CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Code smells in machine learning pipelines: an MSR sample study

Bachelor of Science Thesis in Software Engineering and Management

Johann Henri Tammen

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022

The Author grants to University of Gothenburg and Chalmers University of Technology the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let University of Gothenburg and Chalmers University of Technology store the Work electronically and make it accessible on the Internet.

Investigating technical debt in form of code smells in machine learning data pipelines

© Johann Henri Tammen, June, 2022.

Supervisor: Daniel Strüber
Examiner: Richard Berntsson Svensson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
University of Gothenburg
Chalmers University of Technology
Gothenburg, Sweden 2022

Code smells in machine learning pipelines: an MSR sample study

Johann Henri Tammen
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
gustammjo@student.gu.se

***Abstract*—As technical debt in software engineering projects continues to negatively impact the development process, this study focuses on technical debt in form of code smells in machine learning pipelines and in code written by data scientists. This study contributes to the body of knowledge on technical debt as it tries to quantify the assumption in the literature that scientists without a software engineering background struggle with software engineering's best practices when writing code. Furthermore, as machine learning continues to evolve in software engineering, it makes sense to minimize technical debt in machine learning pipelines. Therefore, the source code from repositories in the version control system GitHub was analyzed. The results show that indeed data scientists produce more code smells than software engineers. In addition, the study fails to demonstrate that data pipelines yield more code smells than non-data pipelines.**

***Index Terms*—technical debt, code smells, machine learning, data science**

I. Introduction

With the rise of machine learning (ML) in software applications comes an inevitable increase of code related to ML. While the main challenge of software engineering is to produce clean code for applications, ML focuses on the data that builds the basis for machine learning models [1]. The process of transforming raw data into a form that can be used by ML is called the data pipeline. The individual steps that form the data pipeline consist of methods and code that clean, validate and shape the data into the desired form. As Munappy et al. [2] claim, the data pipeline is a highly automated process and can thereafter be regarded as a dedicated piece of software. Hence, the question arises if the code that forms the data pipelines for ML is subject to common anti-patterns or code smells.

Code smells are sections of code that violate common coding best practices. In their book, Fowler et al. [3] gathered a list of typical code smells that should be refactored. It is important to mention that such violations are not directly considered software defects. In many cases, sections with code smells still fulfill the desired functionality but can affect software development efforts badly at a later point in time [4]. Examples of code smells are large classes, also referred to as god classes or blob classes, long methods, and methods with a long parameter list [3]. The main disadvantages of smells in source code concern readability and maintainability as will be shown in the related work section.

Thereafter, code smells and pattern violation become a form of technical debt, that can potentially cost companies a lot of resources to fix [4]. Despite the negative impacts, practitioners still fall for code smells. Literature often lists time pressure from tight project deadlines as the main reason for practitioners to violate best practices [4]. Other reasons for code smells are a lack of awareness of implementing them, improper knowledge transfer to other programming languages, and committing preliminary code [5].

Over time, many methods to identify code smells have been developed [6], [7] of which many use artificial intelligence [8]–[10]. However, very little research has been done about code smells in machine learning code. Investigating code smells in machine learning pipelines is especially interesting to look at as the people responsible for the pipelines are often data scientists and not software engineers [11]. In which case the experiences by Wilson [12] show that scientists without a software engineering background often struggle with software engineering's best practices. This observation gets supported by Falessi and Kazman [5], who state that a major reason for code smells in source code is that the developers did not know that a better solution is available. By investigating code smells in machine learning code, this study will be valuable for future research in machine learning and technical debt to further improve code quality in machine learning data pipelines. For example, this study can help to identify common code smells in machine learning pipelines with potential contextual explanations of how they arise. Researchers and tool builders who want to develop ML domain-specific analyzing tools can benefit from this knowledge by providing specialized refactoring support, tailored to machine learning code.

In essence, the objective of this study is to investigate code smells in the specific domain of data pipelines for machine learning algorithms. This study will focus on open source repositories on GitHub. A series of filters will be set up to find repositories with machine learning data pipelines and code sections written by data scientists. Additionally, a set of similar code sections that do not include ML pipelines and code sections written by software engineers will be selected for comparison. To investigate code smells in ML pipelines, code analysis tools will be applied to the identified code sections, and statistical tests will be performed for comparison.

II. Related Work

One instance that aggregates best practices for object-oriented code design is the book by Booch [13] first published in 1994. It gives guidance to software engineers about principles to keep in mind to write good code according to Booch. Most of the suggested principles were designed for an object-oriented paradigm. Indeed, many of the principles apply to other programming paradigms as well. Examples of such principles that can be transferred regard code complexity and modularity. Violations of the best practices introduced by Booch can be called code smells. In 1999 Fowler [3] published a seminal book regarding these code smells with a list of common violations of best practices. The book contains clear suggestions for specific code sections that violate best practices and should therefore be refactored to increase the modifiability and readability of the code.

Many studies have shown that code smells can have a negative impact on software practices. One factor is that code smells contribute negatively to the maintainability of software. For example changes in large classes are more likely to break the code, compared to smaller, individual classes [14]. Additionally, the study by Falessi and Voegele [15] shows that sections with a high density of best practice violations are more prone to faults introduced by changes. Another negative impact of code smells is that they make the code harder to understand [16]. As pointed out by Brown et al. [4], this kind of technical debt can cost organizations substantial resources to fix. With low maintainability in code, developers introduce more bugs that need to be fixed and with low readability, developers simply need more time to understand the code to make changes which results in slower progress.

Code quality is already a challenge for software engineers, who, via their training, are aware of these issues, but are sometimes required to make compromises due to time pressure. But the situation might be even worse for programmers without dedicated software engineering training. As described by Wilson [12], not only software developers write code nowadays. In his experience, scientists from physics, life science, or mechanical and civil engineering are involved in writing code. Wilson claims that scientists from other domains than software engineering often struggle with basic software engineering practices like version control, unit tests, or debugging. Moreover, he states that many scientists also lack the knowledge of understanding code quality principles which can lead to meaningless variable names and poor software modularity. Both meaningless variable names and poor software modularity are both typical code smells [3]. To help such scientists with software engineering practices, Wilson [12] taught software carpeting courses and seminars. Furthermore, Wilson et al. [17] proposed a set of practices for all kinds of scientists for research computing, including basic software engineering practices. With this experienced lack of knowledge in the software domain by scientists, one can come to the assumption that the code produced by scientists other than software engineers is also not conforming to best practices. This is in line with the observation by Falessi and Kazman [5]. However, this assumption in the literature is based on personal experiences and assumptions.

Code smells are still an actively studied topic in software engineering research, in particular in conferences such as Technical Debt (co-located with ICSE, first edition in 2018). The combination of code smells and machine learning has so far mostly been considered in the opposite direction, of using machine learning to detect code smells. The results of the study by Patnaik and Padhy [18] show that the random forest classifier has the best performance to identify code smells. MacDonald's research [19] supports this claim. Recent literature, like the study by Pigazzini et al. [20], also focuses on detecting code smells in a specific context. That particular study engages in detecting code smells for microservices. After mapping Wilson's experiences with recent literature, it becomes evident that there is a lack of quantifiable data about code smells in data pipelines. Hence, this study will fill that gap in the literature by quantifying Wilson's claims in the context of data pipelines with open source repositories.

III. Research Methodology

This study tries to quantify the underlying assumption of Wilson's research [12], [17] that scientists without a software engineering background are not familiar with software engineering's best practices in terms of code smells. Thereafter, the goal of this study according to Goal-Question-Metric as described by Basili et al. [21] is the following:
Analyse source code
For the purpose of comparing code smells
With respect to the role of the author of the code and the type of code (ML data pipeline, not ML data pipeline)
From the point of view of researchers
In the context of an mining software repositories sample study run with open source projects on GitHub.

A. Definitions

For the context of this study, I analyze open-source code from data scientists and software engineers gathered in the version control system GitHub. I consider profiles for this study to belong to either population if the profile states "data scientist" or "software engineer" in the biography respectively. The resulting validity threats are discussed in section VI.

B. Research questions and hypotheses

Thereafter, this study suggests the following research questions:

RQ1: How does the code quality from data scientists compare to code quality from software engineers in terms of code smells?

Research question one is directly related to the research by Wilson [12], [17]. The goal of RQ1 is to quantify Wilson's experiences that code produced by scientists other than software engineers compares worse than code produced by software engineers in terms of code quality. Since code quality has many attributes, the focus of this study is code smells. I

consider code smells as anti-patterns in the code, similar to the list of code smells described by Fowler et al. [3] and broken coding conventions like bad indentations or variable names that do not follow conventions. Furthermore, I also consider other anomalies in the source code like unused import statements and depreciated methods and classes as code smells. A full list of the individual anti patterns can be found in [22].

RQ2: How does the code quality of data pipelines compare to non-data pipeline code in terms of code smells?

Similar to the definition of code smells for this study as stated for RQ1, RQ2 also analyzes source code in terms of code smells. Albeit, the context of analyzed code is different. RQ2 focuses on code smells in machine learning pipelines compared to none machine learning pipelines. As code smells proof to negatively impact the development process [4], [14]–[16], and with the gap in the literature that investigates code smells in machine learning pipelines, I deem it worthwhile to understand if code smells are a factor in machine learning pipelines as well.

According to the proposed research questions the study states the following hypotheses regarding code smells (CS):

H1 states that: *Code produced by software engineers yields fewer code smells per file than code produced by data scientists.*

$$H1_A : \frac{CS(software\ engineers)}{files} < \frac{CS(data\ scientists)}{files}$$

and the corresponding null hypothesis is:

$$H1_0 : \frac{CS(software\ engineers)}{files} \geq \frac{CS(data\ scientists)}{files}$$

By comparing the code smells per file for software engineers and data scientists, I can quantify the code smells for both groups. This helps to understand which population is more prone to produce code smells, which in turn helps to answer RQ1.

H2 states that: *Data pipelines yield more code smells per file compared to ordinary code sections.*

$$H2_A : \frac{CS(data\ pipeline)}{files} > \frac{CS(not\ data\ pipeline)}{files}$$

and the corresponding null hypothesis is:

$$H2_0 : \frac{CS(data\ pipeline)}{files} \leq \frac{CS(not\ data\ pipeline)}{files}$$

Similar to hypothesis one, the second hypothesis compares code smells in data pipelines to none data pipelines. With this comparison, I can evaluate to which degree code smells appear in data pipelines to answer RQ2.

C. *Used Research Methodology*

Gathering the required data via an mining software repositories (MSR) sample study in GitHub seems appropriate as the version control system GitHub is the most commonly used tool to store code online currently with more than 287 million repositories [23]. Hence, mining repositories there offers state-of-the-art open-source projects with which one can quantify the above-stated hypothesis. In this case, the author has control over which data to sample from desired populations, but can not influence the outcome or artifacts of the sampled data. Thereafter, a sample study seems the appropriate research method as the goal of a sample study is to generalize over a certain population [24]. This aligns with the aim of this study which tries to generalize the occurrence of code smells in data pipelines in comparison to regular code and code smells produced by data scientists compared to code smells produced by software engineers. Subsequently, this study samples code from the four populations; data scientists, software engineers, data pipelines, and regular code. The unit of analysis or investigated artifact of this study is the source code of selected files in repositories that are part of the desired population.

Since the most common code smells are defined by Fowler et al. [3], one can identify such code smells in source code either by manually reviewing code and searching for such anti-patterns or one can make use of static code analysis tools that identify code smells as well. Static code analysis tools are made upon a set of rules that are checked on the syntax of a given file. These rules identify errors, warnings, pattern violations, and other information in the code. Hence, when a rule is broken, that line in the code gets added to the report of the analysis tool stating which rule was broken in that specific line. For the purpose of this study, the author was mostly interested in broken conventions, warnings, and refactoring suggestions which are in essence code smells.

To establish a fair comparison, the chosen metric of this study is the amount of code smells per file for a subject of the given population. Thereby it does not matter how many files are analyzed per subject. Another advantage is that this metric can be used for all populations in this study.

D. *Data collection*

As pointed out in the previous section, this study focuses on open source repositories from GitHub. The main challenge in data gathering is to filter the vast amount of open-source repositories to a subset of interest. To filter the GitHub repositories, the author used the GitHub REST API. The API offers a search endpoint with which one can search GitHub resources via query strings. The limitation of that API is that it provides only the first 1000 results for a query. A summary of the mining process for data scientists and software engineers can be found in Fig. 1, while the process for pipelines and non-data pipelines is summarized in Fig. 2.

1) Data Scientists & Software Engineers: To find GitHub users that belong to the software engineer and data scientist population, the main criterion was found in the biography of the user's account. Many GitHub users state their role in their

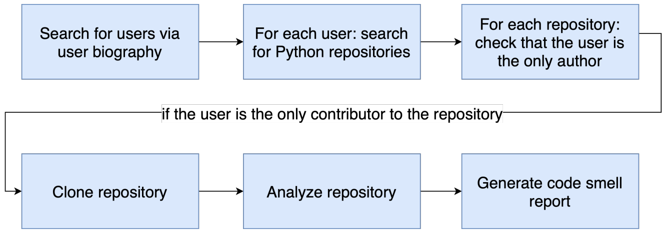

Fig. 1. Repository mining process for data scientists and software engineers

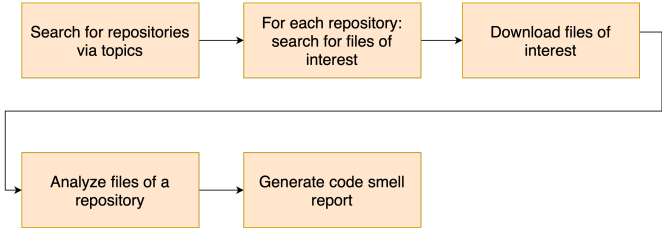

Fig. 2. Repository mining process for pipelines and non-data pipelines

biography. Thereafter, the selected criteria in a user's biography were the keywords "data scientist" or "data engineer" for the data scientist population and "software engineer" or "software-engineer" for the software engineer population. Due to the REST API rate limit, 1000 users could be fetched. Additionally, only users with more than five public repositories were selected to make sure that only users were chosen who can contribute to this study. Another criterion was that the users had to have at least 30 followers. A summary of the filter criteria can be found in Table I. With these filters, the author tried to exclude experimental accounts. With the list of users that fall into the populations in place, the next step was to select repositories from the users. First of all, only a maximum amount of five repositories have been taken into consideration for this study to avoid an over-representation of individual users. Next, only repositories where Python is the main programming language were selected to be able to use the same static code analysis tool for all populations which makes the results comparable. Another selection criterion for the repositories was that the repository size had to be between 70 and 20000 kilobytes. The lower threshold was selected to exclude small experimental repositories that are almost empty and do not represent a working program, while the upper threshold was selected to exclude very large repositories with potentially many files that would take a considerably long time to clone and process during the analysis. Finally, for the repository selection, the author had to make sure that the code in the selected repositories was written by the studied user. Hence only repositories were selected that were not forks of another repository and where the user was the only contributor. In GitHub, users can copy open source repositories from other users and modify the source code. Such a copy of a repository is called a fork and it is important to exclude such repositories from this study because the forked project might show just one contributor that did changes to the copy,

TABLE I
FILTER CRITERIA FOR DATA SCIENTISTS AND SOFTWARE ENGINEERS

	users	repositories
data scientist	**biography**: "data scientist" OR "data engineer" **public repositories:** >5 **followers:** >30	**maximal repositories per user:** 5 **programming language:** Python **repository size:** >70kB & <20 000kB **contributors:** 1 & user is only contributor **fork:** repository is not a fork
software engineer	**biography:** "software engineer" OR "software-engineer" **public repositories:** >5 **followers:** >30	**maximal repositories per user:** 5 **programming language:** Python **repository size:** >70kB & <20 000kB **contributors:** 1 & user is only contributor **fork:** repository is not a fork

while the original authors do not show up in the contributor list of the forked repository. After applying all these filters, the resulting repositories have been cloned to a local disk where the analysis was performed.

2) Data Pipelines & Not Data Pipelines: To find files for the comparison of data pipelines vs. none data pipelines, the most promising approach seemed to search for files that include common machine learning libraries. For the purpose of this study, the author selected the Tensorflow library. This library is primarily designed for the Python programming language. To increase the chance of finding such files in repositories, the author applied filters to repositories via the GitHub REST API. The main criterion in the query string was to filter for topics. In GitHub, the owners and contributors of repositories can tag their repositories with keywords. These tags are called topics and are comparable to tags on social media. The chosen topic for this study was to include repositories that are tagged with TensorFlow for the data pipeline population. For the not data pipeline population, repositories were excluded that were tagged with ai, machine-learning, TensorFlow, and PyTorch. Furthermore, to exclude experimental repositories, further filters were applied for both populations to only include repositories that have more than 30 stars, more than 20 forks, are bigger than 35 kilobytes, and have had at least one contribution since the first of April 2019. After filtering the repositories, the GitHub REST API was used again to search for files in the repositories. For the data pipeline population, files were selected that included an "import tensorflow" statement. For the not pipeline population, the author excluded files that imported the TensorFlow library, and to exclude data pipelines implemented with other libraries, files including the PyTorch or scikit-learn libraries were also excluded. The filter criteria for this comparison is summarized in Table II. During the study, it showed that data pipeline files can be very long. Hence the number of files per repository was limited to ten for both populations in an attempt to make sure that the analysis tool could handle the files. The selected files were downloaded and stored per repository on a local machine to be able to run the analysis.

E. Data analysis

After cloning the source code to a local machine, the objective was to analyze the source code of the repositories in terms of code smells. An efficient way for this kind of

TABLE II
FILTER CRITERIA FOR DATA PIPELINES AND NON-DATA PIPELINES

	repositories	files
data pipelines	**topic:** TensorFlow **programming language:** Python **stars:** >30 **forks:** >20 **size:** >35kB **latest contribution:** >2019-04-01	**maximal files per repository:** 10 **line of code:** "import tensorflow"
non-data pipelines	**topic:** NOT TensorFlow AND NOT ai AND NOT machine-learning AND NOT PyTorch **programming language:** Python **stars:** >30 **forks:** >20 **size:** >35kB **latest contribution:** >2019-04-01	**maximal files per repository:** 10 **line of code:** NOT "import tensorflow" AND NOT "from tensorflow" AND NOT "import torch" AND NOT "from torch" AND NOT "import sklearn" AND NOT "from sklearn"

analysis is to use a static analyzing tool that can identify code smells in source code and generate descriptive reports automatically without manual review. The selected analysis tool was pylint. The author selected pylint because of the support for the Python programming language. As explained above, this is important because the selected machine learning library TensorFlow is implemented for Python. Additionally, pylint offers great support to find code smells and defects in Python code ranging from fatal defects to broken coding conventions. A comprehensive list of all defects and smells pylint detects can be found in [22]. Thereafter, pylint includes code smell detection. In detail, pylint categorizes anti-patterns into fatal, error, warning, refactor, convention and information. However, for this study, the author took only the warning, refactor, and convention categories into account as the other categories either report on actual defects in the source code or other information. A further advantage is that the produced report can be stored in the JSON file format which makes it possible to automate the analysis and aggregation later on.

For this study, running pylint for a repository was set up in a way that it would take a maximum amount of 20 files into account for a single repository. During the study, it showed that considering too many files at once would crash the analysis tool. Hence, the boundary of 20 files per repository was set. For both the data pipeline and not data pipeline populations this was not an issue because here only a maximum amount of ten files were downloaded per repository in the first place. Since the entire repository was cloned for the data scientist and software engineer populations, there have been instances where more than 20 python files were present in a single repository. In that case, 20 Python files were selected at random for the analysis. With these restrictions, pylint was run on the selected files and the resulting reports were stored per repository. To summarise the findings for a population, another script was set up that went through the JSON reports for a population and counted the number of warnings, refactoring suggestions, broken conventions, and the number of analyzed files per repository. This data enabled the author to calculate the amount of code smells per file, warnings per file, refactor suggestions per file, and broken conventions per file for a repository. The results were stored in one CSV file for a population. The author decided to divide the amount of code smells by the number of files analyzed per repository because the number of files analyzed differed between repositories. Hence, by calculating the amount of code smells per file, the data could be compared between repositories and populations.

To answer the hypothesis and research questions, the code smells per file were statistically analyzed. Firstly, the data were checked for normality with the Shapiro-Wilk Test. As the results section will show, the data of all four populations were not normally distributed. Hence, a Man-Whitney U test was performed to determine statistical significance between the populations. In the case of significant results, the Rank-Biserial correlation indicates the effect size. The advantage of this measure is that it is strictly non-parametric, which means that the underlying distribution of the data does not matter [25]. The Rank-Biserial correlation can be interpreted in a way that a value of +1 means that all ranks of the greater category are above the ranks of the lower category. A value of -1 suggests the opposite, all ranks of the lower category rank higher than the ranks of the greater category.

IV. RESULTS

A. Data Scientists & Software Engineers

The final data set for the comparison of data scientists and software engineers included 1296 repositories with 9741 analyzed files for the data scientist population and 632 repositories with 5537 analyzed files for the software engineering population. With the chosen filter criteria for the selection of users and repositories during the data collection, more than twice as many repositories from data scientists were analyzed compared to software engineers. Such a big difference in the number of selected repositories was definitely not intended with the filter criteria. However, the author argues that the selected criteria are reasonable for this study as both populations show a sufficiently large data set to perform statistical analysis.

By comparing the total code smells per file for both populations, it becomes evident that the data scientist population showed more code smells per file than the software engineering group. The mean value of code smells per file shows 27.204 for data scientists and 20.491 code smells per file for the software engineering group. Considering the minimum amount of code smells per file for both groups, reveals that both populations contain repositories with zero code smells per file according to the used analysis tool. For the data scientist population, 96 out of the 1296 repositories had no code smells, for the software engineer population it is 54 out of the 632 repositories. In contrast to the minimal amount of code smells, the maximal amount differs drastically. While the maximal amount of code smells per file for a repository of the data scientist population is 790.389, the software engineering population showed a maximal 299.950 code smells per file. With the help of the box plot in Fig. 3, one can figure that the maximum values of both populations consist of a few outliers that stand apart from the rest of the data by a big margin. In the case of data scientists, the five repositories with the most code smells per file accumulate the vast amount of code

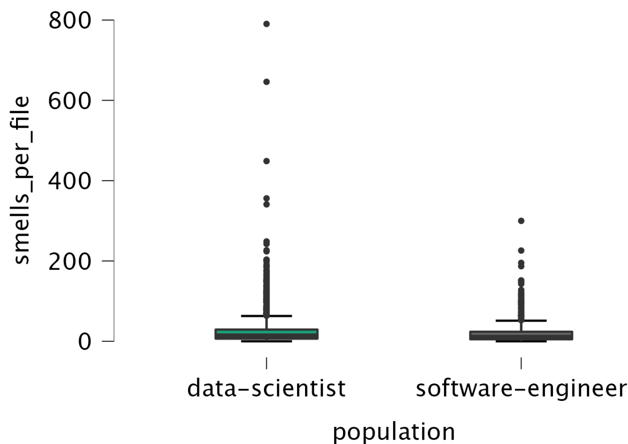

Fig. 3. Box plot for code smells from data scientists and software engineers

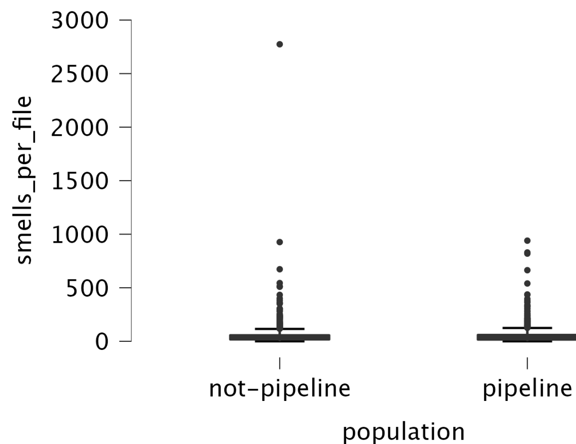

Fig. 4. Box plot for code smells in data pipelines and non-data pipelines

smells mostly via broken conventions such as too long lines and broken naming conventions.

To determine if the data was normally distributed the Shapiro-Wilk was performed. With a p-value smaller than the confidence interval of 0.05, the data is not normally distributed. Hence, the Mann-Whitney U test was performed to determine the statistical significance between the two groups. With a p-value smaller than 0.001 and therefore smaller than the confidence interval of 0.05, the Mann-Whitney U test suggests a significant result. Thereafter, the null hypothesis of H1 can be rejected. This indicates that in this study software engineers produced fewer code smells than data scientists. To measure the effect size of this comparison, the Rank-Biserial Correlation was calculated. With a value of 0.125, the value indicates a small effect size.

Table III shows the descriptive statistics of the different categories from pylint next to the overall code smells per file. As the data suggests, the average code smells per file are higher for the data scientist group in every category. Albeit, the difference in the mean code smells per file in the refactor category is relatively small with a difference of 0.328. Additionally, the p-value of the Shapiro-Wilk test suggests that the data of all categories are not normally distributed. Thereafter, a Mann-Whitney U test was performed for every category. The results of those tests can be found in Table IV. As the table depicts, there is a significant difference between the two populations in the total code smells per file, conventions per file, and warnings per file. On the contrary, the amount of refactoring suggestions is not statistically relevant with a p-value of 0.168. With Rank-Biserial correlation values of 0.143 for the conventions category and 0.100 for the warnings category, the effect sizes for these two comparisons are also small.

B. Data Pipelines & Not Data Pipelines

For the comparison of data pipelines and not data pipelines, 892 repositories with 5763 files have been analyzed for the data pipeline population and 927 repositories with 7336 files for the not data pipeline population. The descriptive statistics for this comparison can be found in Table V. By comparing the overall code smells per file, it can be observed, that the mean code smells per file are very similar with 54.108 and 54.448 for not pipelines and pipelines respectively. Similar to the comparison of software engineers and data scientists, this sample study also includes repositories with no code smells for both groups. In this case, no code smells can be found in ten out of the 892 pipeline repositories and in just three out of the 927, not pipeline repositories. As Table V and the box plot in Fig. 4 show, the non-data pipeline population contains a massive outlier that has 2774 code smells per file. This particular repository represents a monitoring tool that has bundled the source code of the application in a single python file with 5687 lines of code. Most of the code smells for this repository were warnings about bad indentations and too-long lines of code.

As the Shapiro-Wilk test reveals, the amount of code smells per file is not normally distributed for the pipeline and non-pipeline population. The test results in a p-value of less than 0.001 for both populations. Since the data is not normally distributed, a Mann-Whitney U test was performed to determine the statistical significance between the two groups. With a p-value of 0.617 and therefore by a big margin larger than the confidence interval of 0.05, the Mann-Whitney test does not suggest a significant result. In which case it does not make sense to calculate an effect size. Thereafter, this study fails to reject the null hypothesis of H2.

Even with the overall code smells per file not showing significant results, the individual code smells categories provide interesting insights. As Table V depicts, the average code smells per file are actually higher for the not data pipeline population in the refactor and conventions categories. Just the mean value of the raised warnings per file category is lower for the not data pipeline group compared to data pipelines. Which is remarkable because the warning category contains the majority of the earlier described massive outlier in the not-data pipeline group. The maximal amount of code smells per

TABLE III
DESCRIPTIVE STATISTICS DATA SCIENTISTS AND SOFTWARE ENGINEERS

	smells_per_file		conventions_per_file		warnings_per_file		refactor_per_file	
	data-scientist	software-engineer	data-scientist	software-engineer	data-scientist	software-engineer	data-scientist	software-engineer
Valid	1296	632	1296	632	1296	632	1296	632
Missing	0	0	0	0	0	0	0	0
Mean	27.204	20.491	16.257	11.902	8.978	6.949	1.968	1.640
Std. Deviation	47.173	30.410	29.875	16.815	21.654	18.963	5.052	2.555
Variance	2225.317	924.759	892.527	282.748	468.906	359.579	25.526	6.526
Shapiro-Wilk	0.478	0.606	0.409	0.616	0.412	0.348	0.287	0.610
P-value of Shapiro-Wilk	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$
Minimum	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Maximum	790.389	299.950	477.000	149.800	247.500	289.050	135.000	23.450

TABLE IV
INDEPENDENT SAMPLES T-TEST DATA SCIENTIST AND SOFTWARE ENGINEER

	W	df	p	Rank-Biserial Correlation
smells_per_file	460835.500		$< .001$	0.125
conventions_per_file	468263.500		$< .001$	0.143
warnings_per_file	450353.000		$< .001$	0.100
refactor_per_file	420530.500		0.168	0.027

Note. For all tests, the alternative hypothesis specifies that group *data-scientist* is greater than group *software-engineer*.
Note. Mann-Whitney U test.

file depicts that well. While the not data pipeline population scores a maximum of 2213.5 warnings per file, the data pipeline group just counts 703 as the maximum value. Again, the Shapiro-Wilk test indicates that the data for all categories is not normally distributed. Hence, the Mann-Whitney U test was performed to measure the statistical significance, shown in Table VI. The chosen alternative hypothesis for this test was in sync with hypothesis 2 from section III-B stating that not data pipelines contain less code smells than data pipelines. For the refactor and conventions categories, the resulting p-value has a value of 1.0. This indicates that with a certainty of 100% none-data pipelines do not contain less code smells than data pipelines in the refactor and convention category of this study. The p-value of the Mann-Whitney U test for the warning category results in a value of 0.131 and is therefore not lower than the confidence interval of 0.05, which means that from a statistical perspective we can not say that the data pipelines contain fewer code smells in the warning category than none data pipelines.

V. DISCUSSION

A. Research Question 1

In research question one of this study I asked *How does the code quality from data scientists compare to code quality from software engineers in terms of code smells?* As the results show, the study is able to reject the null hypothesis of H1. This means that code produced by software engineers yields fewer code smells than code produced by data scientists in the context of this study. This observation is not only true for the average amount of code smells per file, but it also proves to hold for both the warnings and broken convention code smell categories. Only the refactor category does not provide significant results. This means that the overall results for this research question are in line with anecdotal events in literature which are now quantified with this systematic study. The experience by Wilson [12], [17] claims that scientists without a software engineering background often struggle with software engineering practices. With the results of this study, one can argue that this argument counts as well for data scientists in terms of producing code smells. By looking into the individual categories, one can identify that the broken conventions category seems to be the biggest struggle for data scientists with 16.257 violations per file on average. It has to be said though, that this is also the category where software engineers struggle the most. Albeit, data scientists produce significantly more code smells in this category than software engineers.

Despite the category where the code smells come from, code smells have a negative impact on the maintainability and readability of code [14]–[16]. Therefore, the author advises developers to make an effort in reducing code smells in their projects. One way to do so is by implementing static code analysis tools such as pylint for Python directly into the CI/CD pipeline of a project to fix the code smells before they end up in production code and become technical debt. With the results of this study, this becomes especially true for data scientists. Moreover, the results of this study are also valuable for educators of data scientists. As the results showcase, data scientists produce significantly more code smells than software engineers. Hence it makes sense for educators to sensitize data scientists in their education more to code smells. For researchers, the results provide the opportunity to extend the research and to investigate code smells from data scientists in more detail. Section VIII provides suggestions on how to enhance the findings of this study.

B. Research Question 2

To find answers for research question two which states *How does the code quality of data pipelines compare to non-data pipeline code in terms of code smells?*, I ran a second sample study. The results of that study failed to reject the underlying null hypothesis of H2. Hence, this study does not show that non-data pipelines contain significantly fewer code smells than data pipelines. On the contrary, the results of the statistical analysis of the code smell categories suggested by pylint, reveal that for the refactor and conventions category equal or

TABLE V
DESCRIPTIVE STATISTICS DATA PIPELINES AND NON-DATA PIPELINES

	smells_per_file		conventions_per_file		refactor_per_file		warnings_per_file	
	not-pipeline	pipeline	not-pipeline	pipeline	not-pipeline	pipeline	not-pipeline	pipeline
Valid	927	892	927	892	927	892	927	892
Missing	0	0	0	0	0	0	0	0
Mean	54.108	54.448	30.391	26.248	6.255	4.164	17.461	24.036
Std. Deviation	113.473	80.842	41.045	35.589	7.733	5.088	82.680	56.583
Shapiro-Wilk	0.288	0.528	0.545	0.559	0.640	0.611	0.135	0.426
P-value of Shapiro-Wilk	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$
Minimum	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Maximum	2774.000	940.000	495.000	390.500	70.000	69.000	2213.500	703.000

TABLE VI
INDEPENDENT SAMPLES T-TEST DATA PIPELINES AND NON-DATA PIPELINES

	W	df	p	Rank-Biserial Correlation
smells_per_file	416777.500		0.617	0.008
refactor_per_file	496587.500		1.000	0.201
conventions_per_file	453296.500		1.000	0.096
warnings_per_file	400899.000		0.131	−0.030

Note. For all tests, the alternative hypothesis specifies that group *not-pipeline* is less than group *pipeline*.
Note. Mann-Whitney U test.

fewer code smells are found in data pipelines. Thereafter, the research question can be answered in a way that data pipelines do not contain more code smells than non-data pipelines. However, just because there are not more code smells in data pipelines compared to no data pipelines, does not mean that code smells will not create issues in data pipelines. As code smells continue to be researched in literature in an attempt to understand their implications, the author argues that data pipelines should be involved in that discussion. As Munappy et al. [2] put it, data pipelines are just another piece of software. Hence, the author argues that it is well worth making an effort in trying to decrease the amount of code smells in data pipelines to increase maintainability, readability and to save precious resources in the future.

VI. LIMITATIONS

Like every study, this one comes with a set of validity threats. In terms of **external validity**, a potential issue is to select comparable repositories to establish a fair comparison. Factors that try to keep the comparison of repositories fair in this study are the repository size, number of stars and forks, and the latest contribution date as described in section III-C. Another threat is generalizability in terms of the time frame. As the software community increases, best practices could potentially spread into other domains. Furthermore, the number of files in the repositories differs a lot. To counter this issue and to make sure that the analysis tool could handle the number of files for a single repository, the author took two measures. On one hand, the author decided to only take a maximum amount of 20 files per repository into account for the data scientist and software engineer comparison and just ten files per repository for the data pipeline and not data pipeline comparison. In the case that more files than the threshold were downloaded, the files were selected randomly. This way also larger repositories could participate in this study. On the other hand, the desired metric of this study has been the amount of code smells per file for a repository. With this metric, the author was able to compare the amount of code smells across repositories with different amounts of files. For the comparison of data pipelines and none data pipelines, an additional validity threat is that the exclusion of topics in the GitHub REST API does not work flawlessly. This is a known bug and reported multiple times [26]. With this bug, the author can not be certain that none of the excluded topics ended up in the final data set of the non-data pipeline population. But, to make sure that at least no data pipeline files end up in the final data set, the author excluded files that use import statements from other common machine learning libraries. These libraries are TensorFlow, scikit-learn, and PyTorch. Since excluding code statements from files is done via another API endpoint and different query syntax, the author is convinced that no pipeline files ended up in the data set. Additionally, the author manually validated files from twenty repositories, which did not include data pipeline files.

In regards to **internal validity**, the biggest threat is the selection bias of the criteria that includes and excludes the repositories from the study. However, the author argues that the selected criteria form the borders for a fair comparison. A further internal thread is how the author decided to gather the user profiles that make data scientists and software engineers. Linking users to a population, based on their biography text on GitHub gives the author no chance to verify that the role the user claims to have is actually true. Furthermore, this study does not take the background of individual users into account. Since every user can put whatever they choose in their GitHub biography, there could be users who claim to be in a certain role with no background education at all. On the other hand, there could be users who claim to be data scientists and actually have a software engineering background and vice versa. On a further note, even checking that the studied user is the only author of a repository, does not guarantee that they wrote all the code. There are instances, where people commit external libraries, which have been written by external

sources, to their repositories. Other examples of committing code written by someone else is to not fork a repository directly on GitHub but to download the source code and upload it to a new repository or copying code snippets form public sources like stack overflow.

A further internal threat is the selected tool to analyze the source code. To counter this threat, industries' most recognized tools were selected. The limiting factors of this study are query rates, for the GitHub REST API the query limit is 5000 queries per hour and a maximum of 1000 results for a request to the search endpoint for an authenticated user. Nonetheless, the author argues that the acquired data sets are sufficiently large to form a population study. A different limiting factor is processing power for analyzing the source code with the static code analyzing tools. Analyzing hundreds of code sections can potentially take a lot of time to process. To counter this, access to a specialized computing infrastructure at the Swedish National Infrastructure for Computing has been set up.

VII. Conclusions

This study contributes to the current research of technical debt in software engineering by asking how code written by data scientists compares to code written by software engineers in terms of code smells. Furthermore, this study investigates code smells in machine learning pipelines compared to not machine learning pipelines. To acquire source code relevant to this study, the author selected open source repositories from the version control system GitHub via filter criteria. After cloning selected files to a local machine, the static code analysis tool pylint was used to identify and summarize code smells in the repositories. The results reveal that the code written by data scientists yields more code smells than the code written by software engineers. With the results of this study, the author was able to contribute to the assumption in the literature that scientists without a software engineering background struggle with best practices when writing code. As this study focuses on data scientists, the study contributes to this field for data scientists.

Furthermore, this study failed to showcase that data pipelines contain more code smells than non-data pipelines. On the contrary, that does not mean that code smells can not become technical debt in data pipelines. Thereafter, the author suggests continuing in making an effort to reduce code smells for data pipelines and non-data pipelines in an attempt to increase the maintainability and readability of source code.

VIII. Future Work

To extend this study in the future, two paths seem straightforward. On one hand, this study only takes the TensorFlow library into account for the machine learning population. Hence, this study can be extended by investigating the phenomenon of code smells with other ML libraries. Suggestions here are scikit-learn, Pytorch, and Keras. On the other hand, this study does not take the background of the studied developers into account as section VI discusses further. Thereafter, it makes sense to conduct this study with a qualitative analysis as well. In this case, a further research question emerges in the sense of how aware data scientists and software engineers are of their produced code smells. The following question is: Are the developers aware that they are implementing code smells and do it non the less or do they not know in the first place? To gather the qualitative data, the author suggests either running this study together with industry partners or interviewing developers of open source projects. This way the researchers can also learn more about the individual backgrounds of the developers, which can give a more reliable data set. Furthermore, it would be interesting to investigate how code smells in data pipelines differ in industry projects compared to open source projects. As this study takes mostly generic Python code smells into account, another possible enhancement of this study is to investigate if there are data pipeline-specific code smells.

Acknowledgment

The author of this study would like to thank his supervisor Daniel Strüber for the help and guidance while conducting the research.

References

[1] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," 5 2019, pp. 291–300.

[2] A. R. Munappy, H. H. Olsson, and J. Bosch, "Data pipeline management in practice: Challenges and opportunities," Marco, J. A. M. Maurizio, and Torchiano, Eds. Springer International Publishing, 2020, pp. 168–184.

[3] M. Fowler, E. Gamma., and K. Beck, *Refactoring : improving the design of existing code.* Addison-Wesley, 2000. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=cat07470a&AN\=clc.53577083.7739.4078.a00b.7ae8a17c9750&site=eds-live&scope\=site&authtype=guest&custid=s3911979&groupid=main&profile=eds

[4] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, pp. 18–21, 11 2012.

[5] D. Falessi and R. Kazman, "Worst smells and their worst reasons," 2021, pp. 45–54.

[6] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell: An empirical study." Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2810146.2810155

[7] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, pp. 20–36, 2010.

[8] A. Alazba and H. Aljamaan, "Code smell detection using feature selection and stacking ensemble: An empirical investigation," *Information and Software Technology*, vol. 138, p. 106648, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921001129

[9] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smells with machine learning algorithms: An empirical study." Association for Computing Machinery, 2020, pp. 31–40. [Online]. Available: https://doi.org/10.1145/3387906.3388618

[10] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Software Quality Journal*, vol. 28, pp. 1063–1086, 2020. [Online]. Available: https://doi.org/10.1007/s11219-020-09498-y

[11] Continuous delivery for machine learning. [Accessed: 2022-03-09]. [Online]. Available: https://martinfowler.com/articles/cd4ml.html

[12] G. Wilson, "Software carpentry: Getting scientists to write better code by making them more productive," *Computing in Science Engineering*, vol. 8, pp. 66–69, 11 2006.

[13] G. Booch, *Object-oriented analysis and design with applications.* Addison-Wesley, 2007. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=cat07470a&AN\=clc.3abf3272.f95e.4a2c.9dbe.4499c0d53fec&site=eds-live&scope\=site&authtype=guest&custid=s3911979&groupid=main&profile=eds

[14] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality." Association for Computing Machinery, 2011, pp. 17–23. [Online]. Available: https://doi.org/10.1145/1985362.1985366

[15] D. Falessi and A. Voegele, "Validating and prioritizing quality rules for managing technical debt: An industrial case study," 10 2015, pp. 41–48.

[16] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," 3 2011, pp. 181–190.

[17] G. Wilson, J. Bryan, K. Cranston, J. Kitzes, L. Nederbragt, and T. K. Teal, "Good enough practices in scientific computing," *PLOS Computational Biology*, vol. 13, pp. 1–20, 3 2017. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1005510

[18] A. Patnaik and N. Padhy, "Does code complexity affect the quality of real-time projects? detection of code smell on software projects using machine learning algorithms." Association for Computing Machinery, 2021, pp. 178–185. [Online]. Available: https://doi.org/10.1145/3484824.3484911

[19] R. MacDonald, "Software defect prediction from code quality measurements via machine learning," E. Bagheri and J. C. K. Cheung, Eds. Springer International Publishing, 2018, pp. 331–334.

[20] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi, "Towards microservice smells detection." Association for Computing Machinery, 2020, pp. 92–97. [Online]. Available: https://doi.org/10.1145/3387906.3388625

[21] V. R. Basili, G. Caldiera, and D. H. Rombach, *The Goal Question Metric Approach*. John Wiley & Sons, 1994, vol. I.

[22] Messages overview - pylint 2.14.0-b1 documentation. [Accessed: 2022-05-25]. [Online]. Available: https://pylint.pycqa.org/en/latest/user_guide/messages/messages_overview.html

[23] Code search · github. [Accessed: 2022-03-10]. [Online]. Available: https://github.com/search

[24] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, sep 2018. [Online]. Available: https://doi.org/10.1145/3241743

[25] E. E. Cureton, "Rank-biserial correlation," *Psychometrika*, vol. 21, pp. 287–290, 1956. [Online]. Available: https://doi.org/10.1007/BF02289138

[26] Github search api - exclude certain topics - stack overflow. [Accessed: 2022-05-23]. [Online]. Available: https://stackoverflow.com/questions/65554246/github-search-api-exclude-certain-topics