already pre-trained on natural language and running an additional pre-training phase on code generally allows to achieve better results [58, 74, 119].

4.1.8 Inadequate Source Repositories. The source code repositories from which the dataset is collected play a crucial role both in terms of quality and realism of the dataset. We found 22 occurrences of *Inadequate Source Repositories*.

The *Irrelevant Repositories* [64, 67, 69, 75, 95, 97, 105, 106, 108, 110, 118, 127, 128, 138, 139, 149, 151] refer to the use of repositories that may contain code samples that are not qualitatively useful. Often, these are referred to as toy projects, i.e., repositories created for experimental purposes or as a means to become familiar with GitHub functionalities. It is likely to collect code that lacks depth and real-world application. Such code may not represent realistic software development scenarios, leading to a dataset that misrepresents the complexity of code-related tasks [105]. Note that the *Use of Fork Repositories* smell might cause also the presence of *(Near) Duplicated Instances*.

Inactive Repositories refers to the use of repositories that are no longer maintained and updated. This may have as a consequence the collection of obsolete and vulnerable code, and consequently teaching the model to generate such poor code.

The *Domain-Specific Irrelevant Repositories* sub-category [66, 68, 75] refers to the collection of code instances from repositories within a different application domain than the one targeted by the model. For example, assume to collect a dataset containing instances in Java to build a code completion model tailored for Android projects: Instances unrelated to the Android domain will likely confuse the model and make it more "general-purpose," which is unintended, in this case.

Procedures to Address Inadequate Source Repositories. To tackle the *Irrelevant Repositories* sub-category, many heuristics have been applied. The first common approach is to filter repositories

based on the number of stars on GitHub [64, 67, 69, 75, 95, 97, 105, 106, 108, 110, 118, 127, 128, 138, 139, 149, 151]. For example, a common procedure is to exclude repositories with less than 10 stars [118]. Some other papers consider a repository "mature" and trustworthy based on the number of contributors and the number of commits: For example, previous work excludes repositories with less than 10 contributors and less than 500 commits [105]. In addition, fork repositories are generally excluded [127]. Regarding inactive repositories, a simple solution consists in excluding repositories without commits in the last few years: For example, previous work excluded repositories in which the latest commit was older than 5 years [69].

To address the *Domain-Specific Irrelevant Repositories* sub-category, an approach involves leveraging repository metadata to filter out-of-domain repositories. Metadata such as labels and topic tags are taken into account to build a classifier or on which to apply heuristics [75].

4.1.9 Misleading Instances. Some instances, above all if in conjunction with other conflicting instances, might be misleading and, thus, confuse the model. We found 40 occurrences related to such issues.

The *Non-Meaningful Instances* sub-category refers to those instances in the dataset that have little impact on the model's learning. This includes trivial instances, e.g., consisting of a low number of tokens that do not provide enough context and contribution to model learning [67, 82, 83, 97, 104, 105, 107, 108, 110, 111, 129, 132, 138, 139, 151].

Instead, the *Confusing Instances* sub-category [62, 65–68, 82, 84, 90, 97, 101, 105–108, 110, 111, 118, 125, 129, 132, 139, 151] refers to instances that might confound the model. *Long Instances* are composed of too many tokens that can exceed (inadvertently) the native model capacity. It is possible to truncate such instances to the maximum number of tokens acceptable from the model. However, this may lead to a crucial loss of information, hindering the model from learning the rightful features. *Incorrect References* refers to input-output pairs in which the input, which serves as context, is incorrect. For example, in the Test Case Generation task, the heuristics used to retrieve the method under test may fail, resulting in instances where the reference for the test to be generated does not match the actual one. This teaches the model to generate test cases that are not related to the given method under test. In addition, given the repetitive nature of code, it is possible that coding tasks in which code is given as input and code is expected as output share the same input, but have different expected output. Let us consider the previously mentioned Code Review task and let us imagine that a code snippet from StackOverflow that is both buggy and unreadable is adopted in two distinct projects, A and B. A developer in project A changed it to improve its readability, while another developer in project B fixed the bug. If both instances are kept, the model is trained to perform two conflicting actions given the input code. In such a context, the dataset introduces ambiguity, changing continuously the "ground-truth" during the training phase and, as a consequence, hindering model learning.

Procedures to Address Problematic Instances. To address *Non-Meaningful Instances*, a typical solution is to filter out instances with less than a given amount of tokens. Such a number may vary (e.g., 3 or 10) [110]. As for *Long Instances*, a possibility is to apply the opposite approach used for *Non-Meaningful Instances*, i.e., remove instances with more than a specific number of tokens. This number may also vary, but it is usually set as the maximum number of tokens that the model can natively handle [66, 68, 90, 97, 105, 107, 110, 129, 132]. When taking into account whole files, those bigger than 1 MB are excluded. Additionally, a more subtle procedure is to exclude files with an average line length greater than 100, and a maximum line length greater than 1,000 [62, 65, 118, 139]. Regarding *Incorrect Reference*, the procedure consists of employing more accurate retrieval of the reference, assuring a better alignment between the input reference and the output. For example, in the context of Assert Statement Generation, "Last Call before Assertion" retrieval

Table 4. Data Smell Distribution per Stages

Data Smell	Pre-Training	Fine-Tuning
Limited Informativeness	18/36	36/97
Data Leakage	6/36	10/97
Lack of Context	9/36	28/97
Data Distribution Issues	3/36	18/97
(Near) Duplicated Instances	22/36	40/97
Source Code Quality Issues	7/36	30/97
Language Issues	13/36	18/97
Inadequate Source Repositories	13/36	16/97
Misleading Instances	15/36	21/97

of the focal method is replaced with seven test-to-code traceability techniques [84]. The procedure used to address instances with the same input but different outputs consists in removing such instances. As a running example, take the instances $\langle I_j, O_j \rangle$, and $\langle I_k, O_k \rangle$ with $I_j = I_k$. In this case, the edit distance is calculated: $d_j = d(I_j, O_j)$, and $d_k = d(I_k, O_k)$, then the pair with highest edit distance is discarded.

4.2 RQ2: Pre-Training and Fine-Tuning Datasets

Most of the times (72.9%, i.e., 97 out of the 133) researchers have applied quality filters to improve the fine-tuning dataset. Of these, 26 times researchers have also used quality filters to improve the pre-training dataset. Finally, only 10 times they have applied them to improve the pre-training datasets only. Note that the number of times (133) does not correspond to the number of papers we studied, since a single paper can contain different quality filters aimed at addressing data smells in both datasets. For example, a paper may address the *(Near) Duplicated Instances* smell both in the pre-training dataset and the fine-tuning one. In this case, we count one for pre-training and one for fine-tuning. This result is most likely linked to the very nature of the models we focused on, which allows to reuse models previously pre-trained (transfer learning). The authors often leverage Pre-Trained models such as T5 [40] and CodeT5 [51], tailoring them through fine-tuning to the downstream task(s).

We report in Table 4 the distribution of the addressed data smell categories identified in RQ1 for both stages. It can be observed that the distribution is generally very similar, with a few exceptions. A relevant percentage of *Source Code Quality Issues* filters have been applied to fine-tuning datasets (31%), while they are less prevalent as pre-training filters (19%).

More specifically, as for the pre-training stage, we observed that *Limited Informativeness* and the *(Near) Duplicated Instances* categories are the most addressed data smells. Furthermore, we find a notable discrepancy in the adoption of quality filters between the two stages, in which more specific operations are applied during the fine-tuning stage. This is likely due to the many properties that could be taken into account to run a subtle analysis. Extracting code properties and filtering instances based on such properties may require much time and reduce the initial pre-training dataset size.

Finally, regarding fine-tuning datasets, we observe that quality filters for all the categories of smells are adopted, albeit in different proportions. The most addressed is the *(Near) Duplicated Instances* category (41%), followed by *Limited Informativeness* (37%), and *Source Code Quality Issues (31%)*. On the other hand, *Inadequate Source Repositories* (16%) and *Data Leakage* (10%) are the less addressed ones. This raises concerns regarding the actual evaluation of the models effectiveness, which may overestimate their performance in a real-world setting.

Table 5. Number of Smells per Task with Total Papers

Smell	Code Sum.	Code Gen.	Prog. Repair	Code Compl.	Code Tran.	Code Review	Bug Fix	Vul. Rep.	Asser. Gen.
Limited Info.	11	10	2	8	1	1	1	0	0
Data Leakage	6	5	0	2	2	1	0	0	0
Lack of Context	5	6	5	2	1	1	2	2	1
Data Distr. Iss.	3	5	4	0	3	2	0	2	0
(Near) Duplic. Inst.	10	12	4	6	2	3	2	2	2
Source Code Quality Issues	14	9	2	2	2	1	2	0	0
Language Issues	9	6	2	3	1	2	2	0	0
Inadequate Src Repos	3	3	1	4	1	1	0	0	2
Misleading Instances	3	7	2	4	2	2	1	0	2
# Papers (Total)	27/107	24/107	13/107	12/107	9/107	7/107	5/107	4/107	3/107

We only report tasks for which we have more than two papers. Note that each paper used one or more filters, thus the sum is not equal to the total in the bottom row.

4.3 RQ3: Code-Related Tasks

The papers we selected in our literature review tackle a total of 27 different code-related tasks aimed at generating code. We report in Table 5, for each task, the percentage of papers using at least a quality filter for tacking each category of smells. We only include the tasks for which we have three or more papers. The last row, instead, reports the number of papers by task. It is worth noting that, for some tasks, the large majority of papers use some categories of quality filters. For example, almost all the papers (8 out of 12) tackling the Code Completion task use quality filters to address *Limited Informativeness* smells. On the other hand, some smells are not taken into account at all for some tasks. An example is given by the *Data Leakage* category, which has never been addressed for the Bug-Fixing task. Similarly, *Data Distribution Issues* has never been taken into account for the Code Completion and Bug-Fixing, which are the two of the common tasks tackled in the papers included in our literature review. This raises the need for further investigations in this direction.

5 Implications and Guidelines

We presented a catalog of 71 data smells of code-related tasks. In this section, we provide implications for future research in this field and discuss guidelines for researchers and practitioners who aim at training and experimenting with LLMs.

5.1 Implications

Since building datasets requires plenty of time, it is common to reuse previously defined ones. However, many datasets currently available might be affected by several data smells. Since building datasets requires plenty of time, it is common to reuse previously defined ones. For example, Shi et al. [124] and Sun et al. [45] found different categories of what we call *data smells* in the CodeSearchNet dataset [26]. Such a dataset has been adopted to pre-train most LLMs for coding tasks and even for fine-tuning them on specific downstream tasks [45, 66, 124]. Furthermore, even datasets used as benchmarks have been found to contain data smells [84], raising serious concerns about the actual evaluation procedures of the models. For this reason, updating existing pre-trained LLMs for coding tasks on cleaned versions of existing datasets (e.g., CodeSearchNet) might have benefits for most future works. Finally, manually curating data smell-free benchmarks (similar to HumanEval [62]) for different tasks can be beneficial for a fair and reliable assessment of models performance.

> 💡 Updating existing pre-trained LLMs for coding tasks on cleaned versions of existing datasets (e.g., CodeSearchNet) might have benefits for most future works.

Table 6. Number of Papers in Which Each Category of Fixing Procedure Appear

Procedure	# Papers
Remove	73 (68.2%)
Add	53 (49.5%)
Edit	30 (28.0%)

Despite the previously mentioned studies on CodeSearchNet providing important evidence on the problem of dataset quality issues, our literature review shows that it is currently unknown to what extent existing datasets are affected by data smells. To the best of our knowledge, many current datasets (e.g., CodeXGLUE [35], GitHub Code dataset [3], The Stack [31]) have not been adequately investigated in terms of data smells.

> It is essential to investigate the diffusion of data smells in widely used datasets.

On the other hand, most papers use quality filters despite their effectiveness is mostly based on conjectures. This shows a big weakness in the literature: We do not know whether and to what extent some quality filters benefit the models and thus how data smells of code-related tasks negatively affect them. Some issues have been collectively assessed (✔) and, thus, we do not know to what extent each specific smell impacts the model. Besides, there may be different operations to address such *data smells*. For example, instead of removing instances with non-English terms, a viable approach may be to automatically translate them. Finally, the focus of existing empirical studies on the effects of smells is only limited to some relevant aspects (e.g., effectiveness), while such issues might impact the other aspects as well (e.g., time needed to train the model, robustness, or security).

> Even when empirical evidence is provided that specific data smells negatively affect the model, much more work is needed to enrich our knowledge on what effects they might have and what benefits it could bring removing them. Future work should aim to empirically analyze how most of the data smells of code-related tasks affect training.

The procedures adopted in the primary studies to address the data smells we presented are many and diverse. To understand how researchers generally handle data smells, we manually classified such solutions into three categories and report the number of papers in which such categories have been adopted in Table 6. Note, again, that the sum is greater than the number of papers we analyzed (107) since a paper could include quality filters belonging to different categories. Most papers simply remove the affected instances (or part of them), which reduces the size of the training set and might cause the loss of precious examples (even unique, in some cases). Even the detection strategies adopted are sometimes very simple. For example, Self-Admitted Technical Debt is typically detected by using keyword matching. For most of such detection strategies, we do not know (i) what is their accuracy in detecting affected instances or datasets, and (ii) whether more accurate strategies might exist.

> Future work should aim at devising more precise strategies for detecting data smells and more advanced procedures for improving the quality of datasets. Such work should rigorously validate both the detection and fixing procedures.

Existing empirical evidence on the effects of data smells regards the effectiveness of the models. In other words, previous work only evaluated the *functional* advantages of removing data smells. On the other hand, removing some smells might greatly affect other *non-functional* aspects. For example, removing affected instances might reduce the training time and power consumption of such models, thus improving their sustainability. On the other hand, reducing or modifying the training instances might negatively affect the robustness of the model.

> Future work should investigate the impact of data smells on non-functional aspects, such as training time, sustainability, robustness, security, and performance.

5.2 Guidelines

The smells we found and reported in our taxonomy are, by nature, contextual and dependent on the coding task at hand. Indeed, any given smell might impact some tasks more than others. In some cases, certain smells might even be completely irrelevant. A clear example is the presence of HTML/XML tokens. While such tokens are detrimental when they appear in some dataset (e.g., in the summaries in code summarization datasets), they need to be kept for other tasks, for which they might even be fundamental. Such tokens, indeed, could be a legitimate part of source code that needs to be generated (e.g., in web applications). Similarly, while having unreadable code is generally undesirable, the Code Readability Improvement task requires its presence in the training set to understand how to transform unreadable code into readable code. For this reason, the smells we provide constitute a super set of problems that might affect a given dataset for a given coding task. Nevertheless, there are a few data smells that could be considered *universal*, i.e., they are relevant for any coding task and practitioners should always remove them. We identified only five of such categories of smells: (i) *(Near) Duplicated Instances*, (ii) *Data Contamination*, (iii) *Equal "Input" Instances but Different "Output,"* (iv) *Lack of Context*, and (v) *Data Distribution Issues*. Practitioners should address the five universal categories of data smells for any task. Besides, they should carefully check what additional categories from our taxonomy are relevant to their task.

After practitioners choose a specific set of data smells of interest, they should try to address them at different steps of the training and evaluation of the model. Given the complexity of such a task, we provide in Table 7 a list of guidelines that practitioners can adopt to make sure they do not incur in the problems presented in this article.

6 Threats to Validity

Threats to Internal Validity. The procedure we used to define our catalog of smells (RQ1) is mostly based on manual analysis. Such an approach might result in the introduction of arbitrary decisions. We tried to limit this threat by ensuring that two of the authors independently inspected each paper and extracted the quality filters used in it. Besides, two of the authors extracted the data smells from each quality filter, again, independently one from the other. Since we adopted a tagging-based approach for extracting the quality filters and data smells, we cannot compute the level of agreement through the metrics generally used in similar contexts (e.g., Cohen's kappa). Indeed, a slight variation in the tags used to represent the same concept would result in a false lack of

Table 7. Guidelines for Practitioners to Avoid Data Smells

Pre-Training	
Pre-trained model	Prefer a model pre-trained on natural language, technical language, and source code (see *Language Issues*).
Additional pre-training	If additional pre-training from scratch or additional pre-training needs to be done, the pre-training dataset should not contain instances affected by the relevant sub-categories of *Language Issues*, *Limited Informativeness*, *Inadequate Source Repositories*, *Source Code Quality Issues*.
Fine-Tuning	
Adequate Repositories	Make sure that the instances come from source code repositories sufficiently good (see *Inadequate Source Repositories*).
Context Adequacy	Make sure that the instances have adequate context (see *Lack of Context*). For example, in the API Recommendation task, it is important to provide the model with sufficient API information.
Source Code Quality	Make sure that the source code has sufficient quality (see *Source Code Quality Issues*).
Non-Informative Instances	Make sure that the source code is not outdated and that there is no noise in the dataset (see *Limited Informativeness*).
Problematic Instances	Make sure that no (near) duplicates appear in the dataset and that the instances are not misleading (see *(Near) Duplicate Instances* and *Misleading Instances*).
Data Distribution	Check whether the dataset is affected by data imbalances or low diversity-related issues (see *Data Distribution Issues*).
Evaluation	
Test Representativeness	Make sure that the test set allows to assess the different subtleties of the real-world instances (see *Data Distribution Issues*).
Data Leakage	Make sure that there is no data contamination between training and test sets and that the chronological and project-related aspects are adequately taken into account (see *Data Leakage*).

agreement (e.g., "remove of duplicates" vs. "duplicate removal"). Despite the differences in the tags used, the authors agreed on all the quality filters extracted. On the other hand, the two authors disagreed on 10 out of 71 data smells extracted from the quality filters. In the end, we successfully resolved such conflicts through open discussion until a consensus was reached. Another threat could be related to the design of the query we used to search for articles in the digital libraries. To limit this threat, we took inspiration from queries used in previous literature reviews on similar topics [25, 130] and slightly adapted them to our purpose. Finally, we did not consider papers published

after August 2024. We acknowledge that several relevant works might have been published in the meanwhile.

Threats to External Validity. In our literature review, we focused on papers published in Software Engineering venues. It is possible, however, that relevant papers have been published in other venues related to Natural Language Processing or Machine Learning/Artificial Intelligence. To limit such a threat, during snowballing, we included several papers published in such venues. Besides, we only focused on the Transformer architecture. We did this because it is, nowadays, the state-of-the-art for almost all the generative models tackling coding tasks. There is a risk that papers that present approaches that rely on other models addressed additional relevant data smells that are not covered in this literature review. Still, several data smells we collected are generalizable to different model architectures. The *(Near) Duplicated Instances*, *Data Leakage*, and *Data Distribution Issues* categories arise from poor experimental design, making them quite independent from the model architecture at hand. The same is true for *Misleading Instances* and *Inadequate Source Repositories* categories which derive from poor data management during the collection phase. Furthermore, also the *Lack of Context* category extends to other model architectures (i.e., RNN) as shown by Bansal et al. [12]. Nonetheless, others are not. The *Data Contamination* sub-category and the *Language Issues* category might not be directly transferable to other model architectures, such as RNNs, that do not incorporate the "pre-train then finetune" paradigm. The same regards the data smells involving comments and identifiers (i.e., *Source Code Quality Issues* and *Limited Informativeness*) when removed or abstracted. For example, Tufano et al. [47] did that to overcome the out-of-vocabulary problem, which RNNs are particularly sensitive to. To wrap up, while some of the data smell categories we collected are generalizable to other model architectures, others are not directly extendible and may require tailored analysis.

7 Conclusions

We presented an SLR aimed at acquiring information on the quality filters used in the literature for LLMs for coding tasks. Our analysis resulted in a definition of a catalog of 71 data smells for coding tasks and the related strategies adopted in the literature to remove them.

Our results call for future work on several aspects. First, it is necessary to establish the impact of data smells (and, thus, of the related quality filters) on several aspects of the training (e.g., efficiency) and of the resulting model (e.g., effectiveness). Second, the diffusion of data smells should be explored on state-of-the-art datasets to understand their impact on the current research and practice. Third, researchers should try to improve LLMs for specific coding tasks by adopting quality filters that are currently neglected for those tasks.

References

[1] ACM Digital Library. 2024. Retrieved from https://dl.acm.org/
[2] FastText. 2022. Retrieved from https://fasttext.cc/docs/en/language-identification.html
[3] GitHub Code Dataset. 2022. Retrieved from https://huggingface.co/datasets/codeparrot/github-code
[4] IEEE Xplore Digital Library. 2024. Retrieved from https://ieeexplore.ieee.org/
[5] Langid.py: Standalone Language Identification (LangID) Tool. 2011. Retrieved from https://github.com/saffsd/langid.py
[6] pycld3: Python3 Bindings for the Compact Language Detector v3 (CLD3). 2004. Retrieved from https://github.com/bsolomon1124/pycld3
[7] Scopus. 2024. Retrieved from https://www.scopus.com/
[8] Springer Link Online Library. 2024. Retrieved from https://link.springer.com/
[9] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. arXiv:2303.08774. Retrieved from https://arxiv.org/abs/2303.08774

[10] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A systematic review on code clone detection. *IEEE Access* 7 (2019), 86121–86144.
[11] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 143–153.
[12] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-level encoding for neural source code summarization of subroutines. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 253–264.
[13] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *13th International Conference on Mining Software Repositories*, 315–326.
[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 1877–1901.
[15] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. 2021. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security '21)*, 2633–2650.
[16] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. 2022. Data preparation for software vulnerability prediction: A systematic literature review. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1044–1063.
[17] Anh T. V. Dau, Nghi D. Q. Bui, and Jin L. C. Guo. 2023. Bootstrapping code-text pretrained language model to detect inconsistency between code and comment. arXiv:2306.06347. Retrieved from https://arxiv.org/abs/2306.06347
[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805. Retrieved from https://arxiv.org/abs/1810.04805
[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. arXiv:2002.08155. Retrieved from https://arxiv.org/abs/2002.08155
[20] Harald Foidl, Michael Felderer, and Rudolf Ramler. 2022. Data smells: Categories, causes and consequences, and detection of suspicious data in AI-based systems. In *1st International Conference on AI Engineering: Software Engineering for AI*, 229–239.
[21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training code representations with data flow. arXiv:2009.08366. Retrieved from https://arxiv.org/abs/2009.08366
[22] Haibo He and Yunqian Ma. 2013. *Imbalanced Learning: Foundations, Algorithms, and Applications*. John Wiley & Sons, Ltd.
[23] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Communications of the ACM* 59, 5 (2016), 122–131.
[24] Geoffrey Hinton. 2015. Distilling the knowledge in a neural network. arXiv:1503.02531. Retrieved from https://arxiv.org/abs/1503.02531
[25] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. arXiv:2308.10620. Retrieved from https://arxiv.org/abs/2308.10620
[26] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv:1909.09436. Retrieved from https://arxiv.org/abs/1909.09436
[27] Nathalie Japkowicz and Shaju Stephen. 2002. The class imbalance problem: A systematic study. *Intelligent Data Analysis* 6, 5 (2002), 429–449.
[28] Justin M. Johnson and Taghi M. Khoshgoftaar. 2019. Survey on deep learning with class imbalance. *Journal of Big Data* 6, 1 (2019), 1–54.
[29] Armand Joulin. 2016. FastText.zip: Compressing text classification models. arXiv:1612.03651. Retrieved from https://arxiv.org/abs/1612.03651
[30] Barbara Kitchenham and Stuart Charters. 2007. Guidelines for performing systematic literature reviews in software engineering.
[31] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The Stack: 3 TB of permissively licensed source code. arXiv:2211.15533. Retrieved from https://arxiv.org/abs/2211.15533
[32] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.
[33] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, 9459–9474.
[34] T. Lin. 2017. Focal loss for dense object detection. arXiv:1708.02002. Retrieved from https://arxiv.org/abs/1708.02002

[35] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664. Retrieved from https://arxiv.org/abs/2102.04664

[36] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[37] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *40th Annual Meeting of the Association for Computational Linguistics*, 311–318.

[38] Fanchao Qi, Yangyi Chen, Mukai Li, Yuan Yao, Zhiyuan Liu, and Maosong Sun. 2020. Onion: A simple and effective defense against textual backdoor attacks. arXiv:2011.10369. Retrieved from https://arxiv.org/abs/2011.10369

[39] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[40] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[41] Goutham Ramakrishnan and Aws Albarghouthi. 2022. Backdoors in neural models of source code. In *2022 26th International Conference on Pattern Recognition (ICPR)*. IEEE, 2892–2899.

[42] Gilberto Recupito, Raimondo Rapacciuolo, Dario Di Nucci, and Fabio Palomba. 2024. Unmasking data secrets: An empirical investigation into data smells and their impact on data quality. In *3rd International Conference on AI Engineering - Software Engineering for AI*, 53–63.

[43] Neha Saini, Sukhdip Singh, and Suman. 2018. Code clones: Detection and management. *Procedia Computer Science* 132 (2018), 718–727.

[44] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security '21)*, 1559–1575.

[45] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *44th International Conference on Software Engineering*, 1609–1620.

[46] Brandon Tran, Jerry Li, and Aleksander Madry. 2018. Spectral signatures in backdoor attacks. In *Advances in Neural Information Processing Systems*, 8011–8021.

[47] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology* 28, 4 (2019), 1–29.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, 6000–6010.

[49] Antonio Vitale, Rocco Oliveto, and Simone Scalabrino. 2024. Replication package of "a catalog of data smells for coding tasks." (2024). DOI: https://doi.org/10.6084/m9.figshare.25898650

[50] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. arXiv:2305.07922. Retrieved from https://arxiv.org/abs/2305.07922

[51] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859. Retrieved from https://arxiv.org/abs/2109.00859

[52] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 24824–24837.

[53] Yan Xiao, Xinyue Zuo, Lei Xue, Kailong Wang, Jin Song Dong, and Ivan Beschastnikh. 2023. Empirical study on transformer-based techniques for software engineering. arXiv:2310.00399. Retrieved from https://arxiv.org/abs/2310.00399

[54] Andrei Zlotchevski, Dawn Drain, Alexey Svyatkovskiy, Colin B. Clement, Neel Sundaresan, and Michele Tufano. 2022. Exploring and evaluating personalized models for code generation. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1500–1508.

[55] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. arXiv:2103.06333. Retrieved from https://arxiv.org/abs/2103.06333

[56] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *44th International Conference on Software Engineering*, 1443–1455.

[57] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. 2023. Extending source code pre-trained language models to summarise decompiled binarie. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 260–271.

[58] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*. PMLR, 780–791.

[59] Saikat Chakraborty and Baishakhi Ray. 2021. On multi-modal learning of editing source code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 443–455.

[60] Binger Chen, Jacek Golebiowski, and Ziawasch Abedjan. 2024. Data augmentation for supervised code translation learning. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 444–456.

[61] Fuxiang Chen, Fatemeh H. Fard, David Lo, and Timofey Bryksin. 2022. On the transferability of pre-trained language models for low-resource programming languages. In *30th IEEE/ACM International Conference on Program Comprehension*, 401–412.

[62] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[63] Zimin Chen, Sen Fang, and Martin Monperrus. 2024. Supersonic: Learning to generate source code optimizations in C/C++. *IEEE Transactions on Software Engineering* 50, 11 (2024), 2849–2864. DOI: https://doi.org/10.1109/TSE.2024.3423769

[64] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. Pyty: Repairing static type errors in Python. In *IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[65] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. 2022. Pangu-Coder: Program synthesis with function-level language modeling. arXiv:2207.11280. Retrieved from https://arxiv.org/abs/2207.11280

[66] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of transformer models for code completion. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4818–4837.

[67] Matteo Ciniselli, Alberto Martin-Lopez, and Gabriele Bavota. 2024. On the generalizability of deep learning-based code completion across programming language versions. In *32nd IEEE/ACM International Conference on Program Comprehension*, 99–111.

[68] Matteo Ciniselli, Luca Pascarella, and Gabriele Bavota. 2022. To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set? In *19th International Conference on Mining Software Repositories*, 167–178.

[69] Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: Multi-mode translation of natural language and Python code with transformers. arXiv:2010.03150. Retrieved from https://arxiv.org/abs/2010.03150

[70] Domenico Cotroneo, Cristina Improta, Pietro Liguori, and Roberto Natella. 2024. Vulnerabilities in AI code generators: Exploring targeted data poisoning attacks. In *32nd IEEE/ACM International Conference on Program Comprehension*, 280–292.

[71] Xi Ding, Rui Peng, Xiangping Chen, Yuan Huang, Jing Bian, and Zibin Zheng. 2024. Do code summarization models process too much information? Function signature may be all that is needed. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), 1–35.

[72] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. arXiv:2212.10007. Retrieved from https://arxiv.org/abs/2212.10007

[73] Zishuo Ding, Yiming Tang, Xiaoyu Cheng, Heng Li, and Weiyi Shang. 2023. LoGenText-plus: Improving neural machine translation based logging texts generation with syntactic templates. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–45.

[74] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. In *5th ACM SIGPLAN International Symposium on Machine Programming*, 1–8.

[75] Luke Dramko, Jeremy Lacomis, Pengcheng Yin, Ed Schwartz, Miltiadis Allamanis, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. 2023. DIRE and its data: Neural decompiled variable renamings with respect to software class. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–34.

[76] Chunrong Fang, Weisong Sun, Yuchen Chen, Xiao Chen, Zhao Wei, Quanjun Zhang, Yudu You, Bin Luo, Yang Liu, and Zhenyu Chen. 2024. Esale: Enhancing code-summary alignment learning for source code summarization. *IEEE Transactions on Software Engineering* 50, 8 (2024), 2077–2095. DOI: https://doi.org/10.1109/TSE.2024.3422274

[77] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. arXiv:2204.05999. Retrieved from https://arxiv.org/abs/2204.05999

[78] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision transformer inspired automated vulnerability repair. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–29.

[79] Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, Jun Sun, David Lo, and Yue Yu. 2023. Two sides of the same coin: Exploiting the impact of identifiers in neural code comprehension. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1933–1945.

[80] Shuzheng Gao, Wenxin Mao, Cuiyun Gao, Li Li, Xing Hu, Xin Xia, and Michael R. Lyu. 2024. Learning in the wild: Towards leveraging unlabeled data for effectively tuning pre-trained code models. In *IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[81] Reza Gharibi, Mohammad Hadi Sadreddini, and Seyed Mostafa Fakhrahmad. 2024. T5APR: Empowering automated program repair across languages through checkpoint ensemble. *Journal of Systems and Software* 214 (2024), 112083.

[82] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y. K. Li, et al. 2024. DeepSeek-Coder: When the large language model meets programming–The rise of code intelligence. arXiv:2401.14196. Retrieved from https://arxiv.org/abs/2401.14196

[83] Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md Mahim Anjum Haque, Tahmid Hasan, Wasi Uddin Ahmad, Anindya Iqbal, and Rifat Shahriyar. 2021. CoDesc: A large code-description parallel dataset. arXiv:2105.14220. Retrieved from https://arxiv.org/abs/2105.14220

[84] Yibo He, Jiaming Huang, Hao Yu, and Tao Xie. 2024. An empirical study on focal methods in deep-learning-based approaches for assertion generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1750–1771.

[85] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A deep dive into large language models for automated bug localization and repair. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1471–1493.

[86] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-token code completion by jointly learning from structure and naming sequences. In *44th International Conference on Software Engineering*, 401–412.

[87] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language models for code completion: A practical evaluation. In *IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[88] Kevin Jesse, Premkumar T. Devanbu, and Anand Sawant. 2022. Learning to predict user-defined types. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1508–1522.

[89] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. arXiv:2006.03511. Retrieved from https://arxiv.org/abs/2006.03511

[90] Jingxuan Li, Rui Huang, Wei Li, Kai Yao, and Weiguo Tan. 2021. Toward less hidden cost of code completion with acceptance and ranking models. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 195–205.

[91] Jia Li, Zhuo Li, HuangZhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2024. Poison attack and poison detection on deep source code processing models. *ACM Transactions on Software Engineering and Methodology* 33, 3 (March 2024), 1–31. DOI: https://doi.org/10.1145/3630008

[92] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: May the source be with you! arXiv:2305.06161. Retrieved from https://arxiv.org/abs/2305.06161

[93] Xueyang Li, Shangqing Liu, Ruitao Feng, Guozhu Meng, Xiaofei Xie, Kai Chen, and Yang Liu. 2022. TransRepair: Context-aware program repair for compilation errors. In *37th IEEE/ACM International Conference on Automated Software Engineering*, 1–13.

[94] Zhihao Li, Chuanyi Li, Ze Tang, Wanhong Huang, Jidong Ge, Bin Luo, Vincent Ng, Ting Wang, Yucheng Hu, and Xiaopeng Zhang. 2023. PTM-APIRec: Leveraging pre-trained models of source code in API recommendation. *ACM Transactions on Software Engineering and Methodology* 33 (3) (2023), 1–30.

[95] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. Automating code review activities by large-scale pre-training. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1035–1047.

[96] Zongjie Li, Chaozheng Wang, Pingchuan Ma, Chaowei Liu, Shuai Wang, Daoyuan Wu, Cuiyun Gao, and Yang Liu. 2024. On extracting specialized code abilities from large language models: A feasibility study. In *IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[97] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. CCT5: A code-change-oriented pre-trained model. arXiv:2305.10785. Retrieved from https://arxiv.org/abs/2305.10785

[98] Fang Liu, Zhiyi Fu, Ge Li, Zhi Jin, Hui Liu, Yiyang Hao, and Li Zhang. 2024. Non-autoregressive line-level code completion. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–34.

[99] Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to recommend method names with global context. In *44th International Conference on Software Engineering*, 1294–1306.

[100] Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. CodeGen4Libs: A two-stage approach for library-oriented code generation. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 434–445.

[101] Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. 2024. On the reliability and explainability of language models for program generation. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–26.

[102] Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, Patanamon Thongtanunam, and Li Li. 2022. Automatically recommend code updates: Are we there yet? *ACM Transactions on Software Engineering and Methodology* 33, 8 (2022), 1–27.

[103] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering code large language models with Evol-Instruct. arXiv:2306.08568. Retrieved from https://arxiv.org/abs/2306.08568

[104] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. 2021. An empirical study on code comment completion. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 159–170.

[105] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. 2023. Automated variable renaming: Are we there yet? *Empirical Software Engineering* 28, 2 (2023), 45.

[106] Antonio Mastropaolo, Matteo Ciniselli, Luca Pascarella, Rosalia Tufano, Emad Aghajani, and Gabriele Bavota. 2024. Towards summarizing code snippets using pre-trained transformers. In *32nd IEEE/ACM International Conference on Program Comprehension*, 1–12.

[107] Antonio Mastropaolo, Massimiliano Di Penta, and Gabriele Bavota. 2023. Towards automatically addressing self-admitted technical debt: How far are we? In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–597.

[108] Antonio Mastropaolo, Valentina Ferrari, Luca Pascarella, and Gabriele Bavota. 2024. Log statements generation via deep learning: Widening the support provided to developers. *Journal of Systems and Software* 210 (2024), 111947.

[109] Antonio Mastropaolo, Vittoria Nardone, Gabriele Bavota, and Massimiliano Di Penta. 2024. How the training procedure impacts the performance of deep learning-based vulnerability patching. arXiv:2404.17896. Retrieved from https://arxiv.org/abs/2404.17896

[110] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using deep learning to generate complete log statements. In *44th International Conference on Software Engineering*, 2279–2290.

[111] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. OctoPack: Instruction tuning code large language models. arXiv:2308.07124. Retrieved from https://arxiv.org/abs/2308.07124

[112] Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, Nachiappan Nagappan, and Peter C. Rigby. 2024. AI-assisted code authoring at scale: Fine-tuning, deploying, and mixed methods evaluation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1066–1085.

[113] Mona Nashaat, and James Miller. 2024. Towards efficient fine-tuning of language models with organizational data for automated software review. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2240–2253. DOI: https://doi.org/10.1109/TSE.2024.3428324

[114] Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. 2024. VGX: Large-scale sample generation for boosting learning-based software vulnerability analyses. In *IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[115] Xinglu Pan, Chenxiao Liu, Yanzhen Zou, Tao Xie, and Bing Xie. 2024. MESIA: Understanding and leveraging supplementary nature of method-level comments for automatic comment generation. In *32nd IEEE/ACM International Conference on Program Comprehension*, 74–86.

[116] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. arXiv:2108.11601. Retrieved from https://arxiv.org/abs/2108.11601

[117] Julian Aron Prenner and Romain Robbes. 2024. Out of context: How important is local context in neural program repair? In *IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[118] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. 2023. CAT-LM training language models on aligned code and tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 409–420.

[119] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code Llama: Open foundation models for code. arXiv:2308.12950. Retrieved from https://arxiv.org/abs/2308.12950

[120] Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. arXiv:2110.06773. Retrieved from https://arxiv.org/abs/2110.06773

[121] Ramin Shahbazi, Rishab Sharma, and Fatemeh H. Fard. 2021. API2Com: On the improvement of automatically generated code comments using API documentations. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 411–421.

[122] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating domain knowledge through task augmentation for front-end JavaScript code generation. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1533–1543.

[123] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *44th International Conference on Software Engineering*, 1597–1608.

[124] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? On the importance of data preprocessing for code summarization. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 107–119.

[125] André Silva, Sen Fang, and Martin Monperrus. 2023. RepairLLaMA: Efficient representations and fine-tuned adapters for program repair. arXiv:2312.15698. Retrieved from https://arxiv.org/abs/2312.15698

[126] Weisong Sun, Chunrong Fang, Yuchen Chen, Quanjun Zhang, Guanhong Tao, Yudu You, Tingxu Han, Yifei Ge, Yuling Hu, Bin Luo, et al. 2024. An extractive-and-abstractive framework for source code summarization. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–39.

[127] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1433–1443.

[128] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In *3rd ACM/IEEE International Conference on Automation of Software Test*, 54–64.

[129] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *44th International Conference on Software Engineering*, 2291–2302.

[130] Rosalia Tufano, Luca Pascarella, and Gabriele Bavota. 2023. Automating code-related tasks through transformers: The impact of pre-training. arXiv:2302.04048. Retrieved from https://arxiv.org/abs/2302.04048

[131] Tim van Dam, Maliheh Izadi, and Arie van Deursen. 2023. Enriching source code with contextual data for code completion models: An empirical study. arXiv:2304.12269. Retrieved from https://arxiv.org/abs/2304.12269

[132] Antonio Vitale, Valentina Piantadosi, Simone Scalabrino, and Rocco Oliveto. 2023. Using deep learning to automatically improve code readability. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 573–584.

[133] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One adapter for all programming languages? Adapter tuning for code search and summarization. arXiv:2303.15822. Retrieved from https://arxiv.org/abs/2303.15822

[134] Xu Wang, Hongwei Yu, Xiangxin Meng, Hongliang Cao, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2024. MTL-TRANSFER: Leveraging multi-task learning and transferred knowledge for improving fault localization and program repair. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), 1–31.

[135] Yanlin Wang, Yanxian Huang, Daya Guo, Hongyu Zhang, and Zibin Zheng. 2024. SparseCoder: Identifier-aware sparse transformer for file-level code summarization. arXiv:2401.14727. Retrieved from https://arxiv.org/abs/2401.14727

[136] Yan Wang, Xiaoning Li, Tien N. Nguyen, Shaohua Wang, Chao Ni, and Ling Ding. 2024. Natural is the best: Model-agnostic code simplification for pre-trained large language models. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 586–608.

[137] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. arXiv:2312.02120. Retrieved from https://arxiv.org/abs/2312.02120

[138] Rui Xie, Tianxiang Hu, Wei Ye, and Shikun Zhang. 2022. Low-resources project-specific code summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering*, 1–12.

[139] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10.

[140] Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. arXiv:2004.09015. Retrieved from https://arxiv.org/abs/2004.09015

[141] Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, TerryYue Zhuo, and Taolue Chen. 2024. Chain-of-thought in neural code generation: From and for lightweight language models. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2437–2457. DOI: https://doi.org/10.1109/TSE.2024.3440503

[142] Guang Yang, Yu Zhou, Wenhua Yang, Tao Yue, Xiang Chen, and Taolue Chen. 2022. How important are good method names in neural code generation? A model robustness perspective. arXiv:2211.15844. Retrieved from https://arxiv.org/abs/2211.15844

[143] Zhou Yang, Bowen Xu, JieM Zhang, HongJin Kang, Jieke Shi, Junda He, and David Lo. 2024. Stealthy Backdoor Attack for Code Models. *IEEE Transactions on Software Engineering* 50, 4 (2024), 721–741. DOI: https://doi.org/10.1109/TSE.2024.3361661

[144] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised program repair with test execution diagnostics. In *37th IEEE/ACM International Conference on Automated Software Engineering*, 1–13.

[145] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *44th International Conference on Software Engineering*, 1506–1518.

[146] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When language model meets private library. arXiv:2210.17236. Retrieved from https://arxiv.org/abs/2210.17236

[147] Daoguang Zan, Ailun Yu, Bo Shen, Bei Chen, Wei Li, Yongshun Gong, Xiaolin Chen, Yafen Yao, Weihua Luo, Bei Guan, et al. 2024. DiffCoder: Enhancing large language model on API invocation via analogical code exercises. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 406–426.

[148] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual code co-evolution using large language models. In *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 695–707.

[149] Xiaowei Zhang, Weiqin Zou, Lin Chen, Yanhui Li, and Yuming Zhou. 2022. Towards the analysis and completion of syntactic structure ellipsis for inline comments. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2285–2302.

[150] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1073–1084.

[151] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 5673–5684.

[152] Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2022. DocPrompting: Generating code by retrieving the docs. arXiv:2207.05987. Retrieved from https://arxiv.org/abs/2207.05987

[153] Shasha Zhou, Mingyu Huang, Yanan Sun, and Ke Li. 2024. Evolutionary multi-objective optimization for contextual adversarial example generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2285–2308.

[154] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, Junda He, and David Lo. 2023. Generation-based code review automation: How far are we? arXiv:2303.07221. Retrieved from https://arxiv.org/abs/2303.07221

[155] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[156] Ziyi Zhou, Mingchen Li, Huiqun Yu, Guisheng Fan, Penghui Yang, and Zijie Huang. 2024. Learning to generate structured code summaries from hybrid code context. *IEEE Transactions on Software Engineering* 50, 10 (2024), 2512–2528. DOI: https://doi.org/10.1109/TSE.2024.3439562

[157] Xin Zhout, Kisub Kim, Bowen Xu, Jiakun Liu, DongGyun Han, and David Lo. 2023. The devil is in the tails: How long-tailed code distributions impact large language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 40–52.

[158] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022. XLCoST: A benchmark dataset for cross-lingual code intelligence. arXiv:2206.08474. Retrieved from https://arxiv.org/abs/2206.08474

[159] Tingwei Zhu, Zhong Li, Minxue Pan, Chaoxuan Shi, Tian Zhang, Yu Pei, and Xuandong Li. 2023. Deep is better? An empirical comparison of information retrieval and deep learning approaches to code summarization. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2023), 1–37.

[160] Tingwei Zhu, Zhongxin Liu, Tongtong Xu, Ze Tang, Tian Zhang, Minxue Pan, and Xin Xia. 2024. Exploring and improving code completion for test code. In *32nd IEEE/ACM International Conference on Program Comprehension*, 137–148.

[161] Armin Zirak and Hadi Hemmati. 2024. Improving automated program repair with domain adaptation. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–43.

Appendix

A Data Smell Categories Index

We report in Table A1 the complete mapping between the root categories of our taxonomy and the papers in which they have been tackled.

Table A1. Data Smell Categories Indexing

Dataset Smell Category	Description	References
Limited Informativeness	Instances containing irrelevant or non-contributive elements likely to hinder model learning.	[57, 62–69, 71, 83, 86, 90, 92, 94, 97, 98, 100, 104, 106, 107, 111, 112, 122, 124, 127–130, 132, 135, 138, 142, 146, 149–151, 153, 156, 159, 160]
Data Leakage	Overlap between training and evaluation sets likely to mislead a correct assessment.	[76, 77, 82, 87, 92, 94, 97, 102, 103, 123, 141, 148, 156, 158, 159]
Lack of Context	Insufficient contextual information likely to limit model capabilities.	[58, 59, 64, 72, 73, 78, 82, 85, 89, 94, 95, 99, 100, 106, 116–118, 121, 122, 125, 126, 128, 131, 134, 141, 147, 152, 155, 156, 160]
Data Distribution Issues	Imbalanced distributions likely to limit model generalizability.	[60, 80, 93, 96, 103, 107, 109, 111, 113–115, 119, 120, 137, 144, 147, 157, 161]
(Near) Duplicated Instances	Near or exact duplicates likely to lead to overfitting and poor generalizability.	[56–58, 65–69, 75, 81, 82, 86–88, 92, 93, 100–102, 104–110, 115, 118, 119, 121–125, 128–130, 132, 135, 137, 139, 146, 155, 156, 158]
Source Code Quality Issues	Functional and non-functional code issues likely to hinder model understanding.	[56, 57, 65, 67, 69, 70, 79, 82, 84, 91, 96, 100, 101, 104, 106, 115, 123, 124, 130, 132, 135, 136, 141–143, 145, 149, 150, 153, 156, 159]
Language Issues	Natural and technical language misalignments likely to limit model transferability.	[55–58, 61, 74, 77, 82, 92, 112, 119, 124, 127–130, 133, 138, 140, 146, 149, 154, 156]
Inadequate Source Repositories	Data collection from inappropriate repositories likely to reduce model effectiveness.	[64, 66–69, 75, 95, 97, 105, 106, 108, 110, 118, 127, 128, 138, 139, 149, 151]
Misleading Instances	Confusing or incorrect signals likely to hinder model learning.	[62, 65–68, 82–84, 90, 97, 101, 104–108, 110, 111, 118, 125, 129, 132, 138, 139, 151]

Received 24 May 2024; revised 20 November 2024; accepted 22 November 2024