Institute of Software Engineering

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Machine Learning Frameworks in Open-Source Software: An Exploratory Study on Code and Project Smells

Marius Hauser

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dr. Justus Bogner
Commenced:	December 16, 2022
Completed:	June 16, 2023

Abstract

Machine Learning (ML) gained an increasing amount of interest in recent years. The widespread use of ML systems raises questions regarding technical debt and the utilisation of good software engineering practices. ML systems are complex systems which are faced with additional challenges, compared to traditional software systems. These additional challenges are among others facilitated in the areas of dependency management and data versioning which can lead to an increased susceptibility to technical debt.

To address those new challenges, this study investigates how the choice of a ML library is associated with types and frequency of code and project smells. This study additionally acquires the distribution of application areas in open-source projects that use Machine Learning libraries. In this study repository mining is performed, followed by a large-scale analysis of code and project smells using SonarQube and *mllint*. SonarQube is used to find code smells in python source code and project smells are tracked using a score calculated by *mllint*. A lower *mllint* score corresponds to more project smells. All mined repositories are categorised in domain categories using an automated classifier and in exploratory found topics using Latent Dirichlet Allocation.

This study analyses 6,840 open-source software repositories which use the ML libraries “TensorFlow”, “Scikit-learn”, “Transformers”, “Keras”, “PyTorch”, “Keras”, and “Keras & TensorFlow”. Violations of naming conventions, commented-out code sections, and high cognitive complexity of source code sections are the most prominent code smells found among all repositories.

Several correlations between projects using ML libraries and the frequency of code and project smells have been found. Statistical analysis revealed that using TensorFlow as ML library is associated with ~12.6% more code smells per 1,000 LoC than using Transformers. Using Scikit-learn is correlated with a ~25% higher *mllint* score compared to TensorFlow and a ~44% higher *mllint* score than using PyTorch. Regarding project smells this study revealed that using Transformers is correlated with a ~13% higher average *mllint* score than using TensorFlow.

Application areas of Machine Learning libraries are for the most part in line with the areas advertised by their publishers. This is, among other reasons, due to the general application of their libraries as stated by their publishers.

Based on this study, future work can investigate causal relationships between ML libraries which are associated with a higher frequency of code and project smells than other ML libraries. Furthermore, an investigation of practitioners, maintainers, and developers working with ML libraries can reveal differences in the types of users and formulate best practices to reduce code and project smell.

Kurzfassung

Machine Learning (ML) hat in den letzten Jahren immer mehr an Popularität gewonnen. Der weit verbreitete Einsatz von ML-Systemen wirft Fragen zur technical debt und zur Anwendung guter Softwareentwicklungspraktiken auf. ML-Systeme sind komplexe Systeme, die im Vergleich zu traditionellen Softwaresystemen mit zusätzlichen Herausforderungen konfrontiert sind. Diese zusätzlichen Herausforderungen liegen unter anderem in den Bereichen Abhängigkeitsmanagement und Datenversionierung, die zu einer erhöhten Anfälligkeit für technische Schulden führen können.

Um diesen neuen Herausforderungen zu begegnen, wird in dieser Studie untersucht, wie die Wahl einer ML-Bibliothek mit der Art und Häufigkeit von Code- und Projekt-Smells zusammenhängt. Darüber hinaus wird in dieser Studie die Verteilung der Anwendungsbereiche in Open-Source-Projekten, die Machine Learning-Bibliotheken verwenden, erfasst. In dieser Studie wird ein Repository Mining durchgeführt, gefolgt von einer groß angelegten Analyse von Code- und Projekt-Smells mit SonarQube und *mllint*. SonarQube wird verwendet, um Code-Smells im Python-Quellcode aller analysierten Repositories zu finden. Projektgerüche werden anhand der *mllint*-Punktzahl verfolgt. Eine niedrigere *mllint*-Punktzahl entspricht mehr Projekt-Smells. Alle untersuchten Repositories werden in Domänenkategorien und mithilfe von Latent Dirichlet Allocation in explorativ gefundene Themen kategorisiert.

Diese Studie analysiert 6.840 Open-Source-Software-Repositories, die die ML-Bibliotheken “TensorFlow”, “Scikit-learn”, “Transformers”, “Keras”, “PyTorch”, “Keras” und “Keras & TensorFlow” verwenden. Verstöße gegen Namenskonventionen, auskommentierte Codeabschnitte und eine hohe kognitive Komplexität von Quellcodeabschnitten sind die auffälligsten Code-Smells, die in allen Repositories gefunden wurden.

Es wurden mehrere Korrelationen zwischen Projekten, die ML-Bibliotheken verwenden, und der Häufung von Code- und Projekt-Smells. Die statistische Analyse ergab, dass die Verwendung von TensorFlow als ML-Bibliothek mit ~12,6% mehr Code-Smells pro 1.000 LoC verbunden ist als die Verwendung von Transformers. Die Verwendung von Scikit-learn korreliert mit einem ~25% höheren *mllint*-Wert im Vergleich zu TensorFlow und einem ~44% höheren *mllint*-Wert als die Verwendung von PyTorch. Im Bezug auf Projekt-Smells hat diese Studie ergeben, dass die Verwendung von Transformers, im Vergleich zur Verwendung von TensorFlow, mit einer ~13% höheren durchschnittlichen *mllint*-Punktzahl korreliert.

Anwendungsbereiche von Machine Learning-Bibliotheken stimmen weitgehend mit den von ihren Herausgebern beworbenen Bereichen überein. Dies ist unter anderem auf die von den Herausgebern angegebene allgemeine Anwendbarkeit der Bibliotheken zurückzuführen.

Auf der Grundlage dieser Studie können künftige Arbeiten kausale Zusammenhänge zwischen ML-Bibliotheken untersuchen, die mit einer höheren Häufigkeit von Code- und Projekt-Smells verbunden sind als andere ML-Bibliotheken. Darüber hinaus kann eine Untersuchung von Anwendern, Betreuern und Entwicklern, die mit ML-Bibliotheken arbeiten unterschiede in den Benutzertypen aufdecken und Best Practices zur Reduzierung von Code- und Projekt-Smells formulieren.

Contents

List of Figures

List of Tables

Acronyms

ACE Automatic Content Extraction. 29

AI Artificial Intelligence. 3, 42

API Application Programming Interface. 6, 7

CI Continuous Integration. 35, 37, 40

DVC Data Version Control. 17

LDA Latent Dirichlet Allocation. ii, iii, 1, 11, 15, 24, 25, 28, 29, 30, 31, 38, 41, 42

LoC Line of Code. ii, iii, v, vi, 23, 25, 33, 34, 37, 39, 42, 51, 56

ML Machine Learning. ii, iii, iv, v, vi, 1, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 24, 25, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 56

MSR Mining Software Repositories. 14

NLP Natural Language Processing. 7, 14, 18, 29, 38

ROGUE Recall-Oriented Understudy for Gisting Evaluation. 29

TPU Tensor Processing Unit. 29, 38

1 Introduction

ML systems are complex systems which are faced with additional challenges, compared to traditional software systems. These additional challenges are among others facilitated in the areas of dependency management and data versioning which can lead to an increased susceptibility to technical debt.

Machine Learning (ML) gained an increasing amount of interest in recent years and this trend is predicted to continue to grow in the following years [24]. This trend raises questions regarding technical debt and the use of good software engineering practices. Implementing ML systems come with additional challenges, for example, dependency management and data management [23]. Sculley et al. [61] argue that Machine Learning (ML) systems are prone to technical debt due to the maintenance problem coming from traditional code and the additional ML-specific issues.

Machine Learning libraries are used to ease implementation efforts for the whole workflow of implementing ML systems. A Kaggle survey on Data Science & Machine Learning in 2021 revealed that 55% of respondents have less than three years of experience in Machine Learning [34]. Wilson [71, 72] describes a lack of training in programming for students, resulting in reduced productivity. The growing field of Machine Learning, both for practitioners and in research and the lack of Software Engineering best practices being applied, shows the importance of a further analysis [64]. The partial inexperience of practitioners can lead to bad design decisions and therefore code and project smells which can result in bugs or technical debt. To gain more insights on those issues and provide assistance to practitioners choosing a good-fit Machine Learning (ML) library for their projects, this study looks into code and project smells in open-source software.

The contribution of this thesis is an analysis of the occurrence of Code and Project Smells in connection with the usage of Machine Learning libraries in python. Python is the most common skill for Data Scientists, according to a survey by Mooney [48]. Additionally, the application areas of theseMachine Learning libraries are extracted and compared to the stated application areas and functionalities by their publishers. The application areas of ML libraries in open-source projects are extracted using Latent Dirichlet Allocation (LDA) and an automated domain category classifier. This study reveals correlations between the frequency of code smells and the usage of a specific ML library in open-source software projects on GitHub. This enables another perspective of development behaviour and application fields of ML libraries, other than the classical way of performing surveys or interviews.

Thesis Structure

Chapter 1 – Introduction:

Chapter 2 – Foundations and Related Work: This chapter introduces key concepts of Machine Learning, the motivation behind using ML libraries and related work to this study.

Chapter 3 – Study Design: The Study Design explains the structure of this study with its integral steps and components. Repository sampling steps and data collection are explained and important challenges are highlighted. The methods for the statistical analysis are also laid out.

Chapter 4 – Results: This chapter answers the two research questions this study aims to answer and provides additional insights found by analysis the dataset.

Chapter 5 – Discussion: The discussion sets the results critically into context and real-world implications are discussed. Furthermore, threats to the validity of this study are presented.

Chapter 6 – Conclusion: The conclusion summarises the study design, methods and results of this study. Potential focus areas of future work are presented. Benefits, limitations and lessons learned from this study are also outlined.

2 Foundations and Related Work

In this chapter, the main foundations this study is based on are introduced. Additionally, related work to this study is presented and the key differences to this study are presented.

2.1 Foundations

This section introduces the main concepts of Machine Learning (ML) are introduced and the main motivation in using Machine Learning frameworks. This study is based on Machine Learning systems and the libraries used to implement ML systems.

2.1.1 Machine Learning

Machine Learning is a sub-field of Artificial Intelligence (AI) [3, p. 19]. The main idea is to apply concepts of human learning to machines. It differs from traditional programming in the core concept of how systems are developed. Traditional programming requires inputs and rules to calculate outputs. Machine Learning, on the other hand, tries to find rules, based on inputs and the desired outputs [47]. More generally speaking, "instead of code as the artifact of interest, in Software 2.0 it is all about the data where compilation of source code is replaced by training models with data" [47]. Software 2.0 is a novel synonym of Machine Learning software.

Learning Model

A Learning Model describes the way a machine can acquire knowledge. Figure 2.1 shows a general Learning Model. A teacher or the environment provides a stimulus to a Learner Component. The Learner Component creates or modifies the knowledge structures in the Knowledge Base. The Performance Component uses the Knowledge base to perform an action (e.g. solving a problem). This action is then evaluated by the Performance evaluation component. Feedback is derived from it and sent to the Learner Component, which updates the Knowledge base according to the feedback. This cycle is repeated until a desired performance level has been reached.

The concept of a learning machine is put into practice by seven learning strategies listed in increasing order of complexity [14]:

- Rote Learning
- Learning by Instruction
- Learning by Deduction

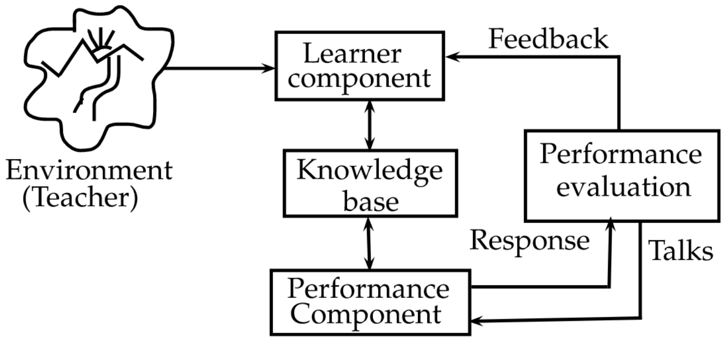

Figure 2.1: Learning Model by Chowdhary [14]

- Learning by Analogy
- Learning by Induction (or Similarity)
- Reinforcement Learning
- Discovery-based Learning

Overall, there are two classes of learning: "Supervised Learning" and "Unsupervised Learning".

Supervised Learning Supervised Learning methods require some user intervention, for example, users must provide at least some training data. Training data consists of data points with their corresponding label [65, p. 51-52]. This concept of needing a "teacher" who knows the solution to a given problem, resulted in the name of this learning class "supervised learning". In addition to classification problems, *supervised learning* can also solve regression problems. In this case, outputs are real values like a height or sum of money and no class or category [3, p. 13-14]. According to Chowdhary [14], supervised methods assume the following:

- Existence of some teacher (environment)
- A fitness function to measure the fitness of an example for a class
- External methods used to classify the training instances.

Supervised Learning can also be used to solve classification problems. Even if a problem does not seem to be a classification problem, many Machine Learning problems can be translated into one. For example, the detection of cars in pictures can be altered into a classification problem "car in picture" and "no car in picture" [3, p. 13-14].

Unsupervised Learning Unsupervised learning allows searching for patterns in big datasets. Learning algorithms get input data without a label. They only get the raw input data without the corresponding output. There is no *teacher* or *supervisor* like in "supervised learning" that knows output data or labels. An example by El-Amir and Hamdy [3] of *unsupervised learning* is a model which can predict which student passes a course, based on certain factors (demographic and pedagogical). The two most common problem-solving techniques are *clustering* and *association*. A basic association rule is “People who buy product X also buy product Y”. Clustering divides, for example, customers into groups with different purchasing behaviour [3, p. 15-16].

A common unsupervised learning method is Reinforcement Learning.

Reinforcement Learning Reinforcement Learning methods consist of four main components [35]:

- Environment states
- Agent actions
- Reinforcement signals

Possible actions of a *reinforcement learning* algorithm are defined under a policy. A policy contains a set of rules. The reward function maps each state of the system to a reward measure. Reward measures reflect the need of achieving the stated goal [14]. Generally, the agent tries to find a policy that maximizes a long-term measure of reinforcement [35].

2.1.2 Machine Learning Development

Machine Learning Systems

Sculley et al. [61] state that Machine Learning Systems only consist of a small part of ML code and mostly of support code. Support code is used to adapt a system to a generic software package (e.g., a ML library), but is not limited to ML systems [61]. In the context of data analytics platforms, support code is used to combine various components of a data analytics platform into one workflow [42].

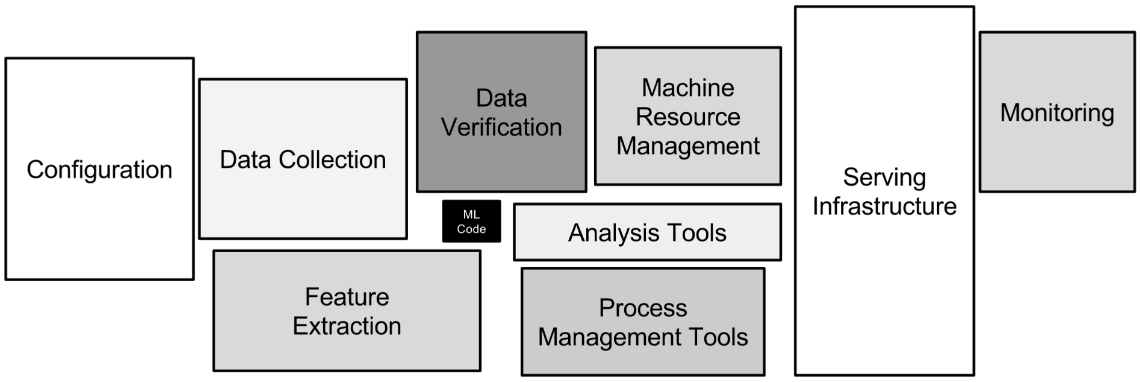

Figure 2.2: Components of a Machine Learning system [61, p. 4]

Figure 2.2 shows an exemplary Machine Learning system.

Libraries like TensorFlow and Keras[1] are advertised as end-to-end frameworks, implying they not only cover ML code but also support code [67].

[1]https://keras.io/about/, accessed: 01.06.2023

Using a specific library can therefore affect the whole system since they are designed as end-to-end libraries. Library design decisions paving the way for code smells would apply to the whole Machine Learning system.

Machine Learning Libraries

Python is the main programming language for Machine Learning applications and the main skill for Data Scientists in 2020 [48]. Machine Learning libraries ease the development effort when implementing ML systems and enable fast access to ML algorithms. Using ML poses challenges due to the dependence on data, pre-trained models, rapid evolution and the need for appropriate hardware. Companies like Facebook, Microsoft, and Google released their own libraries, and investing in open-source libraries allows practitioners to implement and use state-of-the-art ML systems and algorithms with less effort than without those libraries [20].

In the following, five common Machine Learning libraries are introduced with a focus on application areas and their characteristics.

TensorFlow TensorFlow is an end-to-end machine learning platform used to create production-grade machine learning models. An additional focus is the creation of scalable ML solutions. The official website[2] states three main reasons to use TensorFlow:

- Easy model building
- Robust ML production everywhere
- Powerful experimentation for research

Additionally, its high-level Application Programming Interface (API) results in good ease of use. The official tutorials[3] for TensorFlow emphasise “vision”, “Text”, “Audio”, “Structured Data”, “Generative” and “Reinforcement Learning”. TensorFlow is open-source[4] and licensed under the Apache License 2.0.

Wang et al. [70] state the main benefits of TensorFlow in its multi-language support, good support for multi-CPU, GPU or hybrid support and real portability.

Keras *Keras*[5] is an open-source deep learning library. *Keras* is built on top of the TensorFlow platform [38]. Its key design philosophy is a consistent and simple API and it follows best practices to reduce *cognitive load*. The official website provides code examples[6] for the categories “Computer Vision”, “Natural Language Processing”, “Structured Data”, “Timeseries Data”, “Generative Deep Learning”, “Audio Data”, “Reinforcement Learning”, and “Graph Data”. Just like TensorFlow, Keras is licensed under Apache License 2.0[7].

[2] `https://www.tensorflow.org`, accessed: 27.05.2023
[3] `https://www.tensorflow.org/tutorials`, accessed: 28.05.2023
[4] `https://github.com/tensorflow/tensorflow`, accessed: 28.05.2023
[5] `https://keras.io`, accessed: 28.05.2023
[6] `https://keras.io/examples/`, accessed: 28.05.2023
[7] `https://github.com/keras-team/keras`, accessed: 28.05.2023

PyTorch PyTorch is an open-source ML framework that “enables fast, flexible experimentation and efficient production through a user-friendly front-end, distributed training, and ecosystem of tools and libraries”[8]. The principles of PyTorch are[9]

- “Usability over Performance”
- “Simple Over Easy”
- “Python First With Best In Class Language Interoperability”

The library is open-source and available on GitHub and licensed under the BSD-3 license[10]. It consists of two high-level features:

- Tensor computation (like NumPy) with strong GPU acceleration
- Deep neural networks built on a tape-based autograd system

Generally, it is seen as a replacement for NumPy to use the power of GPUs and a deep learning research platform that provides maximum flexibility and speed. Nevertheless, it can be extended by NumPy, SciPy, and Cython [56].

Official tutorials are available for the categories “Image and Video”, “Audio”, “Text”, “Reinforcement Learning”, “Recommendation Systems”, and “Mobile” [55]. All in all, the main application areas are deep neural networks with GPU acceleration [56].

Scikit-Learn Scikit-Learn[11] is an open-source ML library that provides “simple and efficient tools for predictive data analysis” [60] and is built on NumPy, SciPy, and matplotlib. The library is licensed under the 3-clause BSD license.[12]

The official examples reveal many application areas of *Scikit-Learn*. Among those are “Decision Trees”, “Neural Networks”, and “Working with text documents”. Contrary to libraries like Keras or PyTorch, Scikit-learn does not have capabilities for deep learning or reinforcement learning.

Transformers *Transformers*[13] provide ML Application Programming Interfaces and tools to download and train models. A focus is put on the use of pre-trained models instead of training a model from scratch. The website emphasizes models for “Natural Language Processing”, “Computer Vision”, “Audio”, and “Modal”. *Transformers* support interoperability with PyTorch and TensorFlow to allow high flexibility [68].

The library is designed to reflect the standard NLP pipeline: data processing, model application, and prediction. A Transformers model is built up by three building blocks [73]:

- **Tokenizer**, converting input text to sparse index encodings

[8] `https://pytorch.org/features/`, accessed: 28.05.2023
[9] `https://pytorch.org/docs/stable/community/design.html`, accessed: 21.05.2023
[10] `https://github.com/pytorch/pytorch`, accessed: 27.05.2023
[11] `https://scikit-learn.org/stable/`, accessed: 29.05.2023
[12] `https://github.com/scikit-learn/scikit-learn`, accessed: 28.05.2023
[13] `https://huggingface.co/docs/transformers/index`, accessed: 28.05.2023

- **Transformer**, transforming encodings to contextual embeddings
- **Head**, that makes task-specific predictions based on the embeddings

Transformers is developed by the company Huggingface[14] and licensed under the Apache License 2.0[15].

2.1.3 Static Code Analysis

Code Smells

Booch et al. [9] aggregate best principles for object-oriented design in their book "Object-Oriented Analysis and Design with Applications". These principles are not limited to Object-Oriented systems and can be applied to other areas as well. Not following such guidelines could lead to so-called code smells. Code smells are "common poor code design choices that negatively affect the system and violate best practice or original design vision" [40]. Generally, code smells get introduced by developers implementing new features to meet new requirements. The new features break the original design, increase complexity and lower software quality [40]. by Martin Fowler et al. [44] describe finding the point in time when one should consider refactoring, with a kind of smell. Among others, a large class, long parameter lists or duplicated code can give off such a smell, indicating a perhaps needed refactoring [44].

A counterpart to code smells are data smells, introduced by Foidl et al. [22]. Foidl et al. [22] address a lack of research on data quality and potential data quality issues. Data smells share similar characteristics to code smell, like violating established best practices, which can lead to misinterpretations of software components. Just like code smells, data smells contribute to technical debt, but unlike code smells, data smells can represent bugs in the form of data errors [22].

In the following four code smells are introduced in detail based on their description provided by SonarQube[16].

Variable and function (parameters) naming convention Severity: Minor (2)
SonarQube states that using naming conventions enables effective collaborations in teams. Using naming conventions They use regular expressions to check the use of naming conventions. The default regular expression of SonarQube is based on the PEP-8 standard[17]. SonarQube allow the use of the so called "snake_case" and "CapWords" for class names (regex: `^_?([A-Z_][a-zA-Z0-9]*|[a-z_][a-z0-9_]$*)` and only "snake_case" (regex: `^[a-z_][a-z0-9_]$*)` for method names. No regex is provided for local variables, but the PEP-8 standard recommends "snake_case" and only allows "CapWords" when working with complying with components using other styles to ensure backward compatibility. Variable names of loops are excluded from this rule.

[14] https://huggingface.co/, accessed: 28.05.2023
[15] https://github.com/huggingface/transformers, accessed 28.05.2023
[16] https://rules.sonarsource.com/python/type/Code%20Smell, accessed: 11.06.2023
[17] https://peps.python.org/pep-0008/, accessed: 001.06.2023

Sections of code should not be commented out Severity: Major (3)
Commented-out code reduces readability and bloats programs. Old, removed code should be accessed through version control if needed.

Cognitive Complexity Severity: Critical (4)
Cognitive Complexity describes how hard to understand the control flow of a function is. A high Cognitive Complexity indicates functions that are difficult to maintain. SonarQube published a detailed white paper on how they calculate cognitive complexity[18]. Mccabe [45] published in 1967 Cyclomatic Complexity, a complexity measure of code which has been the de facto standard of measuring code complexity. SonarQube addresses issues when applying this measuring approach to modern languages since the original method was formulated in a Fortran environment. One shortcoming of the approach by Mccabe is that methods with a similar Cyclomatic Complexity are not necessarily similar difficult to maintain. SonarQube's complexity score consists of three rules, each increasing the complexity measure as stated in their white paper:

- Ignore shorthand formulation of multiple lines of code into one
- Increment for each break in the linear flow
- Increment for nested flow-breaking structures

Duplicated string literals Severity: Critical (4)
The duplication of string literals makes code updates susceptible to errors. All occurrences of the same string need to be updated. Using constants allows the string literal to be referenced from many places and updates only need to take place in a single location.

Project Smells

Implementing ML systems pose additional challenges, like data management, dependency management, and testing, compared to traditional software development [23]. Traditional software engineering established many tools to cope with those challenges. In the domain of ML systems, some tools from traditional software engineering are used, but they do not cover all aspects needed to be tackled. The concept of project smells was first described by van Oort et al. [51] and emerged from a previous study of Oort et al. [52], where they found out that many ML projects struggle with their code dependencies.** According to [51], code smells are only a small piece in the software quality puzzle. The term project smells resulted from this and describes a more holistic approach to code smells [51]. One important project smells deals with dependency management. Oort et al. [52] found major shortcoming in the dependency management of open-source ML projects on GitHub. Best practices to track and manage dependencies are Pipenv and Poetry as proposed by the official python packaging tutorial[19].

[18]https://www.sonarsource.com/docs/CognitiveComplexity.pdf, accessed: 10.06.2023
[19]https://packaging.python.org/en/latest/tutorials/managing-dependencies/#managing-dependencies, accessed: 13.06.2023

Combating Code Smells

The main approach to combating code smells is refactoring [40, 44]. In a tertiary systematic review of code smells and refactoring, Lacerda et al. [40] analysed the relationship between code smells and refactoring. They state that combating code smells is closely related to refactoring, despite differentiating in some key characteristics. Refactoring should be performed as early as possible to prevent the need for larger refactoring, interrupting daily work [40].

2.2 Related Work

This study encompasses different areas of research. After introducing related work covering most aspects of this study, related work for essential sub-parts of this study is presented. These sub-parts are "Mining Software Repositories", "Code Smell Analysis", "Project Smell Analysis" and "Project Categorisation". Following the related work, additional justification is provided as to why this study is necessary and of interest.

A related work similar to this study is the work by Jebnoun et al. [33]. Jebnoun et al. analysed open-source deep learning projects on GitHub for code smells. They analysed 59 deep learning and 59 traditional software projects and performed a comparative analysis between those two groups. Their main findings are the high frequency of the smells "long lambda expression", "long ternary conditionally expression", and "complex container comprehension" in deep learning projects.

Simmons et al. [64] analysed to which extent open-source data science projects follow code standards. They compared 1,048 data science to 1,099 non-data science projects using Pylint[20]. Their assumption is, that traditional code standards and software engineering conventions may not be appropriate for data science projects.

2.2.1 Mining Software Repositories

Gonzalez et al. [26] analysed 5,224 GitHub repositories that use Machine Learning with 4,101 repositories not using Machine Learning. Their goal was to analyse the "ML-Universe" since Machine Learning techniques have been made accessible to a wider audience in recent years, but not much is known about this domain from a software engineering perspective. A similarity to this study is performing software repository mining with the goal of finding active repositories that develop or apply Machine Learning.

Dilhara et al. [20] analysed 3,340 top-rated (stargazers count > 50) GitHub projects in the domain of Machine Learning. They focused on Machine Learning library usage and the evolution of ML systems. They performed a static code analysis using Jedi[21] to understand how developers use APIs of ML libraries. The main finding is the increasing ratio of ML projects in GitHub (2013: 1.75%, 2018: 49.63%) and the number of projects using at least two ML libraries. 40.1% of projects in their dataset use two or more ML libraries.

[20] https://www.pylint.org/, accessed: 15.06.2023

[21] https://jedi.readthedocs.io/en/latest/, accessed. 02.06.2023

2.2.2 Code Smell Analysis

Many studies in the past looked into code smells in traditional code [1, 21, 75] with a focus on the technical debt created by code smells. Oort et al. [52] analysed code smells in Machine Learning projects. They used pylint[22] to analyse 74 open-source python projects. The study revealed major flaws in dependency management of ML projects and that pylint cannot reliably check dependency imports in python code, resulting in many false positives. These flaws are according to Oort et al. major obstacles in the implementation of CI systems for ML projects.

Zhang et al. [77] looked into Machine Learning specific code which focuses on the ML part of a system, which only makes up a small part of the whole system [61]. They proposed a list of 22 machine learning-specific code smells, collected from various sources including GitHub and StackOverflow. Despite making up only a small part of a system, it is essential to avoid issues to prevent problems in the long run.

2.2.3 Project Smell Analysis

van Oort et al. [51] introduced the term "Project Smell" and implemented *mllint* to automatically analyse projects. They additionally evaluated their tool and the concept of project smells in a case study with ING. In a survey, they evaluated the perceived benefit of tools such as *mllint*. Participants find tools like *mllint* useful for enforcing best practices, maintaining consistency, and improving project quality. Since *mllint* and the related paper [51] were published in October 2022, no follow-up studies have been published.

In a previous study, Oort et al. [52] examined code smells in Machine Learning Projects. Nearly half of the projects they analysed had problems managing their code dependencies.

2.2.4 Project Categorisation

Sharma et al. [63] and Tavares et al. [66] analysed the README files of GitHub projects to extract topics. Tavares et al. [66] focused on the change of topics of Data Science projects caused by the global pandemic, starting in 2020. They found an increase in classification methods, which shows an interest in using Data Science to classify, identify, and detect data related to COVID-19. Additionally, the keyword "covid_19" appears and is standing out in 2020, compared to data from 2019. Drifts on the topics worked on can be detected and measured using Latent Dirichlet Allocation (LDA).

Sharma et al. [63] tried to create an automatic cataloguing system for GitHub repositories. On GitHub, the most popular projects (e.g., DevOps Frameworks, game engines, ...) get manually categorised. To enable a categorisation for all repositories, they introduced a cataloguing system. The system used README files of GitHub repositories as input and fed into a LDA model to infer categories. They evaluated the effectiveness of their approach via seven pairs of labellers. The classification method resulted in a precision of 0.7075, recall of 0.7038, and F-measure of 0.7057.

[22] https://pylint.readthedocs.io/en/latest/, accessed: 27.05.2023

The introduced related work covers many areas on the usage of ML libraries in open-source software. Differences between ML libraries are analysed infrequently and not on a large scale. The same applies to the real-world application areas of ML libraries.

Previous studies did not compare projects using ML libraries on a larger scale. This study aims to fill this gap and gain insights into the use of Machine Learning libraries in open-source software. Project smells did not receive follow-up studies since they were introduced by van Oort et al. [51]. This study is among the first follow-up studies to investigate project smells on a larger scale and use the toll *mllint*, developed to detect project smells.

ML libraries are often advertised as all-purpose libraries with no dedicated focus areas. Looking into the real-world application areas of ML libraries can reveal such dedicated use cases. This could benefit practitioners in choosing the best-fit library for their project and take special care of found correlations between ML libraries and accumulations of code or project smells. Those correlations also enable follow-up research to investigate causal relationships.

3 Study Design

3.1 Research Questions

This study aims to provide a multilayered categorisation of ML projects and a comparison with the specifications of the libraries used in open-source projects. Categorised ML projects are analysed to determine if a category (or a combination of several categories) can be associated with a significant accumulation of code or project smells. Those associations (e.g., "building ML applications with PyTorch is associated with more code smells than doing the same with TensorFlow") can generate hypotheses that may then be further examined in future confirmatory research. The following two research questions represent the main research directions:

RQ1: What is the distribution of application areas of machine learning libraries in open-source projects?

RQ2: How is the choice of a machine learning library associated with the types and frequency of code and project smells in open-source software?

3.2 Overall Strategy

The overall strategy for this study is divided into three main tasks, which are visualised with their sub-steps in figure 3.1.

1. ML repository mining
2. Machine Learning repository categorisation
3. Code and project smell analysis

The second and third tasks are dependent on the first one because the Repository Mining task collects the dataset needed for the categorisation and the code and project smell analysis. The following section explains the study design in detail and how the three main tasks mentioned above and their subtasks are performed.

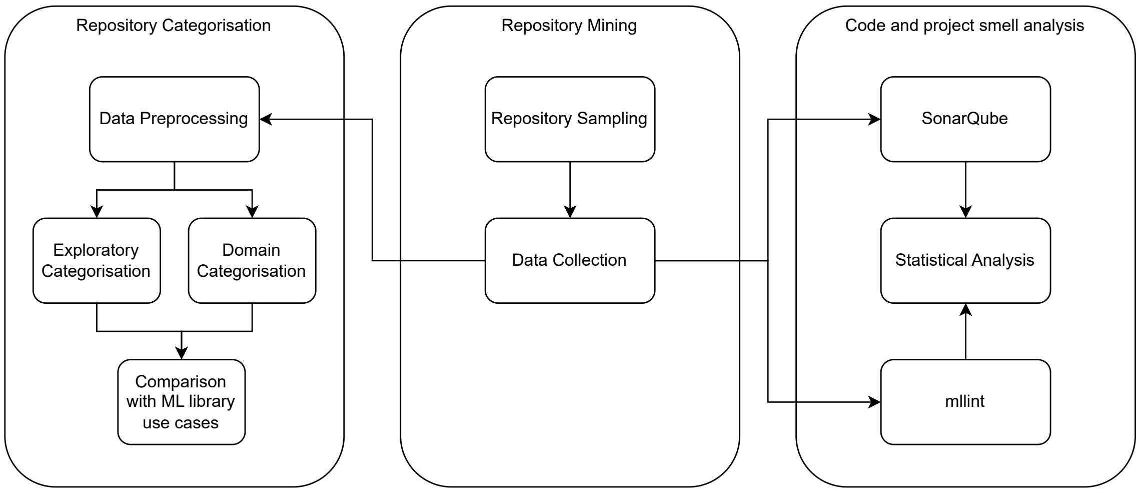

Figure 3.1: Flowchart of the overall strategy of this study comprised of three main tasks with their sub-tasks

3.2.1 Repository Mining

Mining Software Repositories (MSR) describes the analysis of data available in software repositories [28]. With repository mining the dataset, for further analysis, is collected from GitHub. GitHub[1] describes itself as "The complete developer platform to build, scale, and deliver secure software"[2]. Repositories must meet certain criteria to be included. The Repository Mining task is divided into the following subtasks:

Repository Sampling get all repository names that meet certain filter criteria

Data Collection acquire the dataset and apply additional filter criteria

3.2.2 Machine Learning Repository Categorisation

The goal of the project categorisation is to find categories or domains in which ML libraries are used. This category will be determined using NLP algorithms and a ML model which is trained to classify the domain category of software projects. Natural Language Processing (NLP) is an approach to effectively discover the knowledge in text, like a human. Common application areas of NLP are among others *Information Retrieval*, *automatic language translation*, *Question-Answering* and *Text generation/dialogues* [14]. Afterwards, the classifications are compared with the official documents of the libraries to find differences and similarities. The repositories will be categorised in domain categories and, exploratory, into topics prominent in the project's README files.

[1]`https://github.com/`, accessed: 23.05.2023
[2]`https://github.com/about`, accessed: 23.05.2023

Domain Categories Borges et al. [10] introduced domain categories for software projects. They analysed factors that determine the success of public GitHub repositories and created the following six domain categories: “Application & System Software”, “Documentation”, “Non-Web Libraries & Frameworks”, “Software Tools” and “Web Libraries & Frameworks”. These domain categories are not adapted to the field of Machine Learning, but may reflect a tendency of prominent application areas. For this study, an automatic approach introduced by Zanartu et al. [74] is used. Zanartu et al. [74] designed an automated classifier to automatically find the domain categories of open-source software projects. In this study, this classifier is used to classify the collected dataset. The classifier in [74] achieves the following performance:

Application Domain	Precision	Recall	F1 Score
Application & System Software	0.73	0.59	0.65
Documentation	0.84	0.74	0.79
Non-Web Libs & Frameworks	0.76	0.77	0.77
Software Tools	0.72	0.66	0.68
Web Libs & Frameworks	0.74	0.86	0.80

Table 3.1: Performance of automatic Domain Classifier [74]

The authors state that performance dropped when they applied the classifier to a dataset with less popular repositories. Therefore, in this study, this classifier will only be used in combination with exploratory methods. A sole reliance on this classifier cannot be justified due to the performance drops for less popular repositories. Finding exploratory categories can remove uncertainties and create new categories.

Exploratory Categorisation Exploratory Categorisation will be used to find answers to questions like "What kind of projects get implemented using PyTorch?". To analyse the application areas of ML libraries away from common categories, new categories and the assignment of common categories, will be done using an exploratory approach. Exploratory categorisation using NLP algorithms like Latent Dirichlet Allocation is used to find new categories, based on word clusters. Blei et al. [7] introduced Latent Dirichlet Allocation, “a generative probabilistic model for collections of discrete data such as text corpora” [7] in 2003. In this study, LDA will be used with a data set that contains all README files from all repositories for each machine learning library. Previous studies by Tavares et al. [66] and Sharma et al. [63] are used as inspiration for this study. They successfully applied LDA to README files of GitHub and extracted topic and categories.

3.2.3 Code and Project Smell Analysis

All collected repositories get analysed for code and project smells. A statistical significance test is used to find significant differences in the amount of code and project smells between different ML libraries. The goal of the analysis is to find associations between ML libraries and a significant accumulation of code smells. Additionally, the types of code smells get compared to find differences between libraries. This aims to gain insight if projects using a specific ML library are prone to a certain type of code or project smell.

In the following two section the acquisition methods of code and project smells are introduced.

Code Smell Analysis

SonarQube SonarQube will be used to find Code Smells in all repositories of the acquired dataset. SonarQube[3] is a static code analysis tool. It supports 30+ programming languages, including python. SonarQube states it enables consistent and reliable deployment of clean code [62]. SonarQube searches for 107 python code smells in projects, each with a severity measure.

Code Smell severities SonarQube provides a severity for each Code Smell it may detect[4]. Measuring the potential impact of a Code Smell is hard and since the developers of SonarQube are of that they adjusted them in 2016[5]. Code Smells in SonarQube are divided into five categories

1. Info (Severity 1)
2. Minor (Severity 2)
3. Major (Severity 3)
4. Critical (Severity 4)
5. Blocker (Severity 5)

Smells with Severity five and one are less frequent.

Several studies [41, 54, 69] analysed and criticised the severities provided by SonarQube. Lenarduzzi et al. [41] state, that instead of relying on the classification of Code Smells by SonarQube, “Performing a historical analysis of their project and classifying the actual change-and fault-proneness of their code” [41] is recommended. Changing the severity of code smells for every project may improve the informative value on how to fix those smell on a project basis. But this leads to a less informative comparison between projects and libraries. Therefore Code Smell severities of SonarQube will be used in this study.

Project Smell Analysis

Projects smells are analysed using *mllint*. *mllint* is a Open-Source static analysis tool, developed by van Oort [50] and further discussed by van Oort et al. [51]. *mllint* analyses the categories “Version Control”, “Dependency Management”, “Code Quality” and “Testing” to discover so called project smells. In their study, they additionally analysed the perception of *mllint*‘s rules in proof-of-concept and production-ready Machine Learning projects. A weighting of project smells cannot be derived from this analysis, because projects in this study cannot easily be categorised as proof-of-concept or production-ready projects. Therefore all collected project smells are weighted the same.

[3]`https://www.sonarsource.com/products/sonarqube/`, accessed: 23.05.2023

[4]`https://rules.sonarsource.com/python/type/Code%20Smell`, accessed: 20.05.2023

[5]`https://www.sonarsource.com/blog/we-are-adjusting-rules-severities/`, accessed: 20.05.2023

In this study, some rules of *mllint*'s categories are omitted because they are not applicable or hinder generalisations. In the category Version Control, some rules are applied to test the usage of Data Version Control (DVC). Barrak et al. [5] analysed DVC adaption in open-source projects. They found 391 projects on GitHub using DVC and analysed 25 of them in detail. 25% of their studied projects are past an initial exploration phase in using DVC. DVC is therefore not widely used and if projects use DVC, the usage is not sophisticated. *mllint* checks not only if DVC is used, but also if it is used like intended. Considering the non-use or the wrong use of DVC as project smells would lead to many false positives. DVC measures of *mllint* are not considered in this study.

To measure code quality, the configuration of code linters is checked and if they report issues or not. Since code quality and more precisely code smells are covered by SonarQube, only the rules checking for configured code linters are used. One rule of *mllint* checks whether a project uses git. This rule is omitted because all repositories use git due to their origin.

3.3 Repository Sampling

The GitHub REST API[6] will be used to crawl GitHub. Since not all repositories in the area of Machine Learning are of interest, filter criteria are used. Filtering criteria:

Content must be associated with Machine Learning

Size Must have size greater than 0 (KB)

Popularity Must have >=5 stars

Language main programming language must be python

Activity The last commit must have been within the last three months

Data Availability Repository data must be accessible via the GitHub API

To get repositories in the subject area of "Machine Learning", the following keywords are used as search terms in the GitHub API:

- machine learning
- ml
- ai
- deep learning
- deep neural networks
- neural networks
- data science
- natural language processing
- nlp

[6] `https://docs.github.com/en/rest?apiVersion=2022-11-28`, accessed: 28.05.2023

The keywords are inspired by trending topics on GitHub[7] and used as the **content** filter criteria. To retrieve the repositories, GitHub is crawled for each keyword separately. The keywords are used as search terms in the GitHub REST API. Those search terms are searched for in the project's title, description, and README file. For each keyword, the mentioned filter criteria are applied. After retrieving all repositories for each keyword, the repositories are combined into one dataset excluding duplicates.

Using stargazers as filter criteria for **popularity** has been discussed in several studies. Bogner et al. [8] refer to the work of Borges et al. [10] revealing that repositories on GitHub with five stars or less and forks have usually little or no activity. Searching for repositories with at least six stars yielded relevant repositories to compare TypeScript and JavaScript in the study of Bogner et al. Tavares et al. [66] searched for repositories with at least six stars to gain a dataset containing a general representation of repositories on GitHub without repositories with "minimum relevance to the community" [66]. They based their decision on Gonzalez et al. [26], who use filter criteria inspired by [27, 37, 49]. Gonzalez et al. [26] analysed the "State of the ML-Universe" on GitHub. As "popularity" filter, they searched for repositories with at least five stars or at least five forks. Stargazers are a good measure of the popularity of repositories [6, 10, 11, 32] and therefore used as the sole measure of popularity in this study. Just like in the study by Gonzalez et al. [26], the criteria are "purposefully lax to ensure the study represents the whole community and not just the 'top' repositories" [26].

This study is limited to python as a programming language, since it is the most used language for Machine Learning applications and allows to compare repositories with each other equally [26]. As filter criteria, python is chosen as mandatory the main programming language of a repository. Duplicates get excluded based on their GitHub clone url.

Data Filtering

In this study, no distinction is made between frameworks and libraries, and all ML software packages examined are referred to as libraries. Irrelevant repositories are repositories that do not use a Machine Learning library. A repository is kept in the dataset if the (generated) "requirements.txt" file contains at least one of the following ML libraries: "TensorFlow", "Keras", "PyTorch", "Scikit-learn", "Caffe/Pycaffe", "mxnet", "CNTK", "Chainer", "Theano", "Lasagne", "Pylearn2", "h2o". This list of Machine Learning libraries analysed is inspired by Wang et al. [70] and [34, 48]. It contains the most used ML libraries for python. To analyse and filter repositories, they all need to be cloned. Cloning reduces GitHub API calls and allows for a faster generation of "requirements.txt" files if they are missing.

3.4 Data Collection

This section highlights challenges in data collection and what measures have been taken to tackle them. Additionally, preprocessing steps for the analysis with NLP and preparations for analysis of code and project smells are presented.

[7] `https://github.com/topics`, accessed: 28.05.2023

3.4.1 Collecting Repositories from GitHub

To collect data from GitHub the GitHub REST API[8] is used. Using the GitHub REST API comes with drawbacks, which need to be addressed in the way the API is used.

Results limited to 1,000 repositories per request GitHub API returns a maximum of 1,000 results per request. Requesting all repositories with certain filter criteria that return more than 1,000 results, get truncated to 1,000. The remaining repositories cannot be accessed. To retrieve all repositories for a certain keyword, the requests need to be tailored that each response contains less than 1,000 repositories. This is achieved by dynamically adjusting the stargazers filter criteria. The minimum and maximum number of stars of projects for a certain keyword is retrieved. This star range then gets split up into smaller ranges, which each return less than 1,000 results. The ranges are found by a method inspired by binary search, which starts with the whole star range and reduces it until a range is found that returns less than 1,000 results. This procedure is repeated until the whole star range is covered with star ranges, each returning less than 1,000 results. Subsequently for each star range the repositories get can be requested without missing out on repositories.

API rate limits GitHub limits the rate of incoming requests. Using an Access Token of a GitHub Account increases the rate limiting, but does not remove it. To efficiently use all resources, new requests should be sent as early as the API cooldown is expired. This is achieved by getting the expiration time of the cooldown in the API response, informing the user about the cooldown, and sending a new request as soon as the cooldown is expired.

Large binary files Cloning repositories from GitHub automatically includes all binary files. To prevent too long cloning times and wasted hard drive space, large binary files are omitted using the parameter `--filter=blob:none`[9].

3.4.2 Filtering irrelevant repositories

Irrelevant repositories are repositories that do not use at least one ML library. To remove repositories that do not use a Machine Learning library, the following procedure will be used:

1. Clone repository
2. Analyse existing requirements file
3. Generate requirements file if needed
4. Remove repositories without ML library

[8] `https://docs.github.com/en/rest?apiVersion=2022-11-28`, accessed: 28.05.2023
[9] `https://docs.gitlab.com/ee/topics/git/partial_clone.html`, accessed 02.06.2023

After cloning a repository it is searched for a *requirements.txt* file. If a project contains a *requirements.txt* file, it will be used to determine if a project uses a ML library. If no *requirements.txt* file is present in the project files, it will be generated with the tool pipreqs[10] Repositories which do not use a ML library get removed from the dataset. The data acquired in this process are the existence of a *requirements.txt*, a list of used ML libraries and the number of used ML libraries.

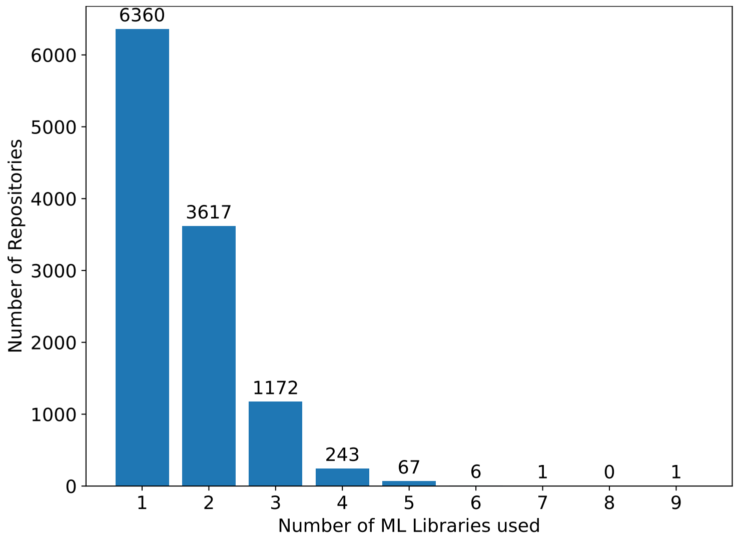

Figure 3.2: Distribution of the Number of Libraries used

To ensure comparability of the data, all repositories should use only one ML library. Figure 3.2 shows how many repositories use how many ML libraries. Around 55.45% use one library, whereas ~44.54% use two or more libraries. Dilhara et al. [20] observed that 40.10% percent of ML projects in their dataset use two or more ML libraries. This resembles the ~55.45% of projects using two or more libraries in this dataset. To ensure that projects are comparable with each other, this study focuses on the analysis of projects using only one ML library (6360 projects).

Library	Project Count
PyTorch	3,948
Scikit-learn	1,126
TensorFlow	1,060
TensorFlow & Keras	420
Transformers	131
Keras	60

Table 3.2: Number of projects for each ML library. Each library is the only ML library used in the repository, except for TensorFlow & Keras

After removing repositories using two or more libraries, the libraries “CNTK”, “Chainer”, “h2o”, “Theano”, and “mxnet” all were left with one to eleven and “Pylearn2”, “Pycaffe”,and “Lasagne” with no repositories. Those libraries are excluded from further analysis due to their low sample size. Table 3.2 shows the number of projects for every ML library included in the dataset.

[10]https://pypi.org/project/pipreqs/, accessed: 28.05.2023

In this study in addition to the ten libraries analysed, Keras and TensorFlow build a pair of libraries, analysed together. 2,536 projects use TensorFlow and Keras together. From those projects, those who use more than two ML libraries are removed. This results in an additional category of "TensorFlow & Keras", which includes 420 projects. Analysing the combination of Keras and TensorFlow together with the isolated analysis of Keras an TensorFlow may reveal additional insights for these two libraries, especially in comparison to using only TensorFlow.

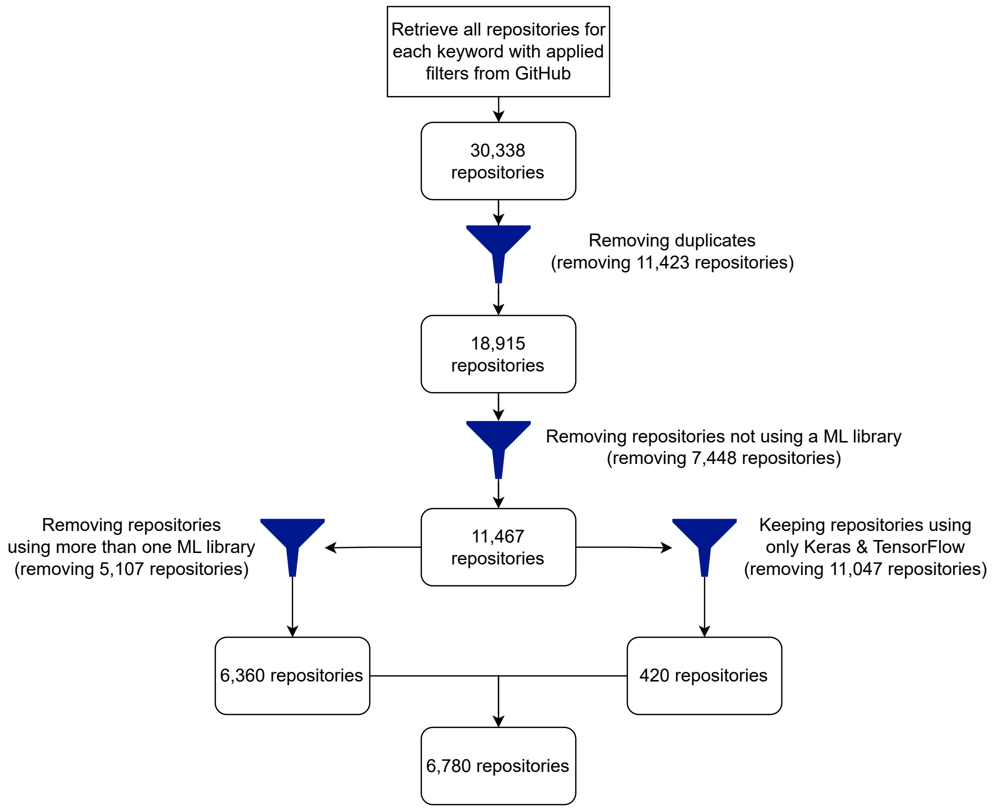
Retrieve all repositories for each keyword with applied filters from GitHub
30,338 repositories
Removing duplicates (removing 11,423 repositories)
18,915 repositories
Removing repositories not using a ML library (removing 7,448 repositories)
11,467 repositories
Removing repositories using more than one ML library (removing 5,107 repositories)
Keeping repositories using only Keras & TensorFlow (removing 11,047 repositories)
6,360 repositories
420 repositories
6,780 repositories

Figure 3.3: Flowchart of the data collection process with the number of repositories in each step

Figure 3.3 shows the data collection steps with the number of repositories removed in each filtering step. In this study the term "dataset" refers to the 6,780 projects containing repositories using only one ML library and repositories using TensorFlow & Keras exclusively.

3.4.3 Machine Learning Libraries

To gain insights on the distribution of combinations of ML libraries, used in open-source projects, the dataset is analysed in an exploratory way. To get frequently used combinations of ML libraries, their combinations get counted and compared afterwards.

Keras and TensorFlow Since Keras is built upon TensorFlow, it can't be installed independently from TensorFlow. Using Keras in a python project and generating a *requirements.txt* file using "pip freeze" results in TensorFlow and Keras listed as requirements. If a project does not have a *requirements.txt*, it will be automatically generated using *pipreqs*. As long as in the python code only Keras is imported and used, *pipreqs* will generate a *requirements.txt* file containing only Keras and not TensorFlow.

3.4.4 Data Preprocessing

Data Preprocessing for NLP To pre-process the README files to be usable by the LDA, an approach inspired by Chowdhary [14] is performed. Pre-processing consists of the following steps:

1. Removal of nearly empty README files: README files with a file size smaller than 2,000 bytes get removed. This ensures that all README files contain at least one or two sentences.
2. Removal of non-english README files: To remove README files written in a language other than english, the python library lingua[11] is used.
3. Removal of non-ASCII symbols
4. Removal of stop word: Stop words are words like “always”, “am”, and “among”. A list of stop words was used in [66] and published on GitHub[12]. Since this list already proved to be sufficient in filtering stop words in README files, it was used in this study.
5. Removal of numbers
6. Removal of frozen symbols (e.g., brackets)
7. Removal of URLs: To remove URLs, a regular expression is used.
8. Removal of single characters: README files sometimes contain, for example, inline code samples or html fragments that can’t be removed completely using the above-mentioned filters. Therefore, the removal of single characters is necessary.

Code and Project Smell Analysis To analyse code smells with SonarQube and project smells with *mllint*, each project needs to be cloned. To run a SonarQube analysis for every project, SonarScanner[13] was used. SonarScanner and mllint do not need preprocessing steps and only need a cloned repository to perform their analysis. SonarQube and a PostgreSQL database were used in a Docker[14] container. *mllint* creates for a repository a markdown document containing all collected metrics and tips to reduce Project Smells for the given repository. Since this data needs to be easily accessible for further analysis, *mllint*’s source code must be extended to export the numerical values for each analysed repository. The adjusted *mllint* source code appends the analysis results of each project to a *.csv* file. This allows for straightforward access to data.

[11] `https://github.com/pemistahl/lingua-py`, accessed: 04.06.2023
[12] `https://github.com/amaipy/lda_topics_metaheuristics/blob/master/stopwords.txt`, accessed: 03.06.2023
[13] `https://docs.sonarqube.org/9.8/analyzing-source-code/scanners/sonarscanner/`, accessed: 04.06.2023
[14] `https://www.docker.com`, accessed: 25.05.2023

Outliers are observations of a dataset that are different from the majority. Those observations can be errors, be a member of a different population or be recorded in other circumstances. Outliers don't fit well to the model and should be therefore detected [57]. Aguinis et al. [2] performed a literature review of 46 methodological resources addressing outliers. One recommendation is to use a visual tool first and afterwards apply quantitative measures.

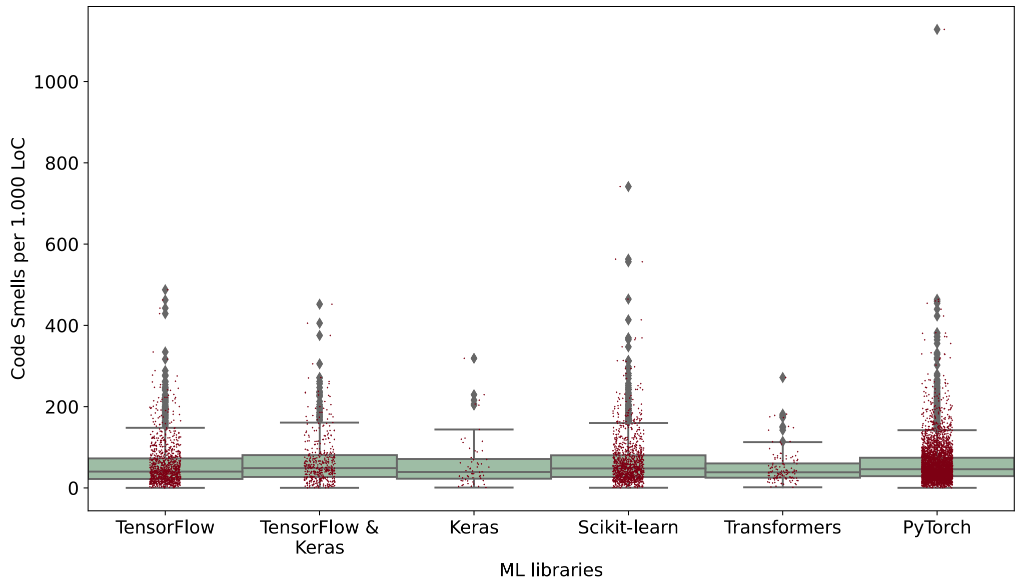

Figure 3.4: Boxplot of code smells per 1,000 LoC for each ML library - outliers not removed

Figure 3.4 shows code smells per 1,000 LoC as a boxplot. Following the recommendation of Aguinis et al. [2] to detect outliers, plotting the data reveals some observations sticking out, especially for PyTorch and Scikit-learn. In the following Interquartile Range and z-score, two common methods to remove outliers, are applied to the dataset and compared [57]. A recommended cutoff is that potential outliers are observations in the bottom and top 2.5% [2].

z-score is a classical rule to detect outliers [57]. It describes for each observation, how many standard deviations an observation lies below or above the population mean [16].

(3.1) $z_i = (x_i - \tilde{x})/s$

The z-score is calculated for each observation using equation 3.1. s is the standard deviation, x the mean of the population and x_i an observation [15, 57]. As threshold z-score 2.5 is chosen, inspired by [2, 57].

Library	Project Count with outliers	removed projects with IQR	removed projects with z-score
TensorFlow	1061	6.03% (64)	3.30% (35)
PyTorch	3953	5.31% (203)	2.45% (97)
Keras	60	8.33% (5)	3.33% (2)
Scikit-learn	1126	7.28% (82)	2.49% (28)
Transformers	131	7.63% (10)	3.82% (5)
TensorFlow & Keras	420	7.86% (33)	2.62% (11)

Table 3.3: SonarQube data outlier removal using z-score and IQR

Table 3.3 shows the number of observations classified as outliers using z-score and IQR. In this case, a datapoint is an outlier if its z-score exceeds 2.5. Using IQR to detect outliers removes over 5% of data from the dataset. For Keras even more than 8% and for Scikit-learn and Transformer more than 7%. This exceeds the recommended cutoff of 5% by Aguinis et al. [2]. Compared to the outlier removal using IQR, a z-score threshold of 2.5 removes fewer outliers and fits the recommendation by Rousseeuw and Hubert [57] and is therefore used as a method to remove outliers.

3.4.5 Project Categorisation

Exploratory Categorisation with LDA The domain classifier by Zanartu et al. [74] was used to automatically categorise all repositories into domain categories. Zanartu et al. [74] published their classifier, dataset and code to reproduce their study[15]. The dataset of this study needs to be adjusted to fit their model. This includes collecting additional data from GitHub, which is not needed for the other analysis in this study.

LDA Configuration After applying data preprocessing steps described in section 3.4.4, the most and least frequent words are removed to ensure finding distinct categories. Words that occur in many repositories, hinder finding distinct categories. Words occurring in more than 20% of the repositories or in less than 10 repositories are removed. In this study the LDA implementation[16] of the gensim topic modelling library is used. The overall goal is to find five to ten categories, respectively application areas, for each ML library. The used LDA implementation allows the specification of various parameters. Important parameters, used by the related work by Sharma et al. [63] and Tavares et al. [66], are the number of topics and the number of passes through the samples. For the initial runs, 10 topics and 50 passes are chosen, inspired by [63, 66]. For ML libraries with a small sample size, the LDA may not terminate and therefore not return associated keywords. The number of searched topics is reduced until the LDA terminates. At the same time, the upper and lower thresholds for words are adjusted. The tool pyLDAvis[17] is used to explore topics, find frequent words and verify the chosen LDA parameters.

3.5 Analysis

The following chapter presents the analysis methods used. It is divided into the analysis of the distribution of ML libraries, code and project smell analysis and the project categorisations.

3.5.1 Project Categorisation

The project categorisation analysis is divided into two parts. Domain categorisation shows the results of the used domain classification model by Zanartu et al. [74]. Exploratory Categorisation aims to find application areas of ML libraries by analysing README files.

[15]https://zenodo.org/record/6423599, accessed: 20.05.2023
[16]https://radimrehurek.com/gensim/models/ldamodel.html, accessed: 04.06.2023
[17]https://pypi.org/project/pyLDAvis/, accessed: 15.06.2023

Domain Categorisation

The categorisation resulting from the inference using the model introduced by Zanartu et al. [74], is compared to application areas stated by ML library publishers. Differences in the share of repositories classified as for example “Software Tool” and “Application Software” may give insights on application areas of the used ML libraries.

Exploratory Categorisation

Since the number of projects for the libraries “Chainer”, “mxnet”, “h2o”, “Caffee”, and “Theano”, “Keras” is low, adjustments need to be made in the LDA configuration. Reducing the amount of topics and accordingly adjusting preprocessing steps, lead to a successful completion for “Keras”. Lowering the amount of topics to only one was successful for “Chainer”, and “Theano”. For “h2o”, “Lasagne”, “Pycaffe”, and “Pylearn2” no configuration has been found to run LDA successful.

LDA results in topics which contain words and their probability of occurring in the context of the topic. Those topics need to be classified manually. To find categories and umbrella terms for the topics, each topic gets manually searched for keywords which give an indication of a learning method (e.g., reinforcement learning), application area (e.g., healthcare), or machine learning type (e.g., supervised learning). Existing topics to fit the topics to are inspired by [34, 48, 59, 76]

Independent of the related keywords provided by the LDA, the most common words across all repositories of each library get analysed. They can give an additional indication of topics or application areas, detached from the division of words into topics by the LDA.

3.5.2 Code and Project Smell Analysis

To answer RQ2, a statistical significance test is performed to find significant differences between ML libraries regarding their code and project smells. Additionally, the most prominent types of code and project smell are analysed and compared. As part of the exploratory aspects, correlations among certain properties of the GitHub repositories and code smells in the dataset are revealed. The following statistical analysis methods are applied for both code and project smells.

Statistical Analysis

To further analyse code smells and test for statistical significance, outliers may need to be removed from the dataset.

The goal of the statistical analysis is to check whether there are pairs of ML libraries with significant differences regarding their average code smells per 1,000 LoC or regarding their average *mmllint* score. To test for differences among more than two samples, ANOVA and Kruskal Wallis test are established methods. These tests have certain assumptions that need to be fulfilled. Tests, testing for those assumptions have assumptions themselves. In the following, the used hypothesis tests are introduced and their purpose of the use is presented.

To check if the samples follow a **normal distribution** a test [18] based on work by D'Agostino and Pearson [17, 18] is used.

To test, whether the variances of the sample have **homogeneous variances**, the Brown-Forsythe test is used. Using the Brown-Forsythe test is a better alternative to Bartletts test, if the samples do not follow a normal distribution [12, 29].

Homogeneous variance is an assumption needed to apply a **Kruskal-Wallis** test. Kruskal-Wallis test tests if there are samples which have a significant difference regarding their mean. A Kruskal-Wallis test can be applied instead of an ANOVA because it can work with non-near-equal variances [30]. Additionally, Kruskal-Wallis Test performed better for non-symmetrical distributions in a power analysis by Hecke [30] and does not require the samples to have homogeneous variances [30, 46].

Since the Kruskal-Wallis test only shows that there are significant differences between the samples, but not exactly which pairs are significantly different, they need to be followed up by a pairwise comparison. **Tukey's HSD Pairwise Group Comparison** compares each sample's mean using a statistical model. Since all samples are compared using the same sampling distribution, it is a more conservative approach [58].

Type I Error Correction Since we want to compare five samples with each other, the tests are prone to a type I error. The type I error describes that the more tests are performed, the more likely it is to find rare events and therefore reject the null hypothesis [58]. To reduce the possibility of type I errors, a p-value correction is performed.

Holm-Bonferroni Correction Bonferroni correction is a procedure to reduce type I errors when performing multiple tests by multiplying the number of tests performed with each probability. Holm [31] introduced in 1979 a more powerful sequential version of the Bonferroni correction. Holm's version is applied to the p-values of the tests after they are performed. To mitigate type I error Holm-Bonferroni Correction is used. One drawback of the Holm-Bonferroni Correction is the resulting increased possibility of a type II error [58]. How to balance type I and type II errors is "in most cases, neither clear nor generally agreed upon" [13].

[18] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.normaltest.html, accessed: 14.06.2023

4 Results

This chapter contains the results of this study. It is divided into general findings of the usage of ML libraries in open-source software, followed by application areas of ML libraries and the prevalence of code and project smells.

4.1 Machine Learning Libraries

Around 55.5% of the examined repositories use one library, whereas ~44.5% use two or more libraries. Dilhara et al. [20] observed that 40.1% percent of ML projects in their dataset use two or more ML libraries. This resembles the ~55.5% of projects using two or more libraries in this dataset. To ensure that projects are comparable with each other, this study focuses on the analysis of projects using only one ML library (6,360 projects).

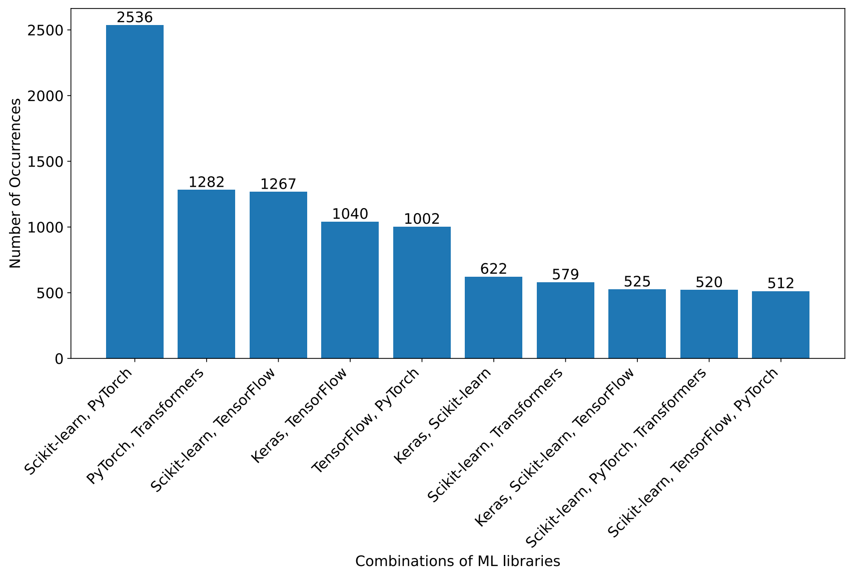

Figure 4.1: Top 10 occurrences of libraries used together - pairs are subsets of all combinations of ML libraries used

The top 10 most used combinations of ML libraries are visualised in figure 4.1. 2,536 projects use Scikit-learn and PyTorch together with potential additional libraries. This is the pair of ML libraries occurring the most in the dataset. Three or more libraries are used by just above 12.9% of repositories. A manual investigation on a random basis of projects using four or more libraries revealed that many of them are repositories containing code associated with published research. In many cases, not all libraries are used to create ML models, but selected functions and libraries are used as a complement to a main library used to create and train a ML model. For example, a repository containing code for a paper on sequence labeling[1] uses PyTorch to create and train a ML model, functions provided by Scikit-learn for mean calculations and Transformers to tokenize input sentences with a pre-trained model.

4.2 Application Areas of Machine Learning Libraries (RQ1)

This section shows the results of the analysis of application areas of Machine Learning libraries. It is divided into exploratory categorisation and domain categorisation.

4.2.1 Exploratory Categorisation

In the following, the results of the performed exploratory categorisation using LDA are presented. The topics of the found connected keywords by the LDA are compared to the official application areas as stated by the library publisher. The found application areas are listed with an excerpt of important keywords indicating the application area. Appendix A.4 contains wordclouds for all analysed ML libraries. Including all wordclouds in this chapter would inflate it and hinder readability and clarity.

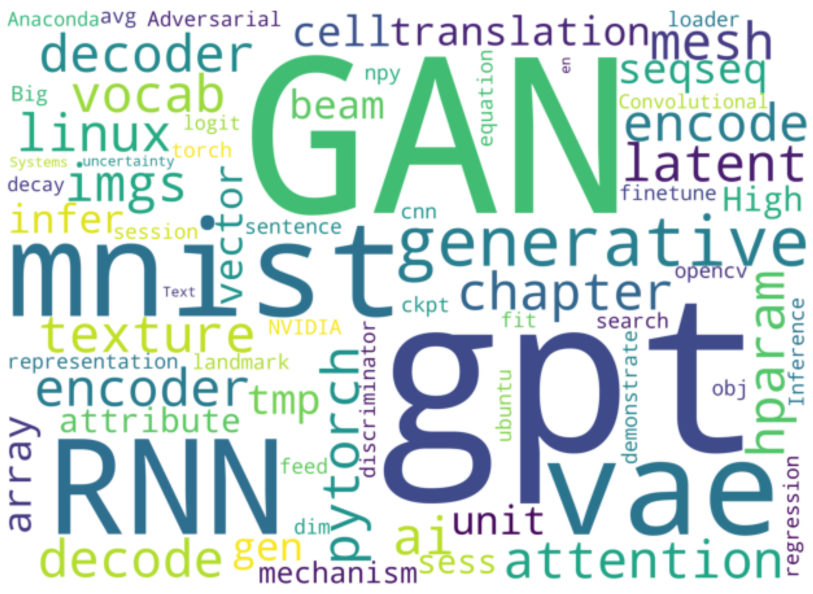

Figure 4.2: TensorFlow wordcloud for Topic “NLP”

Open-source projects using **TensorFlow** are connected to application areas of “Reinforcement Learning”, “Object Recognition”, “NLP / BERT”, and “TPU”. Figure 4.2 shows a wordcloud containing the most frequent keywords for Topic “NLP” of the TensorFlow library. Application areas like “GAN”, “NLP / BERT”, “Reinforcement Learning” and “GAN” represent key application

[1] https://github.com/Alibaba-NLP/AIN, accessed: 12.06.2023

areas of TensorFlow. “Vision” is also represented in the dataset, whereas a focus on “audio” cannot be observed. The most prominent words across all topics are words related to “Reinforcement Learning” (rl, GAN, reward, agent, policy). Another field for TensorFlow is Tensor Processing Unit (TPU). TPUs are custom-developed integrated circuits developed by Google[2] to accelerate ML compute loads. Since TensorFlow is developed by Google, it suggests itself that TPUs are only mentioned in TensorFlow projects with a greater frequency.

According to the exploratory categorisation, **Keras** is associated with the application areas of video detection, language inference and CNN. Prominent words for all found topics are “segmentation”, “CNN”, “video”, “detection”, “generator”, and “loss”. Compared with the data stated by the publishers of Keras (see section 2), the application areas “Computer Vision” and “Natural Language Processing” overlap. “Reinforcement Learing”, “Generative Deep Learning”, and “Audio Data” are stated as important application areas by Keras, but not reflected in this dataset. Since this dataset only consists of a subset of Keras projects, it may not be representative.

The combined category **TensorFlow & Keras** is associated with the topics “YOLO”, “Reinforcement Learning”, “COCO”, “Region Annotation”. Some word combinations found by LDA for TensorFlow & Keras cannot be clearly classified with an overarching topic, which is why those combinations are discarded. A comparison to the intended application areas can only be done with the separate application areas of TensorFlow and Keras as stated by their publisher. TensorFlow & Keras are associated with application areas found in TensorFlow. Overlapping thematic areas are “Reinforcement Learning” and “Object Recognition”. Unique topics for TensorFlow are TPU and BERT. BERT is a language representation model, developed by Google [19].

The LDA found that **Scikit-learn** is associated with “natural language processing”, “camera image segmentation”, and “genome analysis”. Genome analysis is a category only found together with Scikit-learn, a topic not mentioned on the website of Scikit-learn or in public documents. Recent studies use Scikit-learn for genome analysis [25, 43, 53]. Official documents of Scikit-learn suggest a wide range of possible application areas without committing to a few explicit topics.

Transformers allow due to their possibility of using prebuilt ML models an application in various areas. One topic combines Automatic Content Extraction (ACE)[3], a project to develop text extraction programs and Recall-Oriented Understudy for Gisting Evaluation (ROGUE)[4], a set of metrics to evaluate machine translations and summarizations. This topic is named “Context Extraction”. Another topic contains keywords regarding speech processing and translation. This topic is classified as “Speech Translation” which is related to NLP

Application areas of **PyTorch** are according to the performed LDA “(camera) image processing”, “Neural network optimisation”, “NLP / BERT”, “audio processing”, and “reinforcement learning”. These topics are mostly in line with the official information of PyTorch. Only the application areas “Mobile” and “Recommendation Systems” are not represented in the found topics.

Table 4.1 shows a comparison of the exploratory found categories of repositories using specific ML libraries, with the “official” application areas as stated by the publisher of the libraries. The combination of TensorFlow and Keras is not documented in official documents separately from

[2]`https://cloud.google.com/tpu/docs/intro-to-tpu?hl=en`, accessed: 10.06.2023

[3]`https://www.ldc.upenn.edu/collaborations/past-projects/ace`, accessed: 14.06.2023

[4]`https://huggingface.co/spaces/evaluate-metric/rouge`, accessed: 14.06.2023

Library	Exploratory application areas	Overlapping application areas	"Official" application areas
TensorFlow	Reinforcement Learning Object Recognition NLP / BERT TPU GAN	Reinforcement Learning Vision Text (GAN)	Vision Text Audio Reinforcement Learning Generative
Keras	Segmentation Computer Vision NLP	Computer Vision NLP	Computer Vision NLP Generative Deep Learning Audio Reinforcement Learning
TensorFlow & Keras	YOLO Reinforcement Learning COCO Region Annotation	-	-
PyTorch	Image Processing Neural Network Optimisation NLP / BERT Audio Processing Reinforcement Learning	Image Audio Reinforcement Learning NLP	Image & Video Audio Text Reinforcement Learning Recommendation System Mobile
Scikit-learn	NLP Camera Image Segmentation Genome Analysis	NLP	Decision Trees Neural Networks Text / NLP
Transformers	Context Extraction Speech Translation (NLP)	NLP	NLP Computer Vision Audio Modal

Table 4.1: Comparison of exploratory found application areas and application areas stated by the publishers of the ML libraries

Keras or TensorFlow. Therefore no comparison to official documents of TensorFlow and Keras can be made. For all ML libraries at least one applications area overlaps with the official documents of the library. Application areas of TensorFlow found by the LDA overlap in many areas with areas stated on the TensorFlow website and GitHub repository. Especially "Reinforcement Learning" and "Vision" stand out. The exploratory categorisation of application areas for Scikit-learn and Transformers did not reveal several distinct categories. Comparing them to their official documents only found the overlapping area "NLP".

4.2.2 Domain Classification

Table 4.2 shows the inference on all repositories using the domain classifier by Zanartu et al. [74]. The row "All libraries" shows the inference for the complete dataset, including repositories using two or more ML libraries. The largest number of repositories (~82.45%) is categorised as

Library / Category	Documentation	Non-web libraries and frameworks	Software Tools	Application & System software	Web libraries and frameworks
TensorFlow	13.95% (148)	80.40% (853)	4.24% (45)	0.85% (9)	0.57% (6)
Scikit-learn	13.50% (152)	78.42% (883)	6.48% (73)	1.07% (12)	0.53% (6)
PyTorch	11.31% (447)	84.27% (3331)	3.16% (125)	0.89% (35)	0.38% (15)
TensorFlow & Keras	14.76% (62)	82.38% (346)	0.71% (3)	2.14% (9)	0.00% (0)
Transformers	14.50% (19)	80.15% (105)	3.82% (5)	0.76% (1)	0.76% (1)
Keras	13.34% (8)	78.34% (47)	5.00% (3)	1.67% (1)	1.67% (1)
All libraries	12.37% (839)	82.45% (5590)	3.76% (255)	0.99% (67)	0.43% (29)

Table 4.2: Number of repositories in each domain category by ML library

"Non-web libraries and frameworks". For no library large deviations from the share of categories of the whole dataset can be detected. This is also the case when including repositories using two or more libraries. Projects using Scikit-learn have the largest share of classifications as Software Tools. "Non-web library or framework". According to the category description by Borges et al. [10], this library contains a GUI and should therefore be classified as "Application Software".

RQ2: What is the distribution of application areas of machine learning frameworks in open-source projects?

For the majority of the analysed ML libraries, application areas of their use in open-source projects have been found. Many ML libraries are used in application areas that are in line with tutorials and recommendations from their official documents. In some cases LDA revealed additional application areas. For example, the analysis showed that Scikit-learn is used for the analysis of genomes, which is not mentioned in official documents. In addition to that, recently published research uses Scikit-learn to analyse genomes.
Classifying the domain category of repositories using ML libraries showed that the majority of the analysed repositories are classified as "Non-web libraries and frameworks". The distribution of domain categories across the analysed libraries is similar and in line with the findings of Borges et al. [10], that most open-source repositories can be classified as (Non-)web libraries and frameworks.

4.3 Code and project smells in open-Source ML software (RQ2)

This section provides analysis results of code and project smells. The answer to **RQ2: How is the choice of a Machine Learning libraries associated with types and frequency of Code and Project Smells in Open-Source Software?** is split into code and project smells.

4.3.1 Code Smells

Types of Code Smells

Library / Code Smell	Cognitive Complexity of functions should not be too high	Unused local variables should be removed	String literals should not be duplicated
TensorFlow	9.45%	7.85%	7.30%
Keras	8.81%	8.00%	5.90%
TensorFlow & Keras	7.10%	7.35%	5.58%
PyTorch	8.64%	8.83%	4.79%
Scikit-learn	6.97%	6.15%	6.24%
Transformers	15.06%	8.79%	12.00%

Table 4.3: Share of code smells of all code smells present, for each library. The three compared code smells are present in the top 10 code smells of all libraries

In this study, 76 of the possible 106 distinct code smells have been found in the dataset. The missing 30 code smells are listed in appendix A.3. Table 4.3 shows the percentage of three code smells for every analysed ML library. The three code smells are “Cognitive Complexity of functions should not be too high”, “Unused local variables should be removed”, and “String literals should not be duplicated”, which are all in the top-10 most common code smells for every library.

Figure 4.3 shows a heatmap of code smells that account for at least 2.5% of all code smells in the projects of at least one ML library. The most common code smells are shared among the libraries. Scikit-learn is correlated with more than four times the frequency of not applied naming conventions compared to Transformers. On the other hand, the frequency of code with high cognitive complexity is more than doubled in Transformers compared to Scikit-learn. The two smells “Variable and function naming convention” and “Sections of code should not be commented out” are prominent in all of the main six ML libraries. Transformers projects being the only ones among the six to be more prone to the smells “Cognitive Complexity” and “Duplicated string literals” Looking at the share of the smell *Cognitive Complexity* smells of Keras and other libraries, no big differences can be spotted. For this comparison, it needs to be considered, that there are only 60 projects in the dataset using only Keras. Comparing projects using only TensorFlow with projects using TensorFlow and Keras, the share of *cognitive load* is reduced by ~2.35%.

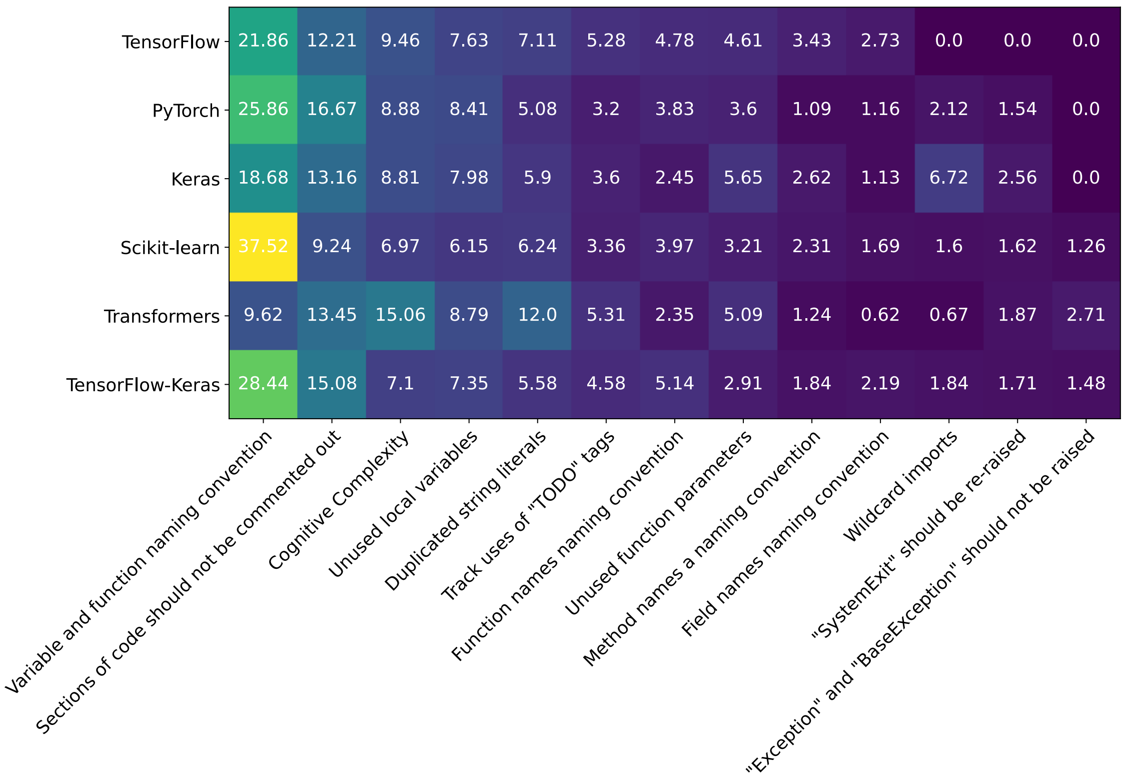

Figure 4.3: Heatmap for code smells with over 2.5% share of at least one library - numbers in percent

Frequency of Code Smells

To test, whether the variances of the sample have homogeneous variances, the Brown-Forsythe test is used. Brown-Forsythe Test is used as a better alternative than Bartletts test if the samples follow a non-normal distribution [12]. To test for a normal distribution a test[5] based on work by D'Agostino and Pearson [17][18] is used. For all tests, H_0 got rejected. All samples do not follow a normal distribution. H_0 states, that all samples are from populations with equal variances. Brown-Forsythe test results in a p-value of 1.564×10^{-8}, which suggest that the variances are not equal. H_0 is therefore rejected.

The samples do not have equal variances, which was already indicated by the sample variances displayed in table 4.4 and in the boxplots for the code smell per 1,000 LoC for every ML library, which can be found in appendix A.6.

Since ANOVA requires a homogeneous variance of the samples, it cannot be applied. Kruskal-Wallis test is instead applied to test whether a pair of ML libraries has a significantly different number of code smells per 1,000 lines of code, compared to another ML library. H_0 states, that there is no significant difference between the Machine Learning (ML) libraries regarding the mean number of

[5] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.normaltest.html, accessed: 14.06.2023

Library	Median	Mean	Variance
TensorFlow	38.97	50.15	1539.12
PyTorch	44.85	53.61	1187.88
Keras	36.96	53.58	2439.90
Scikit-learn	45.66	58.74	1991.59
Transformers	37.94	44.55	837.79
TensorFlow & Keras	47.27	58.88	2036.79

Table 4.4: Descriptive statistics for the code smell frequency per 1,000 LoC

code smells per 1,000 lines of code. Kruskal-Wallis test results in a p-value of 2.12×10^{-6} for the survival function of the chi-square distribution. H_0 gets rejected due to the p-value being smaller than 0.05.

Kruskal-Wallis does not indicate which pairs of the samples differ. To get pairs that differ, posthoc Tukeys HSD Pairwise Group Comparison tests are performed. H_0 for every compared pair states that there is no significant difference between them regarding the mean code smells per 1,000 lines of code.

Comparison pair	Statistic	p-value	Lower CI	Upper CI
(Transformers - TensorFlow)	-6.801	0.027	-21.154	7.551

Table 4.5: Excerpt - Tukey's HSD Pairwise Group Comparisons (95.0% Confidence Interval) for SonarQube Code Smells per 1,000 LoC with applied Bonferroni-Holm correction

Table 4.5 shows the results of Tukeys HSD Pairwise Group Comparison for the number of code smells per 1,000 LoC. The whole table with all results can be found in Appendix A.1. To remove the possibility of statistical error of Type-I, Holm-Bonferroni correction is applied to all p-values. Tukey's HSD Pairwise Group Comparison show a significant difference between Transformers and TensorFlow with a p-value of 0.027. H_0 for the comparison of TensorFlow and Transformers is rejected.

The statistical analysis of code smells in open-source repositories using ML libraries revealed significant differences between the use of Transformers and TensorFlow. On average, using TensorFlow results in 6.8 more code smells per 1,000 LoC compared to Transformers which is an increase of ~12.6%.

4.3.2 Project Smells

Types of Project Smells

This section shows the results of the analysis of types of project smells occurring in the studied projects.

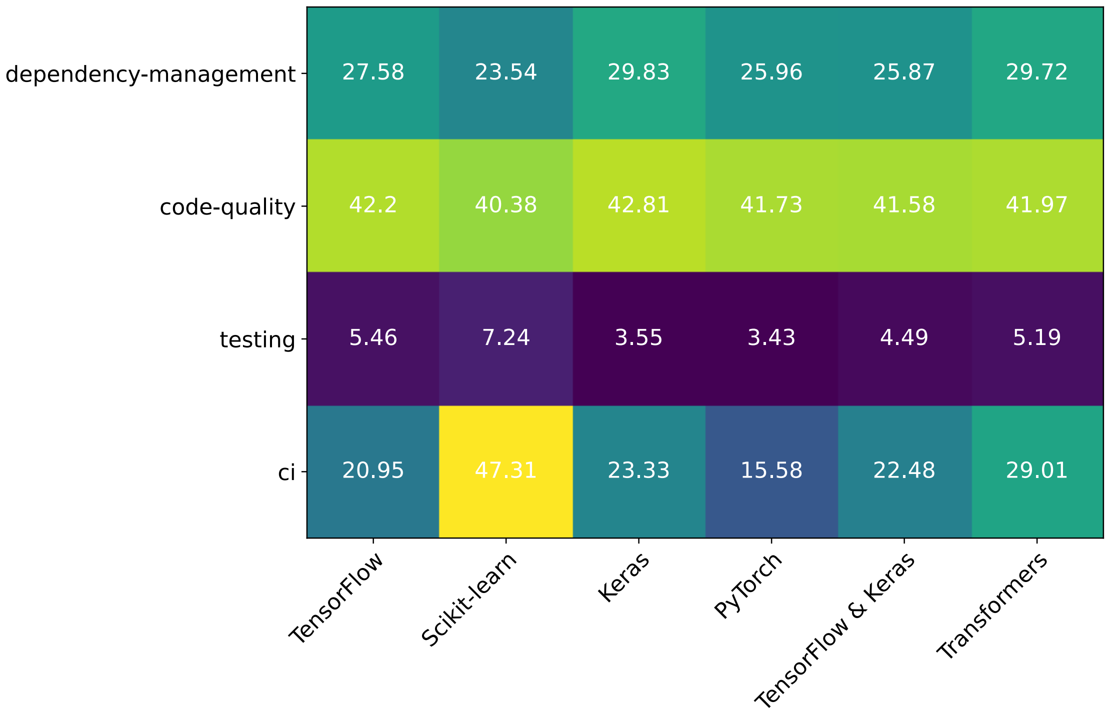

Figure 4.4: Heatmap of mllint scores for each project smell category

Figure 4.4 shows a heatmap for project smell categories analysed by *mllint*. Visually, Scikit-learn stands out with a high score in the “ci” category of *mllint*. The categories “Dependency Management”, “Code Quality” and “Testing” are mostly similar. The Continuous Integration (CI) rule of *mllint* checks whether a CI configuration file is present in the project. Since there is only one rule for the CI category, a CI-file is present (score: 100) or no CI-file is present (score: 0). The more frequent existence of CI-files in Scikit-learn projects may be due to the active distribution of CI-files in their project templates[6]

Contrary to that, less than 20% of every ML library uses Pipenv or Poetry to manage their dependencies. This additionally confirms the findings of Oort et al. [52], who found major barriers to the reproducibility and maintainability of ML projects due to deficient dependency management in python projects.

Frequency of Project Smells

Figure 4.5 shows the mllint average score for all projects with their respective Machine Learning library. The red dots show an excerpt of all data points. Visually, differences between PyTorch and all other libraries can be spotted regarding the mean and first quartile.

[6]https://github.com/scikit-learn-contrib/project-template/, accessed: 13.06.2023

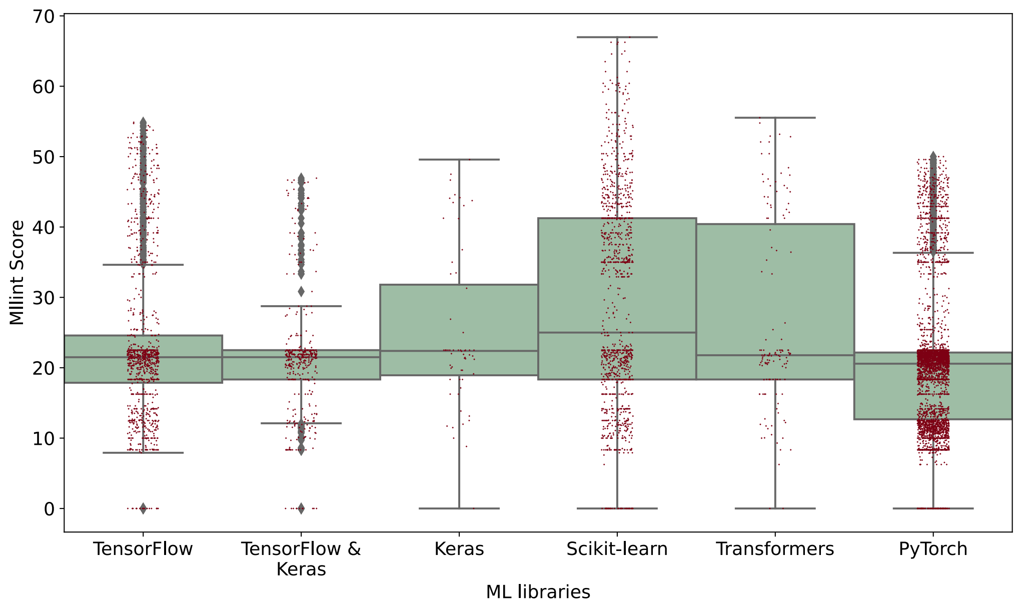

Figure 4.5: Average mllint score with outliers removed

Library	Median	Mean	Variance
TensorFlow	21.49	23.46	132.73
PyTorch	20.58	20.37	90.71
Keras	22.39	24.88	123.52
Scikit-learn	25.00	29.40	225.61
Transformers	21.78	26.47	177.41
TensorFlow & Keras	21.50	21.01	72.03

Table 4.6: Descriptive statistics for the average mllint score per used ML library

Table 4.6 shows the median, mean and variances of the *mllint* score of the analysed libraries. PyTorch has the lowest mean and median and the second lowest variance. This is in line with the visually detected differences.

To test, whether the variances of the sample are homogeneous variances, the Brown-Forsythe test is used. For all tests, except for Keras, H_0 got rejected. H_0 for Keras gets accepted with a p-value of 0.1134. The sample of Keras is therefore normally distributed.

Since the Brown-Forsythe test is robust for non-normal distributions, it can be applied. All samples, except Keras, do not follow a normal distribution. H_0 in the Brown-Forsythe test state, that all samples are from populations with equal variances. Brown-Forsythe test results in a p-value of 2.653×10^{-84}, which suggest that the variances are not equal. H_0 is therefore rejected.

Kruskal-Wallis test results in a p-value of 7.225×10^{-58} for the survival function of the chi-square distribution. H_0 gets rejected due to the p-value being smaller than 0.05. Therefore there is a combination of ML libraries with significant differences in their average *mllint* score.

To get pairs that differ, post hoc test Tukeys HSD Pairwise Group Comparison is performed.

Comparison pair	Statistic	p-value	Lower CI	Upper CI
(Scikit-Learn - TensorFlow)	5.543	0.000	4.045	7.040
(Scikit-Learn - PyTorch)	7.917	0.000	6.735	9.100
(Transformers - TensorFlow)	2.424	0.000	-0.817	5.665

Table 4.7: Excerpt - Tukey's HSD Pairwise Group Comparisons (95.0% Confidence Interval) for *mllint* project smells with applied Bonferroni-Holm correction

Table 4.7 shows all significant group comparisons performed with Tukey's HSD Pairwise Group Comparison. The whole table with all results can be found in appendix A.2. To reduce the possibility of Type I errors, the p-value correction by Bonferroni and Holm was applied. For three pairs, Tukey's HSD Pairwise Group Comparison returned a p-value smaller than 0.05. The three pairs are (Scikit-learn - TensorFlow), (Scikit-learn - PyTorch), and (Transformers - TensorFlow). All other tests returned a p-value greater than 0.05 which rejects H_0 for these comparisons.

The statistical analysis of project smell averages revealed three pairs of ML libraries which are correlated with having a higher average *mllint* score. Using Scikit-learn is correlated with a higher *mllint* score compared to TensorFlow and PyTorch. On average the *mllint* score of Scikit-learn is 5.5 points higher than TensorFlow and 7.9 points higher than PyTorch. This is an increase of ~25% compared to TensorFlow and ~44% compared to PyTorch.

The use of Transformers is correlated with an increasing *mllint* score of on average 2.4 points compared to TensorFlow. This is an increase of ~13%.

RQ2: How is the choice of a machine learning library associated with the types and frequency of code and project smells in open-source software?

Projects using Scikit-learn gained the highest *mllint* score in the category Continuous Integration. Scikit-learn is correlated with a higher *mllint* score than TensorFlow and PyTorch and Transformers is correlated with a higher *mllint* score than TensorFlow.
The most prominent types of code smells are violations of naming schemes, commenting out sections of code and having a high cognitive complexity of code sections. Violations of naming schemes are most prominent in projects using Scikit-learn and high cognitive complexity in projects using Transformers. Commenting out code sections is a smell, prominent across all analyses projects. This study revealed a correlation of the increase of code smells per 1,000 LoC when using TensorFlow instead of Transformers.

5 Discussion

5.1 Project Categorisation

This section discusses the answer to **RQ1: What is the distribution of application areas of machine learning libraries in open-source projects?**

Exploratory Categorisation Machine Learning are widely used across many areas. Trending topics in ML libraries and library-specific areas have been detected. TensorFlow projects focus on Reinforcement Learning, Natural Language Processing, and GANs. TPUs are a topic only prominent with TensorFlow. The dataset only contains 60 repositories for Keras. Such a low sample size may hinder the generalisability of the found categories.

The combination of TensorFlow and Keras yielded the categories YOLO, Reinforcement Learning, COCO, and region annotation. The combined libraries TensorFlow and Keras allow a comparison of projects using only Keras and projects using only TensorFlow. Keras focuses according to the analysed projects on video annotation and segmentation, language inference and CNNs. TensorFlow projects are mostly in the field of GANs, Reinforcement Learning, object recognition and NLPs / BERT. TensorFlow and Keras together are used for reinforcement learning, face recognition and audio. No connection between Keras projects and reinforcement learning could be found, despite various code examples for reinforcement learning on the Keras website. This could be due to the small sample size for projects only using Keras. Small differences between Keras, TensorFlow and the combination of both can be spotted with LDA. The most meaningful difference can be spotted between TensorFlow projects and projects using TensorFlow and Keras. TensorFlow projects have a focus on NLP, namely BERT which is not present in projects using Keras.

Domain Classification The domain classifications resulted in projects classified to over 80% as Non-web libraries & Frameworks. Zanartu et al. [74] used a dataset provided by Borges et al. [10] to train their model. Their dataset consists of 5,000 top-rated GitHub projects, manually labelled into domain categories. Looking at the distribution of the training data, a bias towards "Web Libraries and Frameworks" and "Non-Web Libraries and Frameworks" can be detected. Those two categories make up over 59% of their training data. "Software Tool" and "Application Software" make up 19.5% and 8.6%. The skew in the training data used by Zanartu et al. [74] is a direct result of the categories of top-rated GitHub projects. Despite the skewed training data, the observation that most projects using ML are not associated with "web" technologies in general, coincides with trending topics in ML which are all not related to web technologies [39]. The disproportionate amount of repositories classified as "Non-Web Libraries and Frameworks" could be explained by this skew in training data.

A manual random investigation of projects classified as “Non-web library and framework” identified that many repositories can at least partially be classified as libraries or frameworks. For example an adjusted version of the implementation of the popular ML model code2vec[1]. The adjusted version[2] provides an example and can therefore be used as an application. At the same time, the model can also be used as a kind of library that provides an interface on which new software can be built. Differentiating between framework and application software is not always clear. Still, repositories like a GUI to classify handwritten digits[3] is misclassified as “Non-web library and framework”.

As a consequence of the stated problem with the training data of the classifier, no substantial insight could be gained by using this classifier. Future work may investigate domain categories of ML repositories and lay a focus on their distribution in open-source software.

5.2 Code and Project Smells

This section discusses the answer to **RQ2: How is the choice of a machine learning library associated with the types and frequency of code and project smells in open-source software?**.

Code Smells Regarding code smells, many code smells are of the same type across ML libraries. The smells “Variable and naming convention”, “Sections of code should not be commented out”, “Cognitive Complexity”, and “Unused local variable” are the most prominent across the six analysed ML libraries.

Keras is described on the official website as library “following best practices for reducing cognitive load” and offering simple APIs. Yet the share of the code smell “Cognitive Complexity” is not lower than PyTorch, Scikit-learn or TensorFlow. The same applies to projects using TensorFlow & Keras. Investigating the code smell “Cognitive Complexity” in future can give insights into the effectiveness of such design principles. Additionally, future work could establish a causal link between the choice of ML library and the manifestation of “Cognitive Complexity”. Tukey’s HSD group comparison resulted in one pair with a significant difference regarding their mean.

The statistical analysis of code smells in open-source repositories using ML libraries revealed significant differences between the use of Transformers and TensorFlow. Using Transformers is correlated with a ~12.6% higher number of code smell per 1,000 LoC than using TensorFlow. Those two libraries are different in their use cases and application areas. Choosing one library over the other can not replace the other library fully and may not be applicable. The found correlation, therefore, applies less to practitioners but states a starting point for future research on library design and software engineering practices.

[1] `https://github.com/tech-srl/code2vec`, accessed: 16.06.2023
[2] `https://github.com/Kirili4ik/code2vec`, accessed: 16.06.2023
[3] `https://github.com/ardaakdere/HandwrittenDigitRecognizer`, accessed: 16.06.2023

Project Smells *mllint* checks for the use of Pipenv and Poetry and related to the Python Packaging User Guide[4]. In the context of dependency management, looking into the use of *requirements.txt* files across the dataset shows that the use of *requirement.txt* files differs significantly among projects using only one ML library. Of the 11,467 projects in the collected data set, 6,505 contain an existing *requirements.txt* file, while 4,962 do not. Of projects using only one ML library, ~55.68% have a *requirements.txt* file and ~58.04% of projects using two or more ML libraries. ~64.56% of TensorFlow projects, ~67.67% of Scikit-learn projects, ~46.67% of Keras projects, ~48.95% of PyTorch projects, ~80.24% of TensorFlow & Keras projects, and ~86.26% of Transformers project use a *requirements.txt* file to track their requirements. The usage in PyTorch is less common than in nearly every other library. The more frequent existence of CI-files in Scikit-learn projects may be due to the active distribution of CI-files in their project templates[5]. The remaining libraries do not distribute templates with integrated CI configuration files in that manner. Prominently placing CI files in templated and tutorials may encourage practitioners to use them. This study did not reveal a causal relationship between the use of CI files and their existence in project templates, but it stands to reason that this is one of the reasons they are used so often in projects using Scikit-learn.

The statistical analysis of code smells found by SonarQube found three pairs with significant differences regarding their average *mllint* score. Scikit-learn is associated with a ~25% higher *mllint* score than TensorFlow and a ~44% higher *mllint* score than PyTorch. The use of Transformers is correlated with an on average ~13% higher *mllint* score than TensorFlow.

Since using a ML library does likely not influence the use of software engineering best practices, those correlations do only pose as a starting point for further research. Investigating the effects of good practices in project templates in the use of for example CI, may be part of follow-up research.

5.3 Threats to Validity

Internal Threats to Validity

Severities of Code Smells The severities of code smells by SonarQube have received criticism in the past. Suggestions on ways to improve the valuation found in literature follow a per-project approach. Evaluating each project on its own to ensure a good valuation of each Code Smells, would limit the comparability of the projects with each other. Nevertheless, it is not sure, whether the severities of code smells used are a good measure for those code smells. Using severities provided by SonarQube ensures comparability which is essential in this study and its research questions.

[4]`https://packaging.python.org/en/latest/tutorials/managing-dependencies/#managing-dependencies`, accessed: 13.06.2023

[5]`https://github.com/scikit-learn-contrib/project-template/`, accessed: 13.06.2023

mllint's applicability to real-world systems *mllint* has only been used by one study by the same author as its developer. It has not gained many real-world uses like SonarQube. Flaws may be present in the approach of *mllint* resulting in skewed measurement of project smells in this study. Since *mllint* follows the project smell definition of the creators of the term code smell, *mllint* is the best and only tool available to automatically collect a measure representing project smells.

Sample Size A potential threat to validity is the sample size of repositories. Projects using only Keras may be susceptible to this validity thread. Separating TensorFlow and Keras, which is built upon TensorFlow can not be done reliably which resulted in a small sample size for projects using only Keras. Results of this category may therefore not be generalisable and applicable to all Keras projects.

For the other libraries, this validity threat applies partially due to the limited time frame of three months in which the last commit needed to be pushed. This purposefully removed projects, that are not actively developed or managed but reduced the overall sample size. Compared to related work analysing 6.840 repositories larger than many studies. The generalisability is therefore reduced to actively developed repositories.

External Threats to Validity

Readme files of repositories Maintainers of open-source projects may not maintain the README files of their projects. Those files could be not up-to-date, the project may not be sufficiently described or important keywords are missing. This could lead to a wrong categorisation of repositories using LDA and the automatic domain category classifier. The Domain Categorisation uses more data than only README files of projects. but is also affected by this.

Validity of the dataset This study only analysed open-source projects. Aranda and Venolia [4] claims that not all software development is done open-source. An application of the results to all software projects of ML projects can not be made. Additionally, jupyter notebooks are excluded from this study. Jupyter Notebooks play a key role in the development of ML software. Including Jupyter notebooks would introduce additional problems like analysing them, which is not yet by SonarQube and SonarScanner[6] and the high number of duplicated code snippets in Jupyter notebook on GitHub [36].

quelle jupyter notebooks used in DS, prototyping... -> not representable for ML in general -> therefore excluded

[6]https://portal.productboard.com/sonarsource/4-sonarlint/c/245-find-python-bugs-in-jupyter-notebook-code, accessed: 16.06.2023

6 Conclusion

6.1 Summary

Machine Learning systems are prone to technical debt and practitioners face additional challenges when implementing them. To gain insights on the type and frequency of code and project smells, 6,780 open-source repositories using a ML library have been analysed. In addition to that, the distribution of application areas among the repositories was investigated. To find categories and application areas with which the analysed ML libraries are associated, they were classified into domain categories using a ML model. Additionally, an exploratory categorisation using Latent Dirichlet Allocation was performed on README files to find application areas away from classical categories like domain categories.

Statistical analysis revealed that using TensorFlow as ML library is associated with ~12.6% more code smells per 1,000 LoC than using Transformers. For project smells, three pairs correlated with significant differences regarding their average *mllint* score have been found. Using Scikit-learn is correlated with a ~25% higher *mllint* score compared to TensorFlow and a ~44% higher *mllint* score than PyTorch. The use of Transformers is correlated with a ~13% higher average *mllint* score than TensorFlow.

The derivation of recommended actions from the found correlation cannot be done. Instead, they pose starting point for future research, investigating software engineering practices for developing AI-based systems or investigating the influence of API design on code quality.

Machine Learning libraries are often advertised as all-purpose libraries covering all aspects and varieties of implementation and use cases. Analysing README files of open-source Machine Learning repositories showed that ML libraries have different main application areas. These areas sometimes differ from the areas stated in the official documents of these libraries. The performed LDA found application areas in which specific libraries are used more often.

6.2 Benefits

Practitioners implementing a ML system with a library that is significantly more correlated with code or project smells, may take special care in looking for code and project smells and take additional measures to prevent them. Developers searching for a good-fitting API may use the found application areas in practice revealed by this study. This can lead to using a more appropriate library usage and therefore better code.

Additionally, this study revealed that nearly half of the analysed projects used two or more ML libraries, which confirms the findings of Oort et al. [52]. Different libraries are used for different parts in the development of ML systems, which may complicate the systems. Therefore, special care can be necessary when selecting the libraries needed.

This study shows that there are significant differences coming with the use of different Machine Learning libraries. The found application areas for each Machine Learning library overlap in many cases with areas describes in official documents belonging to the libraries. In some cases, differences and unique areas of ML libraries Therefore, by choosing a ML library for a project, more than only the information provided by the library developers needs to be considered.

6.3 Limitations

Since this study only revealed correlations, special care should be taken when using this data to derive best practices or guidelines in using ML libraries. The dataset is limited to active Python projects on GitHub. Although Python is the most popular programming language for implementing ML, the results cannot be generalised to the development of ML systems in general. Additionally, not all software development is done on GitHub and excluding Jupyter notebooks may reduce the validity of this study.

6.4 Future Work

Since this study only revealed correlations regarding the difference in code smells between ML libraries, analysing causal links is a field of potential future work. Pairs of ML libraries with significant differences in code or project smells may be analysed qualitatively and in detail. This could be done by focusing on API design and the complexity of library functions.

Furthermore, an investigation of practitioners, maintainers, and developers working with ML libraries can reveal differences in the types of users and formulate best practices to reduce code and project smell.

Bibliography

[1] M. Abbes, F. Khomh, Y. G. Guéhéneuc, G. Antoniol. “An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension”. In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* (2011), pp. 181–190. ISSN: 15345351. DOI: 10.1109/CSMR.2011.24. URL: https://www.researchgate.net/publication/221570070_An_Empirical_Study_of_the_Impact_of_Two_Antipatterns_Blob_and_Spaghetti_Code_On_Program_Comprehension (cit. on p. 11).

[2] H. Aguinis, R. K. Gottfredson, H. Joo. “Best-Practice Recommendations for Defining, Identifying, and Handling Outliers”. In: *Organizational Research Methods* 16 (2 Apr. 2013), pp. 270–301. ISSN: 15527425. DOI: 10.1177/1094428112470848/FORMAT/EPUB (cit. on pp. 23, 24).

[3] H. El-Amir, M. Hamdy. “Deep Learning Pipeline Building a Deep Learning Model with TensorFlow”. In: (2020). DOI: 10.1007/978-1-4842-5349-6. URL: https://doi.org/10.1007/978-1-4842-5349-6 (cit. on pp. 3–5).

[4] J. Aranda, G. Venolia. “The secret life of bugs: Going past the errors and omissions in software repositories”. In: *2009 IEEE 31st International Conference on Software Engineering* (2009). URL: https://www.academia.edu/464234/The_secret_life_of_bugs_Going_past_the_errors_and_omissions_in_software_repositories (cit. on p. 41).

[5] A. Barrak, E. E. Eghan, B. Adams. “On the Co-evolution of ML Pipelines and Source Code - Empirical Study of DVC Projects”. In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2021). DOI: 10.1109/SANER50967.2021.00046 (cit. on p. 17).

[6] T. F. Bissyande, F. Thung, D. Lo, L. Jiang, L. Reveillere. “Popularity, interoperability, and impact of programming languages in 100,000 open source projects”. In: *Proceedings - International Computer Software and Applications Conference* (2013), pp. 303–312. ISSN: 07303157. DOI: 10.1109/COMPSAC.2013.55 (cit. on p. 18).

[7] D. M. Blei, A. Y. Ng, J. B. Edu. “Latent Dirichlet Allocation Michael I. Jordan”. In: *Journal of Machine Learning Research* 3 (2003), pp. 993–1022 (cit. on p. 15).

[8] J. Bogner, R. Verdecchia, I. Gerostathopoulos. “Characterizing Technical Debt and Antipatterns in AI-Based Systems: A Systematic Mapping Study”. In: (Mar. 2021). URL: http://arxiv.org/abs/2103.09783 (cit. on p. 18).

[9] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. W. Newkirk, K. Houston, A. Brown, J. Conallen. *Object-oriented analysis and design with applications*. en. 3rd ed. The Addison-Wesley object technology series. Boston, MA: Addison Wesley, Mar. 2001 (cit. on p. 8).

[10] H. Borges, A. Hora, M. T. Valente. "Understanding the factors that impact the popularity of GitHub repositories". In: *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016* (Jan. 2017), pp. 334–344. DOI: 10.1109/ICSME.2016.31 (cit. on pp. 15, 18, 31, 38).

[11] H. Borges, M. T. Valente. "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform". In: *Journal of Systems and Software* 146 (Nov. 2018), pp. 112–129. DOI: 10.1016/j.jss.2018.09.016. URL: http://arxiv.org/abs/1811.07643%20http://dx.doi.org/10.1016/j.jss.2018.09.016 (cit. on p. 18).

[12] M. B. Brown, A. B. Forsythe. "Robust Tests for the Equality of Variances". In: *Journal of the American Statistical Association* 69 (346 June 1974), p. 364. ISSN: 01621459. DOI: 10.2307/2285659 (cit. on pp. 26, 33).

[13] R. J. Cabin, R. J. Mitchell. "To Bonferroni or Not to Bonferroni: When and How Are the Questions". In: *Source: Bulletin of the Ecological Society of America* 81 (3 2000), pp. 246–248 (cit. on p. 26).

[14] K. R. Chowdhary. *Fundamentals of Artificial Intelligence*. Springer, 2020. ISBN: 9788132239703. DOI: 10.1007/978-81-322-3972-7. URL: https://doi.org/10.1007/978-81-322-3972-7 (cit. on pp. 3–5, 14, 22).

[15] H. Chubb, J. M. Simpson. "The use of Z-scores in paediatric cardiology". In: *Annals of Pediatric Cardiology* 5 (2 July 2012), pp. 179–184. ISSN: 09742069. DOI: 10.4103/0974-2069.99622. URL: https://journals.lww.com/aopc/Fulltext/2012/05020/The_use_of_Z_scores_in_paediatric_cardiology.11.aspx (cit. on p. 23).

[16] B. Curtis, J. Sappidi, A. Szynkarski. *Estimating the Principal of an Application's Technical Debt*. IEEE Software, Dec. 2012 (cit. on p. 23).

[17] R. D'Agostino, E. S. Pearson. "Tests for departure from normality. Empirical results for the distributions of b2 and $\sqrt{b}1$". In: *Biometrika* 60 (3 1973), pp. 613–622. ISSN: 00063444. DOI: 10.2307/2335012 (cit. on pp. 26, 33).

[18] R. B. D'Agostino. "An Omnibus Test of Normality for Moderate and Large Size Samples". In: *Biometrika* 58 (2 Aug. 1971), p. 341. ISSN: 00063444. DOI: 10.2307/2334522 (cit. on pp. 26, 33).

[19] J. Devlin, M. W. Chang, K. Lee, K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference* 1 (Oct. 2018), pp. 4171–4186. URL: https://arxiv.org/abs/1810.04805v2 (cit. on p. 29).

[20] M. Dilhara, A. Ketkar, D. Dig. "Understanding Software-2.0: A Study of Machine Learning Library Usage and Evolution". In: *ACM Transactions on Software Engineering and Methodology* 30 (4 2021). DOI: 10.1145/3453478. URL: https://doi.org/10.1145/3453478 (cit. on pp. 6, 10, 20, 27).

[21] D. Falessi, A. Voegele. "Validating and prioritizing quality rules for managing technical debt: An industrial case study". In: *2015 IEEE 7th International Workshop on Managing Technical Debt, MTD 2015 - Proceedings* (Nov. 2015), pp. 41–48. DOI: 10.1109/MTD.2015.7332623 (cit. on p. 11).

[22] H. Foidl, M. Felderer, R. Ramler. “Data Smells: Categories, Causes and Consequences, and Detection of Suspicious Data in AI-based Systems”. In: *Proceedings - 1st International Conference on AI Engineering - Software Engineering for AI, CAIN 2022* (Mar. 2022), pp. 229–239. DOI: 10.48550/arxiv.2203.10384. URL: https://arxiv.org/abs/2203.10384v3 (cit. on p. 8).

[23] G. Giray. “A software engineering perspective on engineering machine learning systems: State of the art and challenges”. In: *Journal of Systems and Software* 180 (Oct. 2021), p. 111031. ISSN: 0164-1212. DOI: 10.1016/J.JSS.2021.111031 (cit. on pp. 1, 9).

[24] *Global explainable AI market revenues 2021-2030*. en. https://www.statista.com/statistics/1256246/worldwide-explainable-ai-market-revenues/. Accessed: 2023-5-28 (cit. on p. 1).

[25] “Global genome analysis reveals a vast and dynamic anellovirus landscape within the human virome”. In: *Cell Host & Microbe* 29 (8 Aug. 2021), 1305–1315.e6. ISSN: 1931-3128. DOI: 10.1016/J.CHOM.2021.07.001 (cit. on p. 29).

[26] D. Gonzalez, T. Zimmermann, N. Nagappan. “The State of the ML-universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub”. In: (2020). DOI: 10.1145/3379597.3387473. URL: https://doi.org/10.1145/3379597.3387473 (cit. on pp. 10, 18).

[27] G. Gousios, D. Spinellis. “Mining Software Engineering Data from GitHub”. In: (2017). DOI: 10.1109/ICSE-C.2017.164. URL: https://speakerdeck.com/gousiosg/working-effectively-with-pull-requests (cit. on p. 18).

[28] A. E. Hassan. “The road ahead for mining software repositories”. In: *Proceedings of the 2008 Frontiers of Software Maintenance, FoSM 2008* (2008), pp. 48–57. DOI: 10.1109/FOSM.2008.4659248 (cit. on p. 14).

[29] D. Hatchavanich. “A COMPARISON OF TYPE I ERROR AND POWER OF BARTLETT’S TEST, LEVENE’S TEST AND O’BRIEN’S TEST FOR HOMOGENEITY OF VARIANCE TESTS”. In: *Southeast Asian Journal of Sciences* 3 (2 2014), pp. 181–194. ISSN: 2615-9015. URL: http://sajs.ntt.edu.vn/index.php/jst/article/view/106 (cit. on p. 26).

[30] T. V. Hecke. “Power study of anova versus Kruskal-Wallis test”. In: *Journal of Statistics and Management Systems* 15 (2-3 May 2013), pp. 241–247. ISSN: 0972-0510. DOI: 10.1080/09720510.2012.10701623. URL: https://www.tandfonline.com/doi/abs/10.1080/09720510.2012.10701623 (cit. on p. 26).

[31] S. Holm. “A Simple Sequentially Rejective Multiple Test Procedure”. In: *Scandinavian Journal of Statistics* 6 (2 1979), pp. 65–70 (cit. on p. 26).

[32] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski, A. Wierzbicki. “Github projects. quality analysis of open-source software”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8851 (2014), pp. 80–94. ISSN: 16113349. DOI: 10.1007/978-3-319-13734-6_6. URL: https://www.researchgate.net/publication/278703145_GitHub_Projects_Quality_Analysis_of_Open-Source_Software (cit. on p. 18).

[33] H. Jebnoun, H. B. Braiek, M. M. Rahman, F. Khomh, P. Montreal. “The Scent of Deep Learning Code: An Empirical Study”. In: *Proceedings of the 17th International Conference on Mining Software Repositories* (2020), p. 11. DOI: 10.1145/3379597. URL: https://doi.org/10.1145/3379597.3387479 (cit. on p. 10).

[34] P. M. Julia Elliott. *2021 Kaggle Machine Learning & Data Science Survey*. 2021. URL: https://kaggle.com/competitions/kaggle-survey-2021 (cit. on pp. 1, 18, 25).

[35] L. P. Kaelbling, M. L. Littman, A. W. Moore. “Reinforcement Learning: A Survey”. In: *Journal of Artificial Intelligence Research* 4 (May 1996), pp. 237–285. ISSN: 1076-9757. DOI: 10.1613/JAIR.301. URL: https://www.jair.org/index.php/jair/article/view/10166 (cit. on p. 5).

[36] M. Källén, T. Wrigstad. “Jupyter Notebooks on GitHub: Characteristics and Code Clones”. In: *Art, Science, and Engineering of Programming* 5 (3 July 2020). DOI: 10.22152/programming-journal.org/2021/5/15. URL: http://arxiv.org/abs/2007.10146%20http://dx.doi.org/10.22152/programming-journal.org/2021/5/15 (cit. on p. 41).

[37] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian. “An in-depth study of the promises and perils of mining GitHub”. In: *Empirical Software Engineering* 21 (5 Oct. 2016), pp. 2035–2071. ISSN: 15737616. DOI: 10.1007/S10664-015-9393-5/TABLES/5. URL: https://link.springer.com/article/10.1007/s10664-015-9393-5 (cit. on p. 18).

[38] *Keras: Deep Learning for humans*. URL: https://keras.io/ (cit. on p. 6).

[39] *Künstliche Intelligenz - Adaption nach Branchen*. de. https://de.statista.com/statistik/daten/studie/1248554/umfrage/ki-adaption-nach-branchen-und-funktionen-weltweit/. Accessed: 2023-6-10 (cit. on p. 38).

[40] G. Lacerda, F. Petrillo, M. Pimenta, Y. G. Guéhéneuc. “Code smells and refactoring: A tertiary systematic review of challenges and observations”. In: *Journal of Systems and Software* 167 (Sept. 2020), p. 110610. ISSN: 0164-1212. DOI: 10.1016/J.JSS.2020.110610 (cit. on pp. 8, 10).

[41] V. Lenarduzzi, N. Saarimäki, D. Taibi. “Some SonarQube Issues have a Significant but Small Effect on Faults and Changes. A large-scale empirical study”. In: (2019) (cit. on p. 16).

[42] LinJimmy, RyaboyDmitriy. “Scaling big data mining infrastructure”. In: *ACM SIGKDD Explorations Newsletter* 14 (2 Apr. 2013), pp. 6–19. ISSN: 1931-0145. DOI: 10.1145/2481244.2481247. URL: https://dl.acm.org/doi/10.1145/2481244.2481247 (cit. on p. 5).

[43] Y. Y. Liu, J. W. Lin, C. C. Chen. “Cano-wgMLST_BacCompare: A bacterial genome analysis platform for epidemiological investigation and comparative genomic analysis”. In: *Frontiers in Microbiology* 10 (JULY 2019). ISSN: 1664302X. DOI: 10.3389/FMICB.2019.01687/FULL (cit. on p. 29).

[44] by Martin Fowler, K. Beck, J. Brant, W. Opdyke, don Roberts. *Refactoring: Improving the Design of Existing Code*. 2002 (cit. on pp. 8, 10).

[45] T. J. Mccabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2 (4 1976), pp. 308–320. ISSN: 00985589. DOI: 10.1109/TSE.1976.233837 (cit. on p. 9).

[46] P. E. McKight, J. Najab. “Kruskal-Wallis Test”. In: *The Corsini Encyclopedia of Psychology* (Jan. 2010), pp. 1–1. DOI: 10.1002/9780470479216.CORPSY0491. URL: https://onlinelibrary.wiley.com/doi/10.1002/9780470479216.corpsy0491 (cit. on p. 26).

[47] E. Meijer. “Behind every great deep learning framework is an even greater programming languages concept (keynote)”. In: (Oct. 2018), pp. 1–1. DOI: 10.1145/3236024.3280855. URL: https://dl.acm.org/doi/10.1145/3236024.3280855 (cit. on p. 3).

[48] P. Mooney. *2022 Kaggle Machine Learning & Data Science Survey*. 2022. URL: https://kaggle.com/competitions/kaggle-survey-2022 (cit. on pp. 1, 6, 18, 25).

[49] E. Murphy-Hill, C. Jaspan, C. Sadowski, D. Shepherd, M. Phillips, C. Winter, A. Knight, E. Smith, M. Jorde. "What Predicts Software Developers' Productivity?" In: *IEEE Transactions on Software Engineering* 47 (3 Mar. 2021), pp. 582–594. ISSN: 19393520. DOI: 10.1109/TSE.2019.2900308 (cit. on p. 18).

[50] B. van Oort. *Code Smells & Software Quality in Machine Learning Projects*. 2021. URL: https://repository.tudelft.nl/islandora/object/uuid%5C%3Ab20883f8-a921-487a-8a65-89374a1f3867 (cit. on p. 16).

[51] B. van Oort, L. Cruz, B. Loni, A. van Deursen. ""Project smells" - Experiences in Analysing the Software Quality of ML Projects with mllint". In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (May 2022), pp. 211–220. DOI: 10.1145/3510457.3513041. URL: https://dl.acm.org/doi/10.1145/3510457.3513041 (cit. on pp. 9, 11, 12, 16).

[52] B. V. Oort, L. Cruz, M. Aniche, A. V. Deursen. "The prevalence of code smells in machine learning projects". In: *Proceedings - 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI, WAIN 2021* (May 2021), pp. 35–42. DOI: 10.1109/WAIN52551.2021.00011 (cit. on pp. 9, 11, 35, 43).

[53] "Pan-Genome Analysis of Transcriptional Regulation in Six Salmonella enterica Serovar Typhimurium Strains Reveals Their Different Regulatory Structures". In: *mSystems* 7 (6 Dec. 2022). ISSN: 23795077. DOI: 10.1128/MSYSTEMS.00467-22/SUPPL_FILE/MSYSTEMS.00467-22-S0002.CSV. URL: %5Curl%7Bhttps://journals.asm.org/doi/10.1128/msystems.00467-22%7D (cit. on p. 29).

[54] T. van den Pol. *Sonarqube Rule Violations That Actually Lead To Bugs*. University of Utrecht, 2021 (cit. on p. 16).

[55] *PyTorch*. URL: https://pytorch.org/ (cit. on p. 7).

[56] pytorch. *GitHub - pytorch/pytorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration*. URL: https://github.com/pytorch/pytorch (cit. on p. 7).

[57] P. J. Rousseeuw, M. Hubert. "Robust statistics for outlier detection". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 1 (1 Jan. 2011), pp. 73–79. ISSN: 19424795. DOI: 10.1002/WIDM.2 (cit. on pp. 23, 24).

[58] N. Salkind. *Encyclopedia of Research Design*. SAGE Publications, Inc., 2010. DOI: 10.4135/9781412961288. URL: https://doi.org/10.4135/9781412961288 (cit. on p. 26).

[59] I. H. Sarker. "Machine Learning: Algorithms, Real-World Applications and Research Directions". In: *SN Computer Science* 2 (2021), p. 160. DOI: 10.1007/s42979-021-00592-x. URL: https://doi.org/10.1007/s42979-021-00592-x (cit. on p. 25).

[60] *scikit-learn: machine learning in Python — scikit-learn 1.2.2 documentation*. URL: https://scikit-learn.org/stable/ (cit. on p. 7).

[61] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. F. Crespo, D. Dennison. "Hidden technical debt in machine learning systems". In: *Advances in Neural Information Processing Systems* 2015-Janua (2015), pp. 2503–2511. ISSN: 10495258 (cit. on pp. 1, 5, 11).

[62] *Self-managed | SonarQube | Sonar*. URL: https://www.sonarsource.com/products/sonarqube/ (cit. on p. 16).

[63] A. Sharma, F. Thung, P. S. KOCHHAR, A. Sulistya, P. Singh, F. nung, P. S. Kochhar, D. Lo. “Cataloging GitHub repositories”. In: *EASE’17 Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering* (2017), pp. 15–16. DOI: 10.1145/3084226.3084287. URL: https://ink.library.smu.edu.sg/sis_research (cit. on pp. 11, 15, 24).

[64] A. J. Simmons, S. Barnett, J. Rivera-Villicana, A. Bajaj, R. Vasa. “A large-scale comparative analysis of Coding Standard conformance in Open-Source Data Science projects”. In: *International Symposium on Empirical Software Engineering and Measurement* (July 2020). DOI: 10.1145/3382494.3410680. URL: http://arxiv.org/abs/2007.08978%20http://dx.doi.org/10.1145/3382494.3410680 (cit. on pp. 1, 10).

[65] S. Skansi. *Introduction to deep learning: From Logical Calculus to Artificial Intelligence*. Vol. 114. 6. Springer International Publishing AG, 2018, p. 196. ISBN: 978-3-319-73003-5. DOI: 10.1007/978-3-319-73004-2 (cit. on p. 4).

[66] A. C. R. Tavares, N. A. Batista, M. M. Moro. “How COVID-19 Impacted Data Science: a Topic Retrieval and Analysis from GitHub Projects’ Descriptions”. In: *Anais do Simpósio Brasileiro de Banco de Dados (SBBD)* (Oct. 2021), pp. 325–330. ISSN: 2763-8979. DOI: 10.5753/SBBD.2021.17893. URL: https://sol.sbc.org.br/index.php/sbbd/article/view/17893 (cit. on pp. 11, 15, 18, 22, 24).

[67] *TensorFlow Core | Machine Learning for Beginners and Experts*. URL: https://www.tensorflow.org/overview (cit. on p. 5).

[68] *Transformers*. URL: https://huggingface.co/docs/transformers/index (cit. on p. 7).

[69] S. A. Vidal, C. Marcos, J. A. Díaz-Pace. “An approach to prioritize code smells for refactoring”. In: *Automated Software Engineering* 23 (3 Sept. 2016), pp. 501–532. ISSN: 15737535. DOI: 10.1007/S10515-014-0175-X/TABLES/11. URL: https://link.springer.com/article/10.1007/s10515-014-0175-x (cit. on p. 16).

[70] Z. Wang, K. Liu, J. Li, Y. Zhu, Y. Zhang. “Various Frameworks and Libraries of Machine Learning and Deep Learning: A Survey”. In: *Archives of Computational Methods in Engineering* (Feb. 2019), pp. 1–24. ISSN: 18861784. DOI: 10.1007/S11831-018-09312-W/FIGURES/19. URL: https://link.springer.com/article/10.1007/s11831-018-09312-w (cit. on pp. 6, 18).

[71] G. Wilson. “Software carpentry: Getting scientists to write better code by making them more productive”. In: *Computing in Science and Engineering* 8 (6 Nov. 2006), pp. 66–69. ISSN: 15219615. DOI: 10.1109/MCSE.2006.122 (cit. on p. 1).

[72] G. Wilson, J. Bryan, K. Cranston, J. Kitzes, L. Nederbragt, T. K. Teal. “Good enough practices in scientific computing”. In: *PLOS Computational Biology* 13 (6 June 2017), e1005510. ISSN: 1553-7358. DOI: 10.1371/JOURNAL.PCBI.1005510. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510 (cit. on p. 1).

[73] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. V. Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, A. M. Rush. “Transformers: State-of-the-Art Natural Language Processing”. In: (2020). URL: https://github.com/huggingface/ (cit. on p. 7).

[74] F. Zanartu, C. Treude, B. Cartaxo, ·. Hudson, S. Borges, P. Moura, M. Wagner, G. Pinto, H. S. Borges. "Automatically Categorising GitHub Repositories by Application Domain". In: (2022). DOI: 10.5281/zenodo.6423599. URL: https://doi.org/10.5281/zenodo.6423599 (cit. on pp. 15, 24, 25, 30, 38).

[75] N. Zazworka, M. A. Shaw, F. Shull, C. Seaman. "Investigating the impact of design debt on software quality". In: *Proceedings - International Conference on Software Engineering* (2011), pp. 17–23. ISSN: 02705257. DOI: 10.1145/1985362.1985366. URL: https://dl.acm.org/doi/10.1145/1985362.1985366 (cit. on p. 11).

[76] D. Zhang, N. Maslej, E. Brynjolfsson, J. Etchemendy, T. Lyons, J. Manyika, H. Ngo, J. C. Niebles, M. Sellitto, E. Sakhaee, Y. Shoham, J. Clark, R. Perrault. *The AI Index 2022 Annual Report*. AI Index Steering Committee, Stanford Institute for Human-Centered AI, 2022, p. 19. URL: https://aiindex.stanford.edu/wp-content/uploads/2022/03/2022-AI-Index-Report_Master.pdf (cit. on p. 25).

[77] H. Zhang, L. Cruz, A. V. Deursen. "Code Smells for Machine Learning Applications". In: *Proceedings - 1st International Conference on AI Engineering - Software Engineering for AI, CAIN 2022* (2022), pp. 217–228. DOI: 10.1145/3522664.3528620. URL: https://doi.org/10.1145/3522664.3528620 (cit. on p. 11).

All links were last followed on June 16, 2023.

A Appendix

A.1 Tukey HSD group comparisons - SonarQube Code Smells

Comparison	Statistic	p-value	Lower CI	Upper CI
(TensorFlow - PyTorch)	-1.586	0.118	-7.912	4.740
(TensorFlow - Keras)	-3.796	1.000	-28.075	20.482
(TensorFlow - Scikit-Learn)	-8.359	1.000	-16.187	-0.531
(TensorFlow - Transformers)	6.801	1.000	-10.142	23.745
(TensorFlow - TensorFlow / Keras)	-9.728	1.000	-20.275	0.820
(PyTorch - TensorFlow)	1.586	1.000	-4.740	7.912
(PyTorch - Keras)	-2.210	1.000	-26.009	21.588
(PyTorch - Scikit-Learn)	-6.773	1.000	-12.953	-0.592
(PyTorch - Transformers)	8.388	1.000	-7.860	24.636
(PyTorch - TensorFlow / Keras)	-8.142	1.000	-17.531	1.248
(Keras - TensorFlow)	3.796	1.000	-20.482	28.075
(Keras - PyTorch)	2.210	1.000	-21.588	26.009
(Keras - Scikit-Learn)	-4.563	1.000	-28.804	19.678
(Keras - Transformers)	10.598	1.000	-17.923	39.118
(Keras - TensorFlow / Keras)	-5.931	1.000	-31.182	19.319
(Scikit-Learn - TensorFlow)	8.359	1.000	0.531	16.187
(Scikit-Learn - PyTorch)	6.773	1.000	0.592	12.953
(Scikit-Learn - Keras)	4.563	1.000	-19.678	28.804
(Scikit-Learn - Transformers)	15.160	1.000	-1.729	32.050
(Scikit-learn - TensorFlow / Keras)	-1.369	1.000	-11.830	9.092
(Transformers - TensorFlow)	-6.801	0.027	-23.745	10.142
(Transformers - PyTorch)	-8.388	0.118	-24.636	7.860
(Transformers - Keras)	-10.598	1.000	-39.118	17.923
(Transformers - Scikit-Learn)	-15.160	1.000	-32.050	1.729
(Transformers - TensorFlow / Keras)	-16.529	1.000	-34.838	1.780
(TensorFlow / Keras - TensorFlow)	9.728	1.000	-0.820	20.275
(TensorFlow / Keras - PyTorch)	8.142	1.000	-1.248	17.531
(TensorFlow / Keras - Keras)	5.931	1.000	-19.319	31.182
(TensorFlow / Keras - Scikit-Learn)	1.369	1.000	-9.092	11.830
(TensorFlow / Keras - Transformers)	16.529	1.000	-1.780	34.838

Table A.1: Tukey's HSD Pairwise Group Comparisons (95.0% Confidence Interval) for SonarQube Code Smells per 1,000 LoC with applied Bonferroni-Holm correction

A.2 Tukey HSD group comparisons - Mllint Project Smells

Comparison	Statistic	p-value	Lower CI	Upper CI
(TensorFlow - PyTorch)	2.375	1.000	1.165	3.585
(TensorFlow - Keras)	-0.836	1.000	-5.481	3.808
(TensorFlow - Scikit-Learn)	-5.543	1.000	-7.040	-4.045
(TensorFlow - Transformers)	-2.424	1.000	-5.665	0.817
(TensorFlow - TensorFlow / Keras)	1.988	1.000	-0.029	4.006
(PyTorch - TensorFlow)	-2.375	1.000	-3.585	-1.165
(PyTorch - Keras)	-3.211	1.000	-7.763	1.341
(PyTorch - Scikit-Learn)	-7.917	1.000	-9.100	-6.735
(PyTorch - Transformers)	-4.798	1.000	-7.907	-1.690
(PyTorch - TensorFlow / Keras)	-0.386	1.000	-2.183	1.410
(Keras - TensorFlow)	0.836	1.000	-3.808	5.481
(Keras - PyTorch)	3.211	1.000	-1.341	7.763
(Keras - Scikit-Learn)	-4.706	1.000	-9.343	-0.069
(Keras - Transformers)	-1.587	1.000	-7.043	3.868
(Keras - TensorFlow / Keras)	2.825	1.000	-2.006	7.655
(Scikit-Learn - TensorFlow)	5.543	0.000	4.045	7.040
(Scikit-Learn - PyTorch)	7.917	0.000	6.735	9.100
(Scikit-Learn - Keras)	4.706	0.225	0.069	9.343
(Scikit-Learn - Transformers)	3.119	1.000	-0.112	6.350
(Scikit-learn - TensorFlow / Keras)	7.531	1.000	5.530	9.532
(Transformers - TensorFlow)	2.424	0.000	-0.817	5.665
(Transformers - PyTorch)	4.798	1.000	1.690	7.907
(Transformers - Keras)	1.587	1.000	-3.868	7.043
(Transformers - Scikit-Learn)	-3.119	1.000	-6.350	0.112
(Transformers - TensorFlow / Keras)	4.412	1.000	0.910	7.914
(TensorFlow / Keras - TensorFlow)	-1.988	1.000	-4.006	0.029
(TensorFlow / Keras - PyTorch)	0.386	1.000	-1.410	2.183
(TensorFlow / Keras - Keras)	-2.825	1.000	-7.655	2.006
(TensorFlow / Keras - Scikit-Learn)	-7.531	1.000	-9.532	-5.530
(TensorFlow / Keras - Transformers)	-4.412	1.000	-7.914	-0.910

Table A.2: Tukey's HSD Pairwise Group Comparisons (95.0% Confidence Interval) for average *mllint* project smells with applied Bonferroni-Holm correction

A.3 Code Smells not present in dataset

List of code smells not present in the dataset:

- "Tests should be skipped explicitly"
- "Bare “raise” statements should not be used in “finally” blocks"
- “Statements should be on separate lines”
- "Module names should comply with a naming convention"
- "Track lack of copyright and license headers"
- "Files should not be too complex"
- "Files should contain an empty newline at the end"
- "Cyclomatic Complexity of classes should not be too high"
- "The most specific “unittest” assertion should be used"
- "Files should not have too many lines of code"
- "Lines should not be too long"
- "Python parser failure"
- "Comments should not be located at the end of lines of code"
- "Functions should not have too many lines of code"
- "Parentheses should not be used after certain keywords"
- "New-style classes should be used"
- "Docstrings should be defined"
- "Track “TODO” and “FIXME” comments that do not contain a reference to a person"
- "Long suffix “L should be upper case”"
- "Track comments matching a regular expression"
- "Cyclomatic Complexity of functions should not be too high"
- "Control flow statements “if”, “for”, “while”, “try” and “with” should not be nested too deeply"
- "Functions should use “return” consistently"
- "Test methods should be discoverable"
- "Track uses of “NOSONAR” comments"
- "Methods and properties that don’t access instance data should be static"
- "The “exec” statement should not be used"
- "Functions should not contain too many return statements"

- "Backticks should not be used"
- "Lines should not end with trailing whitespaces"

A.4 Wordclouds

A.4.1 Keras

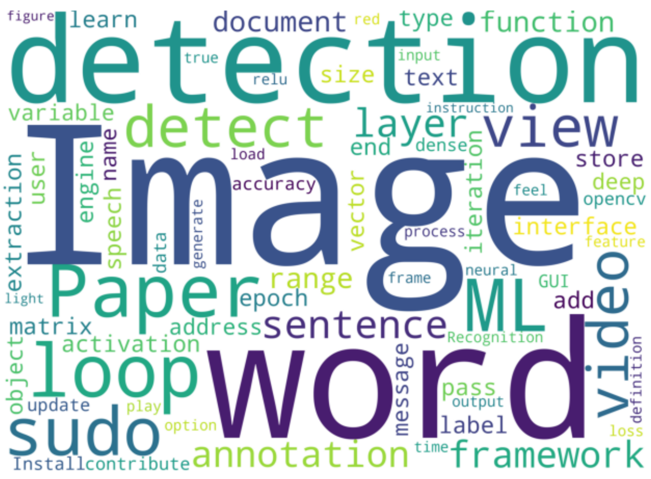

Figure A.1: Keras wordcloud for Topic “Computer Vision”

A.4.2 TensorFlow & Keras

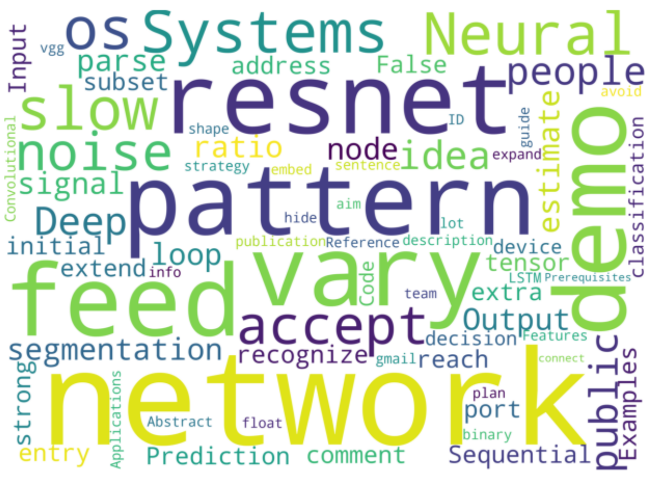

Figure A.2: Transformers wordcloud for Topic “Region Annotation”

A.4.3 Scikit-learn

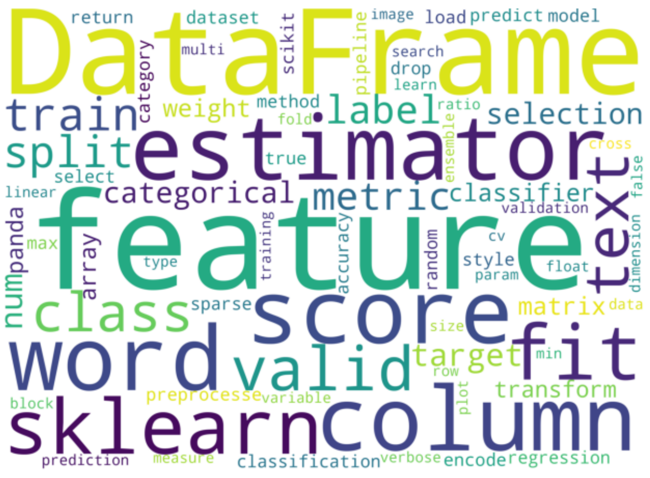

Figure A.3: TensorFlow & Keras wordcloud for Topic “NLP”

A.4.4 Transformers

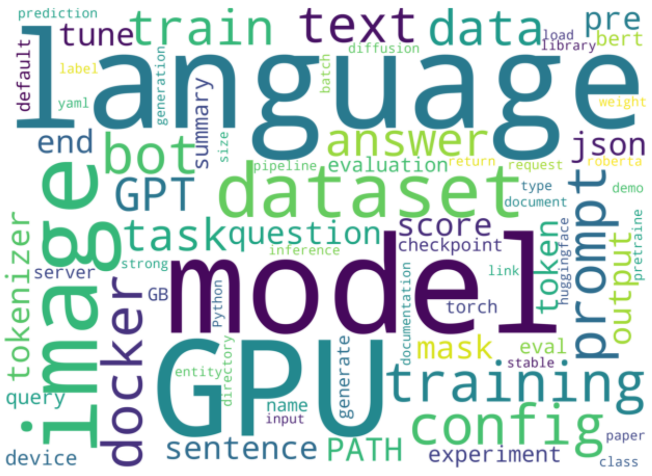

Figure A.4: Transformers wordcloud for Topic “Speech Translation (NLP)”

A.4.5 PyTorch

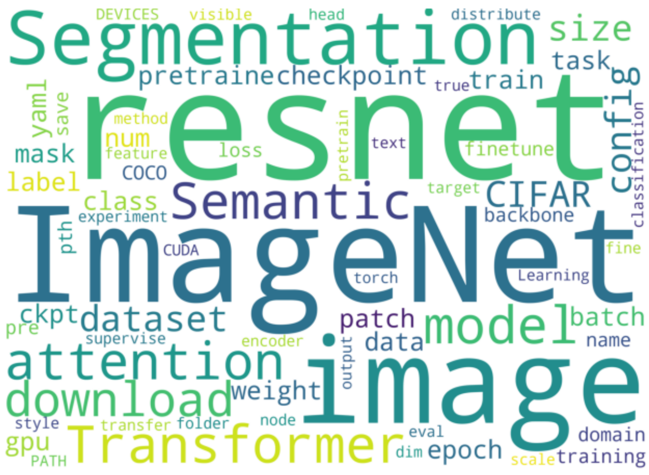

Figure A.5: PyTorch wordcloud for Topic “Image Processing”

A.5 Code Smell Analysis

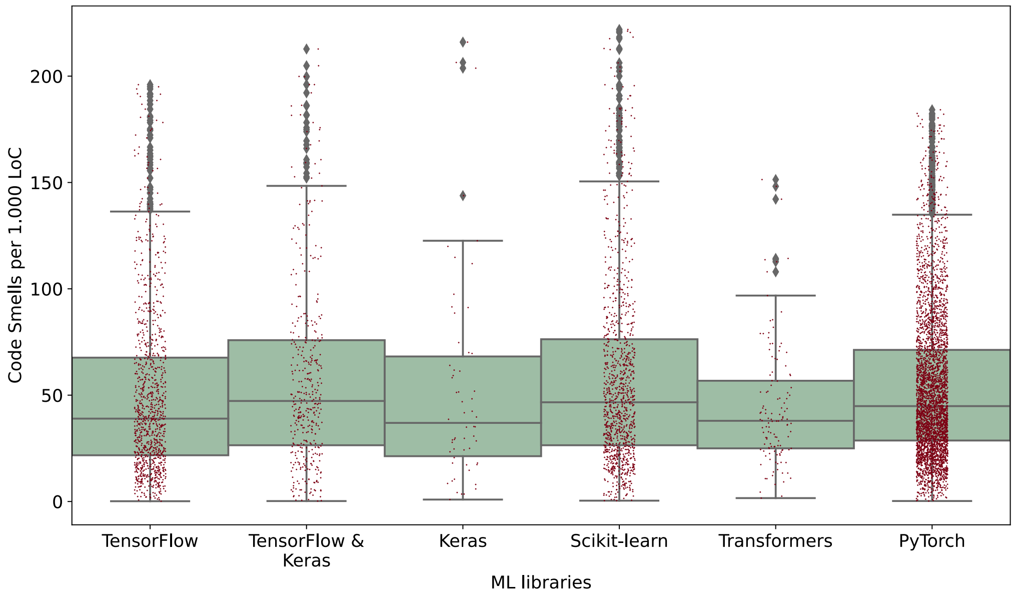

Figure A.6: Boxplot of Code Smells per 1,000 LoC for every ML library