Studying the Machine Learning Lifecycle and Improving Code Quality of Machine Learning Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Mark Haakman

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.tudelft.nl/eemcs/

AI for Fintech Research
ING Bank N.V.
Frankemaheerd 1
Amsterdam, the Netherlands
www.ing.nl

Studying the Machine Learning Lifecycle and Improving Code Quality of Machine Learning Applications

Author: Mark Haakman
Student id: 4467795
Email: m.p.a.haakman@student.tudelft.nl

Abstract

As organizations start to adopt machine learning in critical business scenarios, the development processes change and the reliability of the applications becomes more important. To investigate these changes and improve the reliability of those applications, we conducted two studies in this thesis. The first study aims to understand the evolution of the processes by which machine learning applications are developed and how state-of-the-art lifecycle models fit the current needs of the fintech industry. Therefore, we conducted a case study with seventeen machine learning practitioners at the fintech company ING. The results indicate that the existing lifecycle models CRISP-DM and TDSP largely reflect the current development processes of machine learning applications, but there are crucial steps missing, including a feasibility study, documentation, model evaluation, and model monitoring. Our second study aims to reduce bugs and improve the code quality of machine learning applications. We developed a static code analysis tool consisting of six checkers to find probable bugs and enforcing best practices, specifically in Python code used for processing large amounts of data and modeling in the machine learning lifecycle. The evaluation of the tool using 1000 collected notebooks from Kaggle shows that static code analysis can detect and thus help prevent probable bugs in data science code. Our work shows that the real challenges of applying machine learning go much beyond sophisticated learning algorithms – more focus is needed on the entire lifecycle.

Thesis Committee:

Chair and University Supervisor:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Dr. M. Aniche, Faculty EEMCS, TU Delft
Committee Member:	Dr. Cynthia Liem MMus, Faculty EEMCS, TU Delft
Committee Member and Supervisor:	Dr. L. Cruz, Faculty EEMCS, TU Delft

Contents

Chapter 1

Introduction

Machine learning has generated huge societal impact as it has great practical value in a variety of application domains, such as bioinformatics, computer vision, natural language processing, and robotics. The access to increasing computing systems' analytical power and to massive amounts of data has also resulted in more organizations using machine learning for supporting customer value creation, productivity improvement, and insight discovery [7, 53]. Easy access to machine learning techniques with libraries such as `PyTorch`[1] and `scikit-learn`[2] ease the adoption of machine learning even more. Cloud platforms, such as Google Cloud[3] and Amazon Web Services[4], even facilitate running machine learning techniques on a large scale without the necessity for the sophisticated hardware.

The machine learning lifecycle is the process of developing, training, and serving machine learning applications. As organizations start to adopt machine learning in critical business scenarios, the development processes change and the reliability of the applications becomes more important. To investigate these changes and improve the reliability of those applications, we conducted two studies in this thesis.

1.1 Objectives and Research Questions

This section briefly states the objectives and research questions of both studies performed in this thesis project. The motivation for them will be elaborated in more detail in the introductions of the corresponding chapters.

Studying the Machine Learning Lifecycle The objective of the first study (Chapter 2) is to understand the evolution of machine learning development and how state-of-the-art lifecycle models fit the current needs of the (fintech) industry. We defined the following research questions for our study:

1. *How do existing machine learning lifecycle models fit the fintech domain?*

[1] `https://pytorch.org`
[2] `https://scikit-learn.org`
[3] `https://cloud.google.com`
[4] `https://aws.amazon.com`

2. *What are the specific challenges of developing machine learning applications in fintech organizations?*

Improving Code Quality of Machine Learning Applications The objective of the second study (Chapter 3) is to reduce bugs and improve the code quality of machine learning applications. To work towards this objective, we defined the following research questions:

1. *What bugs and best practices typical to machine learning can be detected using static code analysis?*

2. *To what extent are these bugs present and best practices not followed in existing open source scripts and programs?*

1.2 Approach

This section briefly explains the industry partner and approaches of both studies.

Industry Partner The research in this thesis is conducted at ING, a global bank with a strong European base. ING offers retail and wholesale banking services to 38 million customers in over 40 countries, with over 53,000 employees [18]. ING is an interesting organization for this thesis, as it is currently leveraging a major shift in the organization to adopt AI to improve its services and increase business value.

Studying the Machine Learning Lifecycle We conducted an exploratory case study at ING aimed at understanding how the fintech industry is currently dealing with the challenges of developing machine learning applications at scale. We interviewed 17 people with different roles and from different departments. We report our findings organized among eight core machine learning lifecycle stages and three overarching categories. We thereafter refine the existing lifecycle models based on our observations at ING.

Improving Code Quality of Machine Learning Applications With the goal of reducing bugs and improving the code quality of machine learning applications, we developed a static code analysis tool consisting of six checkers to find probable bugs and enforcing best practices, specifically in Python code used for processing large amounts of data and modeling in the machine learning lifecycle. The tool is thereafter evaluated by running it on 1000 collected scripts from Kaggle, an online platform for data scientists and machine learning practitioners.

1.3 Contributions

The main contributions of this thesis are as follows:

1. A (replicable) case study is designed and performed which gives insight into how the industry is adopting machine learning.

2. A research paper is submitted to the IEEE International Conference on Software Maintenance and Evolution (ICSME) 2020, to share the case study with the research community.
3. Six probable bugs and best practices in data science code are identified, of which practitioners should be aware.
4. The six probable bugs and best practices are integrated as checkers into a static code analysis tool, `dslinter`, which is available on GitHub[5] so that practitioners can run the checkers on their code.

1.4 Report Organization

The remaining of this thesis report consists of three chapters: one for each study and a final conclusions chapter. Chapter 2 contains the case study on the lifecycle of machine learning applications, Chapter 3 contains the study on using static code analysis for data science. Both chapters stand on their own, as both contain sections for the introduction, related work, core sections, discussion, and conclusions specific for the study in that chapter. Chapter 4 concludes the entire thesis. In Appendix A a glossary can be found with frequently used terms.

[5] `https://github.com/MarkHaakman/dslinter`

Chapter 2

Exploratory Case Study on the Lifecycle of Machine Learning Applications

The case study described in this chapter resulted in a paper submission to the research track of the IEEE International Conference on Software Maintenance and Evolution (ICSME) 2020. The preprint can be found online[1].

2.1 Introduction

Artificial Intelligence (AI) has become increasingly important for organizations to support customer value creation, productivity improvement, and insight discovery. Pioneers in the AI industry are asking how to better develop and maintain AI software [31]. In this study we focus on machine learning, the branch of AI that deals with the automatic generation of knowledge models based on sample data.

Although most of the AI techniques are not so recent (e.g., neural networks were already being applied in the 1980s [30]), the recent access to large amounts of data and more computing power has exploded the number of scenarios where AI can be applied [53, 7]. In fact, AI is now being used to add value in critical business scenarios. Consequently, a number of new challenges are emerging in the lifecycle of AI systems, comprising all the stages from their conception to their retirement and disposal. Like normal software applications, these projects need to be planned, tested, debugged, deployed, maintained, and integrated into complex systems.

Companies leading the advent of AI are reinventing their development processes and coming up with new solutions. Thus, there are many lessons to be learned to help other organizations and guide research in a direction that is meaningful to the industry. This is particularly relevant for highly-regulated industries such as fintech, as new processes need to be designed to make sure AI systems meet all required standards.

Recent research has addressed how developing AI systems is different from developing regular Software Engineering systems. A case study at Microsoft identified the following differences [3]: 1) data discovery, management, and versioning are more complex; 2) practitioners ought to have a broader set of skills; and 3) modular design

[1] https://doi.org/10.5281/zenodo.3941475

is not trivial since AI components can be entangled in complex ways. Unfortunately, existing research offers little insight into the challenges of transforming an existing IT organization into an AI-intensive one.

Examples of existing models describing the machine learning lifecycle are the Cross-Industry Standard Process for Data Mining (CRISP-DM) [44] and the Team Data Science Process (TDSP) [32]. However, machine learning is being used for different problems across many different domains. Given the fast pace of change in AI and recent advancements in Software Engineering, we suspect that there are deficiencies in these lifecycle models when applied to a fintech context.

To remedy this, we set out this exploratory case study aimed at understanding and improving how the fintech industry is currently dealing with the challenges of developing machine learning applications at scale. ING is a relevant case to study, since it has a strong focus on financial technology and Software Engineering and it is undergoing a bold digital transformation to embrace AI as an important competitive factor. By studying ING, we provide a snapshot of the rapid evolution of the approach to machine learning development.

We define the following research questions for our study:

RQ1: *How do existing machine learning lifecycle models fit the fintech domain?*

RQ2: *What are the specific challenges of developing machine learning applications in fintech organizations?*

We interviewed 17 people at ING with different roles and from different departments. Thereafter, we triangulated the resulting data with other resources available inside the organization. Furthermore, we refine the existing lifecycle models CRISP-DM and TDSP based on our observations at ING.

Our results unveil important challenges that ought to be addressed when implementing machine learning at scale. Feasibility assessments, documentation, model risk assessment, and model monitoring are stages that have been overlooked by existing lifecycle models. There is a lack of standards and there is a need for automation in the documentation and governance of machine learning models. Finally, we pave the way for shaping the education of AI to address the current needs of the industry.

The remainder of this chapter is structured as follows. In Section 2.2 we introduce existing lifecycle models and describe related work. In Section 2.3, we outline the study design. We report the data collected in Section 2.4 and we answer the research questions in Section 2.5. We discuss our findings and threats to validity in Section 2.6. Finally, in section 2.7, we pinpoint conclusions and outline future work.

2.2 Background and Related Work

In this section, we present the lifecycle models considered in this study and examine existing literature outlining the differences with our study.

2.2.1 Existing Lifecycle Models

In this study, we consider two reference models for the lifecycle of machine learning applications: Cross-Industry Standard Process for Data Mining (CRISP-DM) [44] and Team Data Science Process (TDSP) [32]. We chose CRISP-DM, as although it is

twenty years old, it is still the *de facto* standard for developing data mining and knowledge discovery projects [28]. We selected TDSP as a modern industry methodology, which has at a high level much in common with CRISP-DM. There are other methodologies, but most are similar to CRISP-DM and TDSP. Findings in our study can be extrapolated to those other methodologies.

CRISP-DM aims to provide anyone with "a complete blueprint for conducting a data mining project" [44]. Although data mining is not the common term used nowadays, it is valid for any project applying scientific methods to extracting value from data, including machine learning [28]. CRISP-DM breaks down a project into six phases, as presented in Figure 2.1. It typically starts with **Business Understanding** to determine business objectives, going back and forward with **Data Understanding**. It is followed by **Data Preparation** to make data ready for **Modeling**. The produced model goes through an **Evaluation** in which it is decided whether the model can go for **Deployment** or it needs another round of improvement. The arrows between stages indicate the most relevant and recurrent dependencies, while the arrows in the outer circle indicate the evolution of machine learning systems after being deployed and their iterative nature.

Based on CRISP-DM, a number of lifecycle models have been proposed [28, 26] to address varying objectives. Derived models extend CRISP-DM for projects with geographically dispersed teams [34], with large amounts of data and more focus on automation [54, 40], or targeting the model reuse across different contexts [27].

TDSP is "an agile, iterative data science methodology" by Microsoft, to deliver machine learning solutions efficiently [32]. The original methodology includes four major stages, as can be seen in Figure 2.2: **Business Understanding**, **Data Acquisition**, **Modeling** and **Deployment**. As depicted by the arrows in the figure, TDSP proposes stronger dependencies but does not enforce a particular order between stages,

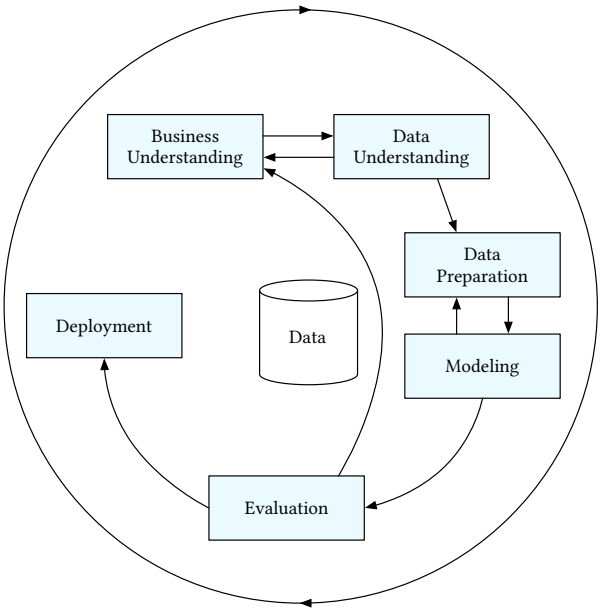

Figure 2.1: Cross-Industry Standard Process for Data Mining (CRISP-DM).

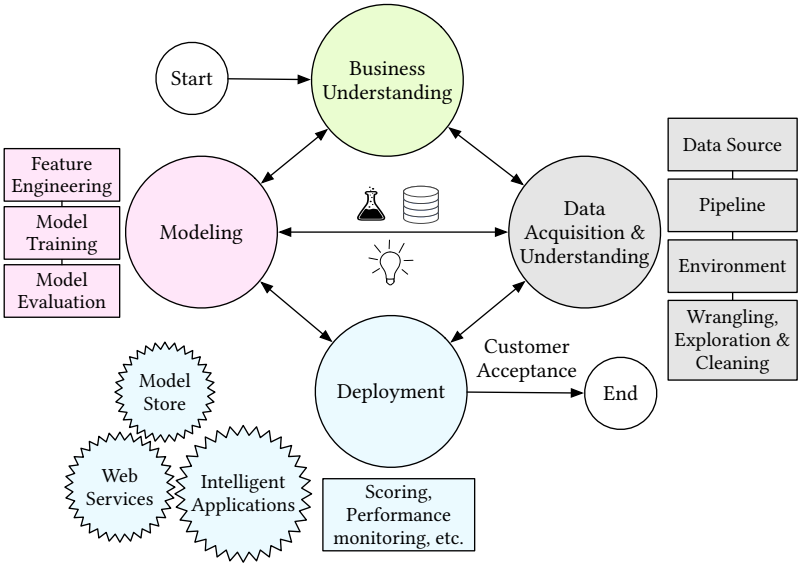

Figure 2.2: Team Data Science Process (TDSP).

emphasizing that different stages can be iteratively repeated at almost any time in the project.

Despite the number of advancements proposed in previous work, they do not tackle AI systems that target challenges faced by the fintech industry. Our work pinpoints the changes that needed to be accommodated for AI systems operating under heavy-regulated scenarios and bringing value over pre-existing non-data-driven approaches.

2.2.2 Related Work

The machine learning development lifecycle has been studied in practice in previous research. Amershi et al. [3] have conducted a case study at Microsoft to study the differences between Software Engineering and machine learning. The most important challenges found are model scaling, evolution, evaluation, deployment, and data management. We complement this study by comparing our observations with existing machine learning lifecycle models.

Another case study from industry has been performed at Booking.com by Bernardi et al. [7]. In contrast with academic research in which machine learning models are validated by means of an error measurement, models at Booking.com are validated through business metrics such as conversion or cancellations. The paper describes process stages such as model designing, deployment, monitoring, and evaluation, but no formal lifecycle model is defined.

Hill et al. [16] studied how people develop intelligent systems in practice. The study leverages a high-level model of the process and identifies the main challenges. Results show that developers struggle with establishing repeatable processes and that there is a basic mismatch between the tools available versus the practical needs. In this study, we extend the work by Hill et al. by looking more closely at what happens after the machine learning model has been evaluated, for example regarding its deployment and monitoring.

The paper by Lin and Ryaboy [24] describes the *big data mining cycle* at Twitter, based on the experience of the two authors. The main points made are that for data-driven projects, most time goes to preparatory work before, and engineering work after the actual model training and that a significant amount of tooling and infrastructure is required. In our study, we validate the recommendations of these two experts with a case study with seventeen participants.

Concrete challenges data scientists face are elaborated upon in the study by Kim et al. [22]. They have surveyed 793 professional data scientists at Microsoft. An example of a challenge found is that the proliferation of data science tools makes it harder to reuse work across teams. This challenge is also reinforced in the study by Ahmed et al. [2]. As models are mostly implemented without standard API, input format, or hyperparameter notation, data scientists spend considerable effort on implementing glue code and wrappers around different algorithms and data formats to employ them in their pipelines. Ahmed et al. [2] show evidence that most models need to be rewritten by a different engineering team for deployment. The root of this challenge lies on run-time constraints, such as a different hardware or software platform, and constraints on the pipeline size or prediction latency.

More studies looked at machine learning from a Software Engineering viewpoint. Sculley et al. [43] identified a number of machine learning-specific factors that increase technical debt, such as boundary erosion and hidden feedback loops. Breck. et al [8] have proposed 28 specific tests for assessing production readiness for machine learning applications. These tests include tests for features and data, model development, infrastructure, and monitoring. Arpteg et al. [4] have identified Software Engineering challenges of building intelligent systems with deep learning components based on seven projects from companies of different types and sizes. These challenges include development, production, and organizational challenges, such as experiment management, dependency management, and effort estimation. In this current study, we will extend this line of research and identify where Software Engineering can help mitigate inefficiencies in the development and evolution of machine learning systems.

2.3 Research Design

To identify the gaps in the existing machine learning lifecycle models and explore key challenges in the field, we perform a single-case exploratory case study. This is a recurrent methodology to define new research by looking at concrete situations and to shed empirical light on existing concepts and principles [55]. We follow the guidelines proposed by Brereton et al. [9] and Yin's [55] case study methodology.

It is not our objective to build an entirely new theory from the ground up. For that reason, we do not adopt a Grounded Theory (GT) approach, although we do use a number of techniques based on GT [45]: e.g., theoretical sampling, memoing, memo sorting, and saturation.

The design of the study is further described in this section.

2.3.1 The Case

The case under study is ING, a global bank with a strong European base. ING offers retail and wholesale banking services to 38 million customers in over 40 countries,

with over 53,000 employees [18]. ING has a strong focus on fintech, the digital transformation of the financial sector, and the professionalization of AI development.

A bank of this size has many use cases where machine learning can help. Examples include traditional banking activities such as assessing credit risk, the execution of customer due diligence and transaction monitoring requirements related to fighting financial economic crime. Other examples of use cases are improving customer service and IT infrastructure monitoring.

Development teams at ING follow an agile way of working and the organization is structured similarly to the Spotify organization model [23] with tribes, squads, and chapters. The basic unit is a *squad*, a self-organizing team similar to a Scrum team. A collection of squads working in related areas form a *tribe*. A *chapter* brings people with the same expertise together across squads and tribes.

ING is currently leveraging a major shift in the organization to adopt AI to improve its services and increase business value. The challenges that ING is facing at the moment make it an interesting case for our study and allow us to identify gaps between current challenges by the industry and academia.

2.3.2 Research Methodology

Semi-structured interviews are the main source of data in this case study. The data is later triangulated with other resources available inside the organization. Documentation in the intranet of ING is used to gain a deeper understanding of the platforms and processes mentioned in the interviews.

The approach used to collect information from interviews and report data is based on the guidelines proposed by Halcomb et al. [15]. It is a reflexive, iterative process:

1. Audio taping of the interview and concurrent note-taking.
2. Reflective journaling immediately post-interview.
3. Listening to the audiotape and revising memos.
4. Data analysis.

Participants

We selected interviewees based on their role and their involvement in the process of developing machine learning applications. We strove to include people of many different roles and from many different departments. The starting position for finding interviewees was the lead of a Software Analytics research team within ING. More interviewees were found by the recommendations of other interviewees. The interviewees were also able to suggest other sources of evidence that might be relevant. We increased the number of participants until we reached a level of saturation in the remarks mentioned by interviewees for each stage of the lifecycle.

An overview of the selected participants, with their role and department, can be seen in Table 2.1. In total, we interviewed seventeen participants. The sixth interview involved two participants. Therefore, they are labeled as P06a and P06b.

Table 2.1: Overview of Interviewees

ID	Role	Department
P01	IT Engineer	Application Platforms
P02	IT Engineer	IT Infrastructure Monitoring
P03	Productmanager	Financial Crime
P04	IT Architect	Enterprise Architects
P05	IT Engineer	IT4IT
P06a*	Advice Professional	Model Risk Management
P06b*	Advice Professional	Model Risk Management
P07	Manager IT	Global Engineering Platform
P08	Feature Engineer	Data & Analytics
P09	Data Scientist	Wholesale Banking Analytics
P10	Data Scientist	Chapter Data Scientists
P11	IT Engineer	Application Platform
P12	Data Scientist	AIOps
P13	Data Scientist	Wholesale Banking Analytics
P14	Data Scientist	Financial Crime
P15	Data Scientist	Analytics
P16	Data Scientist	Chapter Data Scientists

**The sixth interview involved two participants, labeled P06a and P06b.*

Interview Design

The interviews are conducted together with Luís Cruz and took approximately one hour. We took notes during the interviews and we recorded the interviews with the permission of the participants. This section outlines the main steps of our interview design. The full details can be found in our corresponding case study protocol [14].

As interviewers, we started by introducing ourselves and provided a brief description of the purpose of the interview and how it relates to the research being undertaken. We asked the interviewees to introduce themselves and describe their main role within the organization. After the introductions, we asked the interviewee to think about a specific machine learning project he or she was working on recently. Based on that project, we asked the interviewee to describe all the different stages of the project. In particular, we asked questions to understand the main challenges they faced and the solutions they had to design.

Post-interview Strategy

Right after each interview, the two interviewers got together for a collaborative *memoing* process (also called *reflective journaling* [15]). Memoing is the review and formalization of field-notes and the expansion of initial impressions of the interaction with more considered comments and perceptions. Memoing is chosen over creating verbatim transcriptions, because the costs associated with interview transcription, in terms of time, physical, and human resources, are significant. Also, the process of memoing assisted the researchers to capture their thoughts and interpretations of the interview

data [52]. The audio recordings could still be used to facilitate a review of the interviewers' performance, and assist interviewers to fill in blank spaces in their field notes and check the relationship between the notes and the actual responses [11].

The interviewers took between 30–45 minutes to refine their notes. In this process, the notes are assigned under different lifecycle stages. We used the nomenclature from existing frameworks (e.g. CRISP-DM and TDSP) as a rule of thumb, or we defined new stages in case it helps understand a particular part of the process.

After some time, the interviewers amended the memos by reviewing the audiotapes. The purpose of this stage was to ensure that the memos provided an accurate reflection of the interviews [15]. Once the researchers were confident that their memos accurately represented each interview, the process of content analysis is used to elicit common themes [15].

Each interview resulted in three artifacts: the recording of the interview, the field notes taken during the interview, and the memos as a result of the above mentioned memoing.

2.4 Data Analysis

The input of the interviewees does not answer the research questions directly. Therefore, we report the resulting data of the interviews in this section and we use this data to answer the research questions later in Section 2.5.

We organize the data among eight core machine learning lifecycle stages: problem design, requirements, data engineering, modeling, documentation, model evaluation, model deployment, and model monitoring. Overarching data that does not fit these stages is categorized under testing, iterative development, and education. These stages and categories are based on stages defined by CRISP-DM and TDSP (cf. Section 2.2.2) or mentioned by practitioners themselves.

For all the remarks, we identify the practitioner who mentioned them by referencing the corresponding ID from Table 2.1. Given that this is a qualitative analysis, the number of individuals supporting a particular result has no quantitative meaning on its relevance.

2.4.1 Problem Design

Machine learning projects at ING start with the definition of the problem that needs to be solved. Two main approaches are observed in this study:

1. Innovation push: a stakeholder comes up with a question or problem that needs to be solved. A team is set up to design a solution using a suitable machine learning technique.

2. Technology push: a team identifies new data or a set of machine learning techniques that may add business value and are potentially useful or solving problems within the organization. This approach aims to optimize processes, reduce manual work, increase model performance, and create new business opportunities.

The problem is defined together with stakeholders and it is assessed whether using machine learning is appropriate to solve the problem (P01, P14, P15). In the teams of P15 and P14, this is done by collaboratively filling in a project document with the stakeholders which contains information like the problem statement, goals, and the corresponding business case. Also, domain experts outside the teams are part of this.

2.4.2 Requirements

Besides project-specific requirements, many of the requirements come from the organization and are applicable to every machine learning application (P15). These requirements include traceability, interpretability, and explainability (P01, P04, P07, P15). Together with all other regulatory requirements, they pose a big challenge while developing machine learning applications (P04). A natural consequence of regulatory requirements is that black-box AI models cannot be used in most situations (P01, P04, P14). For risk management safeness, only interpretable/explainable AI models are accepted.

Project-specific requirements are often defined by the product owner together with the stakeholders (P10). Data requirements are said to become more clear while working with the model (P04). As the users of the system are often no machine learning experts, defining the model performance requirements is sometimes a challenge (P09, P13).

2.4.3 Data Engineering

Interviewees describe that data engineering requires the major part of the lifetime of a machine learning project (P03, P10, P15) and is also the most important for the success of the project (P10).

Data Collection

Data collection is considered a very challenging and time-consuming task (P03, P04, P12, P14). Typical use cases require access to sensitive data, which needs to be formally requested. ING has an extensive data governance framework that, among others, assigns data management roles (e.g. data owner) and rules for obtaining, sharing, and using data. Each dataset is assigned a criticality rating, to define the degree of data governance and control required.

There might be people with different access privileges to data in the same project. This means that, in the exploratory stages of some projects using critical data, only a restricted number of team members (e.g., data scientists) are able to perform an exploratory analysis of data. The remaining practitioners will only have access to the model specification (P04).

A challenge of data collection is making sure that the (training and test) data collected is representative of the problem (P13). As an example, if a machine learning model is trained on systems logs, it should be made sure that logs of all systems are available. Another challenge is merging data from multiple sources (P10, P12). Going back to the logging example, different systems may have different logging formats, but the configurations of these formats can not be altered by the developers creating the model.

Data Understanding

In the data understanding stage, an assessment is done on the quality of available data and how much processing will be required to use that data. It comprises exploratory data analysis, often including graphical visualizations and summarization of data. According to P09, the temptation of applying groundbreaking machine learning techniques tends to overlook the importance of understanding the data.

Data understanding is also an important step to assess the feasibility of the project. Thus, it entails not only performing an exploratory analysis, but also a considerable effort in communicating the main findings to all the different stakeholders.

Data Preparation

After the data is collected and it is assessed that the data is representative of the problem being solved, the data is prepared to be used for modeling.

A challenge regarding data preparation is that the same pre-processing has to be ensured in the development environment and in the production environment (P08, P09). Data streams in production are different than in the development environment and it is easier to clean training and testing data than production data (P09).

2.4.4 Modeling

Model training is mostly done in on-premises environments such as Hadoop[2] and Spark[3] clusters (P09) or in generic systems using, for example, the scikit-learn[4] library (P01). These private platforms are connected with the data lakes where data is stored, so training can be done on (a copy of) real production data (P01, P03). The on-premises environment has no outgoing connection to the internet, so a connection to other cloud services such as Microsoft Azure[5] or Google Cloud[6] is not possible (P08). This means that data scientists are limited to the tools and platforms available within the organization when dealing with sensitive data. Also, all project dependencies need to be previously approved, after which they are made available in a private package repository (P12), which contains whitelisted packages that have been internally audited. Fewer restrictions are in place if machine learning is applied to public data, for example on stock prices. In that case, external cloud services and packages may be used (P09).

Model training is an iterative process. Usually, multiple models are created for the same problem. First, a simple model is created (e.g., a linear regression model) to set as a baseline (P09). In the following iterations, more advanced models are compared to this baseline model. If an approach other than machine learning already exists (e.g., rule-based software), the models are also compared with this.

To keep track of different versions of models, different teams use different strategies. For example, the team of P08 keeps track of an experiment log using a spread-

[2]Hadoop enables distributed processing of large data sets across clusters of computers. https://hadoop.apache.org

[3]Spark is a unified analytics engine for large-scale data processing. https://spark.apache.org

[4]Scikit-learn is a machine learning library for Python. https://scikit-learn.org

[5]Microsoft Azure is a cloud computing service. https://azure.microsoft.com/en-us

[6]Google Cloud is a cloud computing service. https://cloud.google.com

sheet, in which the training set, validation set, model, and pre-processing steps are specified for each version. This approach for versioning is preferred over solutions like MLFlow[7] for the sake of simplicity (P08, P15).

Model Scoring

An implicit sub stage of modeling is assessing model performance to measure how well the predictions of the model represent ground truth data.

We define *Model Scoring* as assessing the performance of the model based on scoring metrics (e.g., f1-score for supervised learning). It is also known as *Validation* by the machine learning community, which should not be confused with the definition by the Software Engineering community[8] [41, 1].

The main remarks for this stage are related to defining the right set of metrics (P03, P06, P12, P14, P15, P16). The problem is two-fold: 1) identify the right metrics and 2) communicate why the selected metrics are right. Practitioners report that this is very problem-specific. Thus, it requires a good understanding of the business, data, and learning algorithms being used. From an organization's point of view, these different perspectives are a big barrier to defining validation standards.

2.4.5 Documentation

Each model has to be documented (P02). This serves multiple goals. It makes assessing the model from a regulatory perspective possible (P09, P13), it enables reproducibility, and also can make the model better because it is looked at from a broad perspective – i.e., a "helicopter view" (P09). It also provides an audit trail of actions, decisions, versions, etc. that supports evidencing. Documentation also supports the transfer of knowledge, for example, new team members or the end-users which are mostly not machine learning experts (P12). Just like code, documentation is also peer-reviewed (P13).

The content of the documentation differs slightly per department, but all documentation should at least follow the minimum standards defined by the model risk management framework (P06). Some teams extend on this by creating templates for documentation themselves (P13). In general, the following is documented when developing a machine learning application: the purpose, methodology, assumptions, limitations, and the use of the model. More concretely, a *Technical Model Document* is created which includes the model methodology, input, output, performance metrics and measurements, and testing strategy (P14). It furthermore states all faced difficulties and their solutions, plus the main (technical) decisions (P09). It has to explain why a certain model is chosen and what its inner workings are, to be able to demonstrate the application does what the creators claim it is doing.

2.4.6 Model Evaluation

An essential step in the evaluation of the model is communicating how well the model performs according to the defined metrics. It is about demonstrating that the model

[7] MLFlow is a platform to manage the machine learning lifecycle. `https://mlflow.org`

[8] *Validation* in Software Engineering "is the set of activities ensuring and gaining confidence that a system is able to accomplish its intended use, goals and objectives" [1].

meets business and regulatory needs and assessing the design of the model. One key difference between the metrics used in this step and the metrics used for *Model Scoring* is that these metrics are communicated to different stakeholders that do not necessarily have a machine learning or data science background. Thus, the set of metrics needs to be extended to a general audience. One complementary strategy used by practitioners is having live demos of the model with business stakeholders (P03, P15, P16). These demos allow stakeholders to try out different inputs and try corner cases.

Model Risk Assessment

An important aspect of evaluating a model at ING is making sure it complies with regulations, ethics, and organizational values (P15, P06). This is a common task for any type of model built within the organization – i.e., not only machine learning models but also economic models, statistical forecasting models, and so on. In the interviews, *Model Risk Assessment* was mentioned as mandatory within the model governance strategy, undertaken in collaboration with an independent specialized team (P06, P14). Depending on the criticality level of the model, the intensity of the review may vary. Each model owner is responsible for the risk management of their model, but colleagues from the risk department help and challenge the model owner in this process.

During the periodic risk assessment process, assessors inspect the documentation provided by the machine learning team to assess whether all regulations and minimum standards are followed. Although the process is still under development within ING, the following key points are being covered: 1) model identification (identify if the candidate is a model which needs risk management), 2) model boundaries (define which components are part of the model), 3) model categorization (categorize the model into the group of models with a comparable nature, e.g. anti-money-laundering), 4) model classification (classify the model into in the class of models which require a comparable level of model risk management), and 5) assess the model by a number of sources of risk.

2.4.7 Model Deployment

We observed three deployment patterns at ING:

1. A specialized team creates a prototype with a validated methodology, and an engineering team takes care of reimplementing it in a scalable, ready-to-deploy fashion. In some cases, this is a necessity due to the technical requirements of the model, e.g., when models are developed in Python, but should be deployed in Java (P08, P09, P13).

2. A specialized team creates a model and exports its configuration (e.g., a *pickle*[9] and required dependencies) to a system that will semi-automatically bundle it and deploy it without changing the model (P01, P09).

3. The same team takes care of creating the model and taking it into production. This mostly means that software engineers are part of the team and a structured and strict software architecture is ensured.

[9] A *pickle* is a serialized Python object. `https://docs.python.org/3/library/pickle.html`

Similar to the training environments, machine learning systems are deployed to on-premises environments. A reported challenge regarding the deployment environment is that different hardware and platform parameters (e.g., Spark parameters) can result in different model behavior or errors (P16). For example, the deployment environment may have less memory than the training environment. Furthermore, the resources for a machine learning system are dynamically allocated whenever needed. However, it is not trivial understanding when a system is no longer needed and should be scaled down to zero (P01).

2.4.8 Model Monitoring

After having a model in production, it is necessary to keep track of its behavior to make sure it operates as expected. It implies testing the model while the model is deployed online. The main advantage is that it uses real data. Previous work refers to this stage as *online testing* [57].

The inputs and outputs of the model are monitored while it is executing. Each model requires a different approach and different metrics, as standards are not yet defined. In this stage, practitioners also look into whether the statistical properties of the target variable do not change in unforeseen ways (P11). The model behavior is mostly monitored by data science teams and is still lacking automation (P03, P05, P06, P14). Also the impact on user experience is monitored when the model has a direct impact on users. This is mostly done using A/B testing techniques and can have business stakeholders directly involved (P03, P10).

Teams resort to self-developed or highly-customized dashboard platforms to monitor the models (P15, P16). Within the organization, different teams may have different platforms. While standardization is in development, for now, we have not observed solutions that are used across the organization. A big challenge in making these platforms available is the fact that each problem has different monitoring requirements and considerable engineering efforts need to be undertaken to effectively monitor a given model and implement access privileges (P15).

2.4.9 Testing

Testing is a task that is transversal to the whole development process. It is done at the model level and at the software level.

Testing at the model level addresses requirements such as correctness, security, fairness, and interpretability. With the exception of correctness, we have not observed automated approaches to verify these requirements. A challenge for the correctness tests is defining the number of errors that are acceptable – i.e., the right threshold (P14).

For testing at the software level, unit and integration testing is the general approach. It scopes any software used in the lifecycle of the model (P07). It enables the verification of whether the techniques adopted in the design of the machine learning system are working as expected. However, although unit and integration testing is part of the checklist used for *Model Evaluation*, a number of projects are yet not doing it systematically (P12, P15). As reported by P14, tests are not always part of the skill set of a data scientist. Nevertheless, there is a generalized interest in learning code testing best practices (P12).

2.4.10 Iterative Development

At ING, teams adopt agile methodologies. Three practitioners (P03, P09, P14) mentioned that using agile methodologies is not straightforward in the early phases of machine learning projects. They argued that performing a feasibility study does not fit in small iterations. The first sprint requires spending a considerable amount of time understanding and preparing data before being able to deliver any model. On the other hand, interviewees acknowledge the benefits of using agile (P03, P14). It helps keep the team focused on practical achievements and goals. Another advantage is that stakeholders are kept in the loop (P14).

Typically 2–3 data scientists are working together on the same model. For this reason, issues with having many developers working on the same model and merging different versions of a model have not been disruptive yet.

Feasibility Study

The end of the first iteration is also a decisive step in the project. Based on the outcome of this iteration there is a go/no-go assessment with all the stakeholders, in which the project is evaluated in terms of *viability* (i.e., does it solve a business issue), *desirability* (i.e., is it complying with ethics or governance rules), and *feasibility* (i.e., cost-effectiveness) (P04, P09, P15, P16). This process is well-defined within the organization for all innovation projects. According to P04 and P09, feasibility assessments are essential at any point of the project – it is important to adopt a *fail-fast* approach.

2.4.11 Education

Interviewees indicated multiple ways in which education can be improved to make graduates better machine learning practitioners in the industry. Firstly, data scientists should have more knowledge of Software Engineering and vice-versa (P01, P11, P14, P16). P11 indicates that data scientists with little software engineering knowledge will produce code that is harder to maintain and likely increases technical debt. On the other hand, a software engineer without data science expertise may write clean code, which nevertheless may not add much business value, because of ineffective data exploration strategies (P09).

Another remark by practitioners is that education should put more focus on the process instead of techniques (P08). While graduates are appreciated for their broad sense of the state-of-the-art, they must learn how to tackle machine learning issues in large organizations (P08, P10). Academia knows well how to work with new projects, but in reality, the history of the company affects how to perform machine learning – e.g., integration with legacy systems (P08). Graduates seem to underestimate the efforts needed for data engineering, especially data collection (P03, P09, P12). Also, too much attention lies solely on the performance of models. In reality, over-complex models cannot be applied in organizations, because they tend to be too slow or too hard to explain (P16). These models – squeezing every bit of performance – are great for data science competitions as facilitated on Kaggle, but not for the industry, where more efficient solutions are necessary (P09, P16).

2.5 Data Synthesis

In this section, we answer each research question.

2.5.1 RQ1: How do existing machine learning lifecycle models fit the fintech domain?

To answer this research question, we analyze lifecycle models existing in the literature and adapt them according to the findings observed in our study. We select two reference models, as described in Section 2.2.1: *CRISP-DM* [44] and *TDSP* [32]. The changes we propose can be constrained to this specific case of ING, the fintech domain, or be extendable to general machine learning projects. We justify and define these constrains for each change.

Most stages we observed at ING naturally fit CRISP-DM and TDSP. Similarities between CRISP-DM and the lifecycle of machine learning models at ING are *Business Understanding*, *Data Understanding*, *Data Preparation*, *Modeling*, *Evaluation*, *Deployment*. Similarities between TDSP and the lifecycle at ING are *Business Understanding*, *Data Acquisition & Understanding*, *Modeling*, and *Deployment*. Nevertheless, based on the observations collected in our study, changes to these models are called for.

We propose the changes of CRISP-DM in Figure 2.3. We add three new essential stages: *Data Collection* (as part of *Data Engineering*), *Documentation*, and *Model Monitoring*. Furthermore, we emphasize the feasibility assessment with the "Go/No-go" checkpoint and a sub-stage *Model Risk Assessment*, part of *Evaluation*.

As depicted in Figure 2.4, we adapt the TDSP model to include *Documentation*, *Model Evaluation*, and *Model Monitoring* as major stages. We also emphasize *Model Risk Assessment* (as part of *Evaluation*) and a *Feasibility Study*.

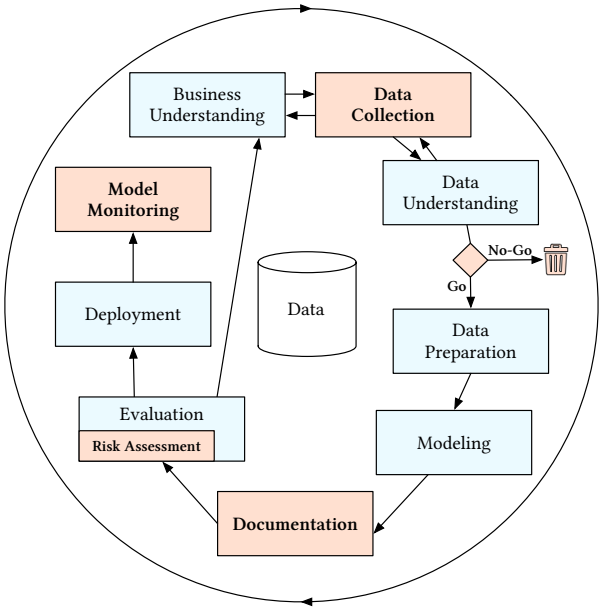

Figure 2.3: Refined CRISP-DM model. Additions in red, with bold text.

The adaptations of the models will be further elaborated upon in the following paragraphs.

Data Collection Although CRISP-DM encompasses *Data Collection* within *Data Understanding* and *Data Preparation*, our observations reveal important tasks and challenges that need to be highlighted. As reported in Section 2.4.3, *Data Collection* requires getting privileges to access data with different criticality-levels and making sure the data is representative of the problem being tackled. Our proposition is that the characteristics observed at ING regarding this phase generalize to any large organization dealing with sensitive data.

Go/No-go or Feasibility Study The aforementioned *Feasibility study* (cf. Section 2.4.10) is an essential part of a machine learning project to ensure projects have everything in place to deliver the long-term expectations. It was a recurrent step observed in our study, which is aligned with the agile approach, *Fail Fast*, promoted at ING and many organizations alike. It may generalize to other cases, depending on the agile culture of the organization.

Documentation In our case, documentation revealed to be a quintessential artifact for a machine learning project. Documentation is the key source of knowledge on how the model is designed, evaluated, tested, deployed, and so on. The documentation is used to evaluate, maintain, debug, and keep track of any other decision regarding the model. It is hard to replace documentation with other strategies because stakeholders

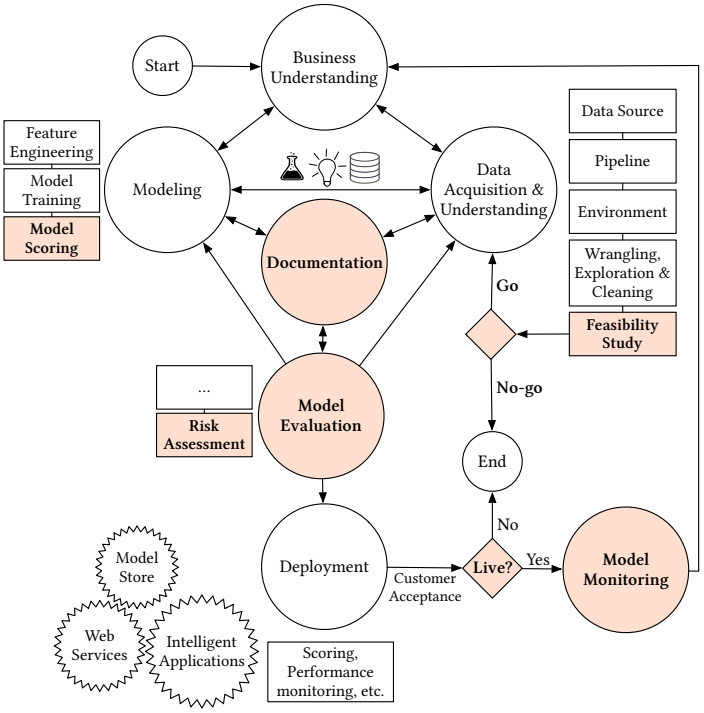

Figure 2.4: Refined TDSP model. Additions in red, with bold text.

with a non-technical background also need to understand the model and have confidence in how the machine learning model is designed. Although documentation is also important in traditional Software Engineering applications, the codebase is usually the main target of analysis from audits. In machine learning, documentation contains important problem-specific decisions that cannot be understood in the code itself. We have no evidence on how this stage generalizes to other organizations, but believe this to be crucial in any highly regulated environment.

Model Evaluation Although the original version of TDSP also included *Model Evaluation*, it was proposed as an activity under the *Modeling* stage. We observed that, when we refer to assessing the performance of a model (i.e., *Model Scoring*), it is indeed part of the *Modeling* activities. However, there is an important part of the evaluation that requires more stable versions of the models. Moreover, it is undertaken with stakeholders that are not part of the *Modeling* loop – e.g., live demos with business managers (cf. Section 2.4.6). Thus, we highlight this part of the evaluation as its own stage. This is also relevant for projects in different domains.

Model Risk Assessment Model Risk Assessment is crucial to any banking or finance organization. Although these companies already have a big history of traditional risk management, it does not cover machine learning models. At ING, this is mandatory for any model.

Model Monitoring Most machine learning models operate continuously and produce outputs online. Our study shows that the natural step after deployment is *Monitoring* – for example, using dashboards – to ensure the model is behaving as expected. *Model Monitoring* is not explicit in neither CRISP-DM nor TDSP, but it is relevant to any domain.

Finally, although not depicted in the proposed lifecycles, *Education* is a stage implicit throughout the whole lifecycle. We observe that universities and courses on machine learning need to provide a more holistic approach to focus on all the different stages of the lifecycle of a machine learning system.

A lifecycle stage that we did not yet observe is the end of life of a machine learning system – i.e., the *Disposal* stage. We presume that a disposal stage is not relevant yet due to the recency of machine learning in fintech.

2.5.2 RQ2: What are the specific challenges of developing machine learning applications in fintech organizations?

We highlighted many challenges of developing machine learning applications in Section 2.4. Most challenges fit in the CRISP-DM and TDSP models. However, two challenges specifically related to fintech and to our extensions of CRISP-DM and TDSP stand out: 1) *Model Governance* and 2) *Technology Access*.

Model Governance is on top of the agenda of the case in this study. A well-defined process is in place to validate regulations, ethics, and social responsibility in every machine learning model. The relevance of this problem to fintech organizations goes beyond machine learning applications: math-based financial models have long been

deployed under well-defined risk management processes. Nevertheless, AI brings the need to revise and recreate model governance that suits the particularities of models that are now automatically trained. Model risk experts are now required to have a strong background in two disjoint fields: 1) *Governance, Risk Management, and Compliance* and 2) *AI*.

Technology Access is the second big challenge in developing AI in fintech organizations. All AI technologies, tools, and libraries need to be audited to make sure they are safe to be used in fintech applications. However, the field of AI is changing very fast with new tools. Industries that want to shift towards AI-based systems need to be able to quickly, yet safely, adopt new technologies.

2.6 Discussion

In this section we discuss the implications of our results and the threats and limitations of the study design.

2.6.1 Implications

We divide the implications of our results under implications for machine learning practitioners, process architects, researchers, and educators.

Implications for Machine Learning Practitioners

Machine learning practitioners have to be aware of extra steps and challenges in their process of developing machine learning applications. Although not mentioned in existing lifecycle models, the undertaking of feasibility assessments, documentation, and model monitoring, are crucial while developing machine learning applications.

Implications for Process Architects

Existing lifecycle models provide a canonical overview of the multiple stages in the lifecycle of a machine learning application. However, when being applied to a particular context, such as fintech, these models need to be adapted. From our findings, we suspect that this is also the case for other fields where AI is getting increasing importance. Process architects for intelligent systems for healthcare, autonomous driving, among many others, need to look at their lifecycle models from a critical perspective and update the models accordingly.

Implications for Researchers

Researchers could focus on solving the reported challenges in the machine learning lifecycle with additional tool support and reveal challenges of the ML lifecycle in other domains by extending the case study to more organizations and different types of industries.

More automation is required for exploratory data analysis and data integration techniques. Automation tools are also needed to help trace documentation back to

the codebase and vice versa. Tools that assist model governance will reduce bottlenecks in the development process and will help to ensure machine learning models comply with regulations.

Furthermore, solutions to challenges in the ML lifecycle should be researched. Software testing needs to be extended and adapted for machine learning software to help effectively test the machine learning pipeline at software-, data-, and model-level. It is also necessary to create holistic monitoring solutions that can scale to different models in an organization. There is a need for strategies to help practitioners select the right set of model scoring metrics. Agile development practices need to be adjusted for AI projects. Tools featuring experiment logs (e.g., MLFlow) ought to propose a holistic solution for version control to keep track of changes in data, changes in scoring metrics, and executions of different experiments.

Implications for Educators

Education of machine learning should focus on the whole lifecycle of machine learning development, including exploratory analysis with a focus on statistics, data analysis, and data visualization. Moreover, practitioners with background on both data science and software engineering are a valuable resource for organizations. This emphasizes the importance of a transdisciplinary approach to AI education [51, 37] and it is congruent with previous work that reports that a Software Engineering mindset brings more awareness on the maintainability and stability of an AI project [4].

2.6.2 Threats to Validity

This subsection describes the threats and limitations of the study design and how these are mitigated. These limitations are categorized into researcher bias, respondent bias, interpretive validity, and generalizability, as reported by Maxwell [29] and Lincoln et al. [25].

Researcher Bias

Researcher bias is the threat that the results of the study are influenced by the knowledge and assumptions of the researchers, including the influence of the assumptions of the design, analysis, and sampling strategy.

A threat is introduced by the fact that participants are self-selected. This means that there might be employees in the company which should be included in the study but are not selected. During the planning phase, participants are selected with different roles and from different departments to have an as diverse starting point as possible. Thereafter, more participants are found by the recommendation of other interviewees and employees until we reach saturation on the information we get from the interviews, i.e. until no new information or viewpoint is gained from new subjects [46].

Respondent Bias

Respondent bias refers to the situation where respondents do not provide honest responses.

The results of the interviews rely on self-reported data. All people tend to judge the past disproportionately positive. This psychological phenomenon is known as rosy retrospection [33]. Furthermore, interviewees who know golden standards from for example literature may tell how things are supposed to be, in contrast with how they are in reality. These biases are mitigated by reassuring interviewees their answers will not be evaluated or judged and by asking them to think about a particular project they have been working on.

Another threat of self-reported data is that interviewees might forget to mention aspects that could have impacted the results. Using a mindmap with relevant topics for the interview, interviewers ask questions to cover all aspects of the machine learning development process to minimize the impact of this threat.

A methodological choice which can form a threat to validity is the fact that interviews are recorded. While the participants themselves permit the recording, they might be extra careful in giving risky statements on the record and therefore introduce bias in their answers. This threat is minimized by assuring the recordings themselves will not be published and all results which will be published are first approved by the corporate communication department.

Interpretive Validity

Interpretive validity concerns errors caused by wrongly interpreting participants' statements.

The interviews are processed by field-note taking and memoing. The primary threat to valid interpretation is imposing one's own meaning, instead of understanding the viewpoint of the participants and the meanings they attach to their words. To avoid these interpretation errors, the interviewers used open-ended follow-up questions which allowed the participant to elaborate on answers. Not everything the participants said could (and should have been) noted. Yet, important remarks could not have been noted in both field-notes and the memos. Our methodology of listening to the interview recordings and amending the memos should remedy this.

Generalizability

Generalizability refers to the extent to which one can extend the results to other settings than those directly studied.

This research is conducted in a large financial institution. Results may not seem generalizable to companies of much smaller size or different nature. A bank may be prone to more regulations than most companies and is dealing with more sensitive data. Still, every company has to comply with privacy regulations like the European GDPR. This suggests that results influenced by more strict regulations and compliance are just as relatable to other industries. Multiple case studies at organizations of different scale and nature are required for establishing more general results.

2.7 Conclusions

The goal of this study is to understand the evolution of machine learning development and how state-of-the-art lifecycle models fit the current needs of the fintech industry.

To that end, we conducted a case study with seventeen machine learning practitioners at the fintech company ING. Our key findings are: 1) CRISP-DM and TDSP are largely accurate; but 2) there are crucial steps missing from the fintech perspective, including feasibility study, documentation, model evaluation, and model monitoring; in particular, 3) the key challenges comprise model governance and technology access.

Our research helps practitioners fine-tune their approach to machine learning development to fit fintech use cases. Additionally, it guides educators in defining learning objectives that meet the current needs in the industry. Finally, it paves the way for next research steps in reducing bottlenecks in the machine learning lifecycle, in particular study tool support for exploratory data analysis and data integration techniques, documentation, model governance, monitoring, and version control.

Chapter 3

dslinter: Static Code Analysis for Data Science

3.1 Introduction

As seen in the case study of Chapter 2, machine learning is shifting from experimental use cases to crucial business processes in the industry. Therefore, the reliability of software applications that rely on machine-trained models has become even more important and failures in these systems are more critical. Faults in the software reduce this reliability and increase failures. Detecting these faults will enable data scientists to address them and improve the quality of their software.

While many automated approaches exist to detect bugs, such as automated test case generation [12], this research focusses on using static code analysis to find probable bugs and enforcing best practices, specifically in code used for processing large amounts of data and modeling in the machine learning lifecycle. The static code analysis will be done on the Python programming language, which is widely used for machine learning [19].

Locating bugs in machine learning code is especially hard due to the nondeterministic nature of machine learning and the entanglement of machine learning systems [43]. When a problem occurs, it is for example not clear if it originated in the data, in code processing the data, in code modeling from the data, or even in the infrastructure on which the machine learning pipeline is running. Static code analysis identifies potential issues or inefficiencies, codifies best practices, and educates users about these practices through tool use [17], all while the data scientist is still engaged with the code and thus enabling them to solve the problems directly.

A call for better code quality in machine learning code has already been made by Wang et al. [50] and Nerush [36]. Wang et al. [50] argue that there is a strong need to programmatically analyze Jupyter notebooks[1], calling on our community to pay more attention to the quality and reliability of notebooks.

The use of automated static analysis has been a software engineering best practice for many years [6]. Therefore, it is no surprise that there are existing static code

[1] Jupyter notebooks consist of a sequence of cells that contain code, visualizations, or text to perform data analysis and machine learning.

analysis tools for the Python language, such as `Pylint`[2] and `Flake8`[3]. Yet, none of these focusses on errors and best practices adherent to processing large amounts of data and creating machine learning models from this data.

The objective of the research in this chapter is to reduce bugs and improve the code quality of machine learning applications. To work towards this objective, we define the following research questions:

1. *What bugs and best practices typical to machine learning can be detected using static code analysis?*

2. *To what extent are these bugs present and best practices not followed in existing open source scripts and programs?*

The remainder of this chapter is structured as follows. In Section 3.2 we give an overview of related work with respect to reducing bugs, risk, and technical debt in machine learning systems. In Section 3.3 we give a little background of linting. In Section 3.4 we describe the developed static code analysis tool and the reasoning, implementation details, and limitations of the designed and implemented checkers. Then, in Section 3.5 we evaluate the implemented linter on a large number of open source scripts. We discuss our findings in Section 3.6. Finally, in section 3.7, we conclude this study.

3.2 Related Work

This section gives an overview of related work with respect to reducing bugs, risk, and technical debt in machine learning applications and using static code analysis for these goals.

3.2.1 Improving Quality of Machine Learning Systems

A study with the goal of improving the quality of machine learning systems is not new.

Several machine learning-specific risk factors are identified by Scully et al. [43]. These risk factors include boundary erosion, entanglement, hidden feedback loops, undeclared consumers, data dependencies, configuration issues, changes in the external world, and a variety of system-level anti-patterns such as glue code and pipeline jungles. The difference with these identified risk factors and our study is that these risk factors are quite abstract, mostly at the system level, and should therefore be accounted for in the system design of the machine learning application. In contrast, our static code analysis tool operates at the source code level and therefore is used while developing the machine learning application. Both should be used together to improve the maintainability and quality of machine learning systems.

To assess the production readiness of machine learning systems, Breck et al. [8] have introduced the *ML test score*: a rubric for machine learning systems with 28 specific tests, including tests for features and data, model development, infrastructure, and monitoring needs. Besides identifying the tests, the authors also give actionable

[2] `https://pylint.org`
[3] `https://flake8.pycqa.org`

steps and advice to pass each test. These tests are also concerning system-level best practices and can be complemented with the static code analysis tool of our study.

The literature survey of Zhang et al. [57] gives a good overview of the current state of testing machine learning systems. The focus of the research community mostly lies in testing supervised machine learning systems and most studies center around the correctness and robustness of those systems. A proper testing strategy with both offline testing (before model deployment) and online testing (after model deployment) combined with static code analysis ensures the best possible quality of machine learning systems.

3.2.2 Static Code Analysis

Beller et al. [6] evaluated the use of static code analysis in open source projects using the programming languages Java, JavaScript, Ruby, and Python. They found that its use is widespread, but not ubiquitous. Zampetti et al. [56] extended on that study by investigating how static code analysis tools are used within a continuous integration pipeline. In their sample of 20 Java projects, 6% of the builds break due to issues found by the used static code analysis tools. Those issues are mostly related to adherence to coding standards. Build failures related to tools identifying potential bugs or vulnerabilities occur less frequently, as those issues are mostly set up to result in build warnings, possibly because of the high number of false positives. This strengthens the goal of keeping false positives as low as possible in the tool of this research, as developers will otherwise give warnings less priority. As continuous integration for machine learning is not yet very mature, we do not focus on that in this research.

Tómasdóttir et al. [47] studied developers‘ perceptions on JavaScript linters. The authors found that developers use linters for maintaining code consistency, preventing errors, saving discussion time about which style to use, avoiding complex code, and for automating code reviews. This current research does not focus on code consistency, but preventing errors is one of the main goals. The authors also found that the linter configuration strategy of the majority of the developers is using existing presets, followed by fitting the current project and using the default linter configuration. These findings together with the finding that creating and maintaining configurations is a real challenge for developers, indicates that we should carefully consider the default configuration of the linter in this current research.

There also has been specific research in using static code analysis for the discovery of security vulnerabilities. Goseva et al. [13] found that, despite recent advances in methods for static code analysis, the state-of-the-art tools are not very effective in detecting security vulnerabilities. Baca et al. [5] showed that developers, given the warnings given by the tools, are not great at detecting false positives and security vulnerabilities among those warnings. A combination of security experience and experience with the tool improved the detection of real security vulnerabilities. This shows that simply enabling a static code analysis tool may not be enough to improve the software the tool scans. Therefore, although we do not focus on security vulnerabilities in machine learning, extensive reasoning for each of our checkers will be described in this research.

3.2.3 Static Analysis for Machine Learning

There has only been a limited amount of studies utilizing static analysis for machine learning.

Hynes et al. [17] have created the `data linter`, a tool that automatically inspects machine learning data sets to identify potential issues in the data and suggest potentially useful feature transforms, for a given model type. The authors introduce types of data lint among three categories: miscoding of data (such as a number encoded as a string), outliers (such as an unnormalized feature), and packaging errors (such as duplicate rows of data). End-user evaluation with eight software engineers showed that the tool can be useful to identify ways to improve model quality through specific feature transforms and to help focus the developer's effort and attention. The static code analysis tool of this current research, `dslinter`, aims at the same goals, but by identifying potential issues in the source code instead of the data.

A call for better code quality in machine learning code has already been made by Wang et al. [50] and Nerush [36]. Wang et al. [50] have collected almost 2000 Python-based Jupyter notebooks from Kaggle.com and checked the code against some of the conventions of the *PEP8 guidelines* with use of the `PEP8 checker` tool. This means that Wang et al. focus on general coding standards using an already existing tool, while we focus on machine learning specific probable bugs and best practices and develop a new tool for that. The general conventions tool found almost 75,000 violations in the collected notebooks. While most of those issues are related to stylistic coding standards, such as the use of whitespaces and blank lines, it is still an indication that Python code in Jupyter notebooks is not well aligned with the recommended coding practices. While checking machine learning code with a general Python static code analysis tool and fixing the issues found can improve the code quality, those tools lack the detection of specific machine learning best practices and probable bugs. To improve the quality of machine learning code more, we extend such a general tool with specific checks focused on errors and best practices adhering to processing large amounts of data and creating machine learning models.

3.3 Background

A static code analysis tool that is used to flag issues in software is called a *linter*. A linter usually works by traversing the abstract syntax tree (AST) of the source code. An AST is a tree that represents the syntactic structure of source code, in which each node represents a construct of the programming language of the source code. Examples of nodes in the AST of a Python program are *FunctionDef*, *If*, and *Return*. An example of code and its (slightly simplified) AST containing these nodes can be seen in Listing 1. While traversing AST nodes, a linter checks for violations of predefined rules. An example of such a rule could be that each function can not have more than 5 arguments, which will be checked when the linter processes a *FunctionDef* node.

There exist two main types of lint: logical lint and stylistic lint. Logical lint is a piece of source code that is faulty, leads to potentially unintended results, or does not follow best practices. Stylistic lint is code not conforming to defined stylistic conventions. Although code with a lot of stylistic lint may result in bugs because the code is not well readable, this research will focus on logical lint.

Listing 1 Example code chunk (left) and its abstract syntax tree (right).

```
def abs(x):
  if x >= 0:
    return x
  else:
    return -x
```

```
Module(
   body=[FunctionDef(
         name='abs',
         args=Arguments(
            args=[AssignName(name='x')]),
         body=[If(
               test=Compare(
                  left=Name(name='x'),
                  ops=[['>=', Const(value=0)]]),
               body=[Return(value=Name(name='x'))],
               orelse=[Return(value=UnaryOp(
                        op='-',
                        operand=Name(name='x')))])])])
```

3.4 Design and Implementation

The static code analysis tool described in this section is available on GitHub[4] *under the GNU General Public License and is uploaded to PyPI*[5].

To answer the research questions, we identified rules of checkers to flag probable bugs and check the adherence to best practices in data science and machine learning code. We also implemented these into a static code analysis tool. In this section, we will explain these checkers and the developed tool called `dslinter`.

As briefly mentioned in the introduction of this chapter, the linter created for this research is for the Python programming language. It is the default programming language for data science and machine learning projects within ING, as well as for most other data scientists around the world [19]. The focus of the linter will be on the Python frameworks `pandas`[6] and `scikit-learn`[7], which are the most popular frameworks for data scientists [19]. As `Spark`[8] shares some communalities with `pandas` regarding DataFrames, the `pyspark`[9] library will also be supported slightly.

Creating a Python linter from scratch would be reinventing the wheel, as there exist already multiple Python linters: `Pylint`[10], `PyFlakes`[11], `Bandit`[12], `pycodestyle`[13], `pydocstyle`[14], and `Flake8`[15]. From these linters, only `Pylint` and `Flake8` are general-purpose linters which allow extensibility by means of plugins. For this research, a plugin is created for `Pylint`, because it has better documentation and more development activity compared with `Flake8`. Furthermore, `Pylint` is a linter itself, while

[4] https://github.com/MarkHaakman/dslinter
[5] https://pypi.org/project/dslinter
[6] https://pandas.pydata.org
[7] https://scikit-learn.org
[8] https://spark.apache.org
[9] https://spark.apache.org/docs/latest/api/python
[10] https://pylint.org
[11] https://github.com/PyCQA/pyflakes
[12] https://github.com/PyCQA/bandit
[13] https://github.com/PyCQA/pycodestyle
[14] https://github.com/PyCQA/pydocstyle
[15] https://flake8.pycqa.org

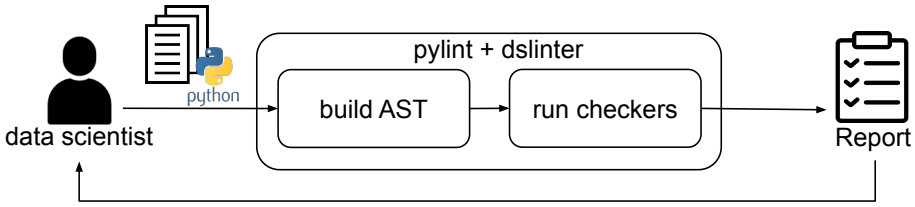

Figure 3.1: Usage and working of `dslinter`.

`Flake8` is actually a wrapper around `PyFlakes`, `pycodestyle`, and `mccabe`[16]. Therefore, `Pylint` is the best linter to extend with a plugin for big data processing and machine learning modeling.

Usage and working of dslinter The created linter in this research is called `dslinter`. It extends `Pylint` with checkers for data science and machine learning. Each checker is a class in the linter which traverses the AST of the source code looking for specific types of probable bugs, best practices, or conventions. The usage and working of `dslinter` can be seen in Figure 3.1. First, the data scientist calls the linter on a set of Python files. Then, `Pylint` and its dependencies take care of building the AST of the source code. After that, `Pylint` and `dslinter` run their configured checkers on the AST. Each violation a checker finds, for example a probable bug, is added to the command line output or a report which is shown to the data scientist. Finally, the data scientist takes the report and fixes all issues, after which the linter can be run again to check whether the issues are resolved.

Identifying rules to check Different resources are used to find rules for the linter which will flag probable bugs and check the adherence to best practices in data science and machine learning code: 1) personal experience in developing data science applications, 2) research papers, 3) grey literature and blog posts, 4) documentation of libraries, 5) most popular questions on question and answer website Stack Overflow[17] with tags *pandas* and *scikit-learn*, and 6) informal talks with ING employees.

A total of six checkers are designed and implemented, which are discussed in detail in the remainder of this section. An overview of the six checkers and their essence can be seen in Table 3.1.

Structure of section The remainder of this section will explain every designed and implemented checker. For each checker, first its essence is given, then the reasoning behind the checker (what problems it solves or prevents and how), then the implementation details are elaborated upon and finally its limitations. Also, for every checker a source code snippet of an example violation and its fix is given.

[16] `https://github.com/pycqa/mccabe`
[17] `https://stackoverflow.com`

Table 3.1: Overview of the six designed and implemented checkers.

Checker (Section)	Essence
Unassigned DataFrame Checker (3.4.1)	Operations on DataFrames return new DataFrames, which should be assigned to a variable.
DataFrame Iteration Checker (3.4.2)	Vectorized solutions are preferred over iterators for DataFrames.
NaN Equality Checker (3.4.3)	Values cannot be compared with `np.nan`, as `np.nan != np.nan`.
Hyperparameter Checker (3.4.4)	For learning algorithms, hyperparameters should be set.
Import Checker (3.4.5)	Check whether data science modules are imported using the correct naming conventions.
Data Leakage Checker (3.4.6)	All `scikit-learn` estimators should be used inside pipelines, to prevent data leakage between training and test data.

3.4.1 Unassigned DataFrame Checker

Essence of the checker: Operations on DataFrames return new DataFrames. These DataFrames should be assigned to a variable.

Reasoning

Most functions that can be called on a DataFrame object (which is a two-dimensional data structure) from the `pandas` or `pyspark` library perform some kind of transformation or filtering on the DataFrame. The DataFrame the operation is called on is not changed, but a new DataFrame with the performed operation is returned instead. This means that the result of the operation should be assigned to a variable, or is lost otherwise. An example of this can be seen in Listing 2. The `abs()` call on the DataFrame returns a new DataFrame with absolute values all elements in the DataFrame, but is not assigned to a variable. Therefore, in code following this chunk, the values of this DataFrame can still be negative.

An exception to this functionality of `pandas` DataFrames is when the *inplace* parameter of the operation is set to True. In that case, the DataFrame resulting from the operation is automatically assigned to the variable of the original DataFrame. It is a misconception that in-place operations will save memory, because a copy of the data is still created. While contributors to the `pandas` library agree the possibility of in-place operations should be removed since 2017[18], it is still supported for the time being.

Listing 2 Example violation (left) and fix (right) of the Unassigned DataFrame Checker.

```
import pandas as pd

df = pd.DataFrame([-1])
df.abs()
```

```
import pandas as pd

df = pd.DataFrame([-1])
df_abs = df.abs()
```

[18] https://github.com/pandas-dev/pandas/issues/16529

Implementation

This checker raises an issue message when a method is called on a `pandas` or `pyspark` DataFrame object, without assigning the result to a variable.

As Python is not a typed language, it is not trivial to statically determine the type of object a method is called on. Therefore, a dedicated TypeInference module is created. This TypeInference module makes use of the `mypy`[19] package to statically infer the type of objects in Python source code. `mypy` uses a combination of type annotations, local type inference, and library stubs to statically infer object types. Luckily, the Python package `data-science-types`[20] contains type information for a lot of modules from `matplotlib`, `numpy`, and `pandas`. This means that `mypy` can infer the type of most objects interacting with these packages.

To reveal the type of an object inferred by `mypy`, source code has to be injected with `reveal_type()` statements and has to be passed to the `mypy` API. Thereafter, the result of `mypy` has to be parsed. Other packages that can be used for static type inference, such as `pytype`[21] and `Pyre`[22], are more sophisticated and allow easier querying of inferred types. Sadly, these tools do not support the Windows operating system yet and would, therefore, limit the linter too much.

When a Module node is visited in the AST by the checker, the TypeInference module is used to infer the types of all objects a function is called on. Then, all calls which are not of the form `a.f()` are filtered out. These calls would result in too many false positives or a too complex implementation if all possibilities a call can be made should be accounted for. Then, if the call is made on a DataFrame, it is evaluated whether the result is assigned to a variable. If this is not the case, the linter will add an ‘unassigned-dataframe‘ message to the source line of the call.

Limitations

While experimenting with the data in Jupyter notebooks, functions like `head()` are often called to see what the data looks like. The resulting DataFrame is then often intentionally not assigned, as the notebook prints the result beneath the code. This problem is partly mitigated by whitelisting certain functions often used this way in notebooks, such as `head()` and `describe()`, but this adds some possible false negatives to the checker. These whitelisted functions are added as a configurable option to the checker, so users can change the list as they see fit.

3.4.2 DataFrame Iteration Checker

Essence of checker: Vectorized solutions are preferred over iterators for DataFrames.

[19] `http://mypy-lang.org`
[20] `https://pypi.org/project/data-science-types`
[21] `https://google.github.io/pytype/`
[22] `https://pyre-check.org/`

Reasoning

As a DataFrame in the `pandas` or `pyspark` library can become very large, iterating through it should be avoided. As stated in the `pandas` documentation[23]: *"Iterating through pandas objects is generally slow. In many cases, iterating manually over the rows is not needed and can be avoided [...]"*. Instead, built-in methods should be used that are vectorized. These built-in methods include most arithmetic, reshaping, join, groupby, comparison, and reduction operations. Besides speed, another advantage of using the built-in methods is that code complexity is reduced, and therefore the code will be less prone to bugs. An example of a violation of this checker and its fix, by using a built-in method, can be seen in Listing 3. The impact of the fix on the runtime can be seen in Figure 3.2, which is created with the `perfplot`[24] package running the code from Listing 3. While the runtime of the code using an iterator grows rapidly when the size of the DataFrame increases, the runtime of the code using a vectorized solution increases barely. Vectorized solutions are so fast because multiple operations can be run from a single instruction.

When a built-in method does not exist for a required operation and the user is dealing with large DataFrames, a custom `Cython`[25] extension could be created to preserve speed. When the effort of creating such an extension does not outweigh the gained speed, a list comprehension can best be created: `[f(x) for x in df['col']]`. When a DataFrame iteration is inevitable, a special comment can be added to the code block to ignore this violation: `## Pylint: disable=dataframe-iteration`.

Listing 3 Example violation (left) and fix (right) of the DataFrame Iteration Checker.

```python
import pandas as pd
df = pd.DataFrame([1, 2, 3])

result = []
for index, row in df.iterrows():
  result.append(row[0] + 1)
result = pd.DataFrame(result)
```

```python
import pandas as pd
df = pd.DataFrame([1, 2, 3])

result = df.add(1)
```

Implementation

With the TypeInterference module already created for the *Unassigned DataFrame Checker* described in Section 3.4.1, the implementation of this checker is quite basic. If the parent of a Call node in the AST is a For node, but the Call node does not occur in the body of the For node, it must be the object iterated over. So, if this Call node is called on a DataFrame, it means there is iterated through a DataFrame and therefore a message is raised.

As a 'bonus', another checker is implemented which checks whether a DataFrame is modified while it is iterated over. This extra checker works by first retrieving all targets (`for target1, target2 in x: ...`) from a For node, then retrieving all

[23] https://pandas.pydata.org/pandas-docs/stable/getting_started/basics.html#iteration

[24] https://github.com/nschloe/perfplot

[25] https://cython.org

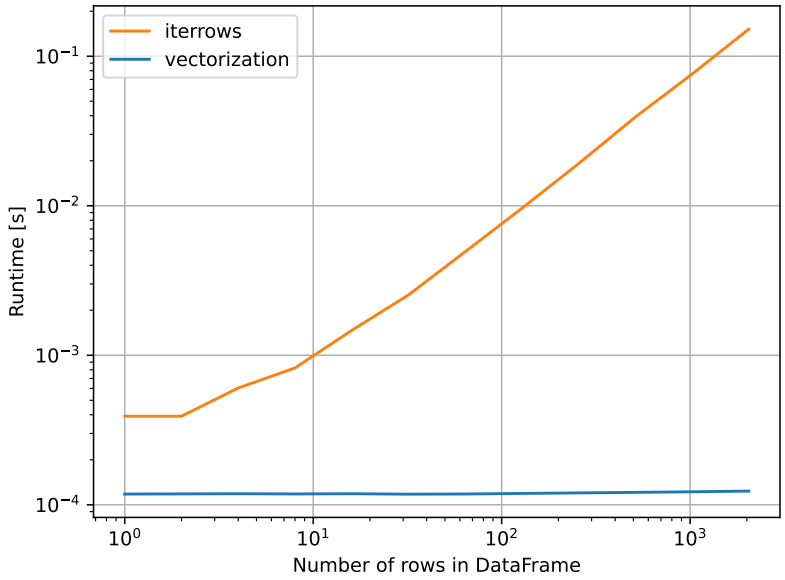

Figure 3.2: Impact on runtime with different techniques from Listing 3.

variables which are assigned to in the body of the For node, and finally checks if a variable name exists in both lists.

3.4.3 NaN Equality Checker

Essence of checker: Values cannot be compared with `np.nan`, as `np.nan != np.nan`.

Reasoning

While **`None`** `==` **`None`** evaluates to **`True`**, `numpy.nan == numpy.nan` evaluates to **`False`**. As `pandas` treats **`None`** like `numpy.nan` for simplicity and performance reasons [10], a comparison of DataFrame elements with `numpy.nan` always returns **`False`**. When a comparison against **`None`** or `NaN` is needed, the functions `isna()` or `notna()` should be used instead.

An example of this can be seen in Listing 4. In the code containing a violation, all values of `df_is_nan` will be **`False`** and therefore may lead to unintentional behavior later in the code.

Listing 4 Example violation (left) and fix (right) of the NaN Equality Checker.

```
import pandas as pd
import numpy as np

df = pd.DataFrame([1, None, 3])
df_is_nan = df == np.nan
```

```
import pandas as pd
import numpy as np

df = pd.DataFrame([1, None, 3])
df_is_nan = df.isna()
```

Implementation

The implementation of this checker is very simple. When a conditional statement is visited in the AST, it is checked whether one of the sides is the expression `np.nan`.

3.4.4 Hyperparameter Checker

Essence of the checker: For learning algorithms, hyperparameters should be set.

Reasoning

Hyperparameters are parameters of learning algorithms used to control the learning process and are usually fixed before the actual process begins. The two main reasons why hyperparameters should be set and tuned are that it improves prediction quality and reproducibility. Tuning hyperparameters is part of the *Machine Learning Test Score* from Breck et al. [8].

Tuning hyperparameters can result in higher prediction quality, because the default parameters of the learning algorithm may be suboptimal for the given data or problem and may result in a local optimum. These parameters directly control the behavior of the training algorithm and therefore have a significant impact on the performance of the model. Examples of such hyperparameters are the learning rate for neural networks and the *C* and *sigma* for Support Vector Machines. Examples of hyperparameter optimization techniques are Grid Search and Random Search, which are also both available in the `scikit-learn` library. These algorithms can be used to optimize any parameter value for an estimator.

The second reason for defining hyperparameters of learning algorithms is that it improves reproducibility. Reproducibility is improved because of (at least) three aspects.

Firstly, while the default parameters of the machine learning library could be perfect for certain problems, these default parameters of the library can change in new versions of the library. When the library is updated to a new version with such changes, the behavior of the model also changes while there are no changes made in the code implementing the model. This means that this can have an impact on the prediction quality, without the maintainer of the model being aware of any changes. Also, when a model is reproduced later on and the default parameters of the library have changed, different results will be obtained. While one can trust in the maintainers of such libraries to not change default parameters too radically, it is better to explicitly set the parameters of the model.

The second aspect involves one particular parameter of algorithms involving randomness: the random seed. Random seeds should always be set to be able to reproduce results and recreate bugs. Otherwise, the result could be different across different runs of the model, even when running the same code and using the same data. It would also be hard to tell if a change in performance is due to a model or data modification, or due to a new random sample. Besides estimators, the random seed should be specified for all steps in the pipeline, e.g., the random split used for cross-validation.

Finally, explicitly setting all hyperparameters allows replication of the model in a different programming language. As seen in the case study in Chapter 2, it can happen in large organizations that models are first created on one programming language, but

for deployment have to be converted to another language. To reproduce the same model in the other programming language, it is crucial to use the same parameters of the learning algorithms. However, libraries from different programming languages could have different default parameters for their algorithms. Therefore, it is better to explicitly set the parameters of the model.

An example of a violation of this checker and its fix can be seen in Listing 5.

Listing 5 Example violation (left) and fix (right) of the Hyperparameter Checker.

```
from sklearn.cluster import KMeans
```

```
kmeans = KMeans()
```

```
from sklearn.cluster import KMeans
```

```
kmeans = KMeans(n_clusters=8,
↪  random_state=0)
# Or, ideally:
kmeans = KMeans(n_clusters=8,
↪  init='k-means++', n_init=10,
↪  max_iter=300, tol=0.0001,
↪  precompute_distances='auto',
↪  verbose=0, random_state=0,
↪  copy_x=True, n_jobs=1,
↪  algorithm='auto')
```

Implementation

The checker works by first retrieving all learning classes and their hyperparameters from a pre-defined list, which is created by collecting the signatures of all learning classes from `scikit-learn`. Then, when a function call in the AST is visited by the checker, it evaluates whether the function name of the call belongs to a learning class. If that is the case, then it is checked whether the function call contains the right keywords or the right number of arguments. A message is raised if this is not the case.

The checker has two modes for checking whether hyperparameters are set. In the *strict* mode, every argument of a call to a learning algorithm of the `scikit-learn` library has to be set. In the *non-strict* mode, only the 'main' hyperparameters have to be set. There exists no strict definition of a main hyperparameter in literature, but there is done research on the importance of certain hyperparameters [39, 49]. The common method used in these studies is given an algorithm and a large number of datasets, determine the tuning of which hyperparameters affect the algorithm's empirical performance most. These studies only look at a limited amount of algorithms, but their results show the following hyperparameters are most important:

AdaBoost: learning rate
GradientBoosting: learning rate
RandomForest: minimum samples per leaf and maximum features
ElasticNet: alpha and lambda
NearestNeighbors: number of neighbors
SVM: kernel, gamma, complexity
DecisionTree: complexity

So in the non-strict mode, these parameters are set as the main hyperparameters. For all algorithms not in this list, defining one parameter is enough to comply with this checker.

Limitations

In workflows where hyperparameter optimization, or parameter search strategies, are used such as Grid Search, learning classes are sometimes initiated without those parameters explicitly set. Instead, these parameters will be set later as a result of parameter optimization. As the checker evaluates whether the parameters are set when constructing the learning class, false-positive violations will be given in this scenario.

3.4.5 Import Checker

Essence of the checker: Check whether data science modules are imported using the correct naming conventions.

Reasoning

Following naming conventions when importing modules will allow all maintainers of a project to quickly see what is exactly going on. For example, looking at the code `pd.read_csv(data.csv)`, it is immediately clear that a `pandas` DataFrame is created from a CSV file if everyone imports the `pandas` library with the alias `pd`. Another reason for implementing this checker is that the *NaN Equality Checker* (cf. Section 3.4.3) and the *Hyperparameter Checker* (cf. Section 3.4.4) depend on imports following the conventions due to technical limitations.

The official documentation of the libraries is used to decide what the actual conventions are. An example of violations of this checker and their fixes can be seen in Listing 6.

Listing 6 Example violations (left) and fixes (right) of the Import Checker.

```
import numpy
import pandas as pand
import matplotlib.pyplot as plot
from sklearn.cluster import KMeans
↪  as km
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
```

Implementation

This checker is the first one implemented and offered a great opportunity to get familiar with creating a `Pylint` plugin and using the Python AST. A Python import statement is represented by an Import node in the AST. When such Import node is visited, the checker evaluates whether either `pandas`, `numpy`, or `matplotlib.pyplot` is imported and the alias is set respectively to `pd`, `np`, or `plt`. When an ImportFrom node is visited, and something is imported from `sklearn`, it is checked whether no alias is set for the import.

3.4.6 Data Leakage Checker

Essence of the checker: All `scikit-learn` estimators should be used inside pipelines, to prevent data leakage between training and test data.

Reasoning

Data leakage is a very important problem in machine learning. It results in overly optimistic performance during testing and bad performance in real-world usage. There are two main sources of data leakage: leaky predictors and a leaky validation strategy. Leaky predictors include data that will not be available at the time the real-world model will create predictions. In a leaky validation strategy information from training data is getting mixed with validation data.

Leaky Predictors An example of a leaky predictor is the inclusion of an *ExitReason* variable if the goal is to predict whether a customer is going to leave the company [38]. All customers who left will have values in this variable, and those who are still in the company will not. Indications of leaky predictors are columns that have a high statistical correlation with the target, or models which have an unexpected high performance.

Leaky predictors can be detected by exploratory data analysis, comparison of modeling results with results of earlier models, and early field-testing [21]. All these methods require some degree of domain knowledge. Therefore, it is not possible to include a detection algorithm for leaky predictors in the static code analysis tool of this research.

Leaky Validation Strategy An example of a leaky validation strategy can be seen in Listing 7, which is an adaptation of the example by Müller et al [35]. In this example, there is no relation between the data and the target, because they are both sampled independently from a Gaussian distribution. Therefore, it should not be possible to learn anything from the data. Regardless, the model results in an accuracy of 91%. This happens because information leaked from the training data to the test data by selecting the best features on the entire dataset, which had by chance a high correlation with the target. This means that while cross-validation is used, information leaked from the training folds to the test folds.

To prevent a leaky validation strategy, all data transformation must happen within each cross-validation fold. This also holds for feature selection, outlier removal, encoding, scaling, and dimensionality reduction. To assure this, the `scikit-learn` library has introduced Pipelines.

Scikit-learn Pipelines Pipelines[26] can be used to chain multiple estimators and data transformers into one. Pipelines serve multiple purposes. The first is that the pipeline executes a fixed sequence of steps in processing the data, with only one call to the pipeline needed. The second is that pipelines can be used to search for optimal parameters of all estimators in the pipeline at once. Thirdly, pipelines help to avoid leaking statistics from test data into the trained model in cross-validation, by ensuring that the same data samples are used to train all transformers and predictors.

[26] `https://scikit-learn.org/stable/modules/compose.html#pipeline`

The last of these purposes is the reason this *Data Leakage Checker* enforces the use of Pipelines when estimators from the `scikit-learn` library are used. An example of the fix for the violation in Listing 7, using a Pipeline, is given in Listing 8. The accuracy of the model on the data is now -0.25%, which is expected.

Listing 7 Example violation of the Data Leakage Checker.

```
import numpy as np
from sklearn.feature_selection import SelectPercentile, f_regression
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

rnd = np.random.RandomState(seed=0)
X = rnd.normal(size=(100, 10000))
y = rnd.normal(size=(100,))

select = SelectPercentile(score_func=f_regression, percentile=5).fit(X, y)
X_selected = select.transform(X)
accuracy = np.mean(cross_val_score(Ridge(), X_selected, y, cv=5))
```

Listing 8 Example fix of the Data Leakage Checker for the violation in Listing 7.

```
import numpy as np
from sklearn.feature_selection import SelectPercentile, f_regression
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline

rnd = np.random.RandomState(seed=0)
X = rnd.normal(size=(100, 10000))
y = rnd.normal(size=(100,))

select = SelectPercentile(score_func=f_regression, percentile=5)
pipe = Pipeline([("select", select), ("ridge", Ridge())])
accuracy = np.mean(cross_val_score(pipe, X, y, cv=5))
```

Implementation

This checker raises an issue message when a learning function is called on an estimator in the `scikit-learn` library. The learning functions are `fit()`, `fit_predict()`, `fit_transform()`, `predict()`, `score()`, and `transform()`. The estimators from `scikit-learn` are all learning classes used for the *Hyperparameter Checker* in Subsection 3.4.4 and the classes in the preprocessing package of `scikit-learn`. All estimators extend the BaseEstimator class.

The difficult part of implementing this checker is that above-mentioned learning functions are often not called directly on a call instantiating the estimator (e.g., `KMeans().fit()`), but on a variable pointing to an estimator instance (e.g., `my_estimator.fit()`). Also, as there is no type information in the `scikit-learn`

library, `mypy` cannot be used to infer the types of those variables. Therefore, a dedicated AssignUtil module is created. This utility module can be used to statically find all expressions a variable gets assigned (e.g. `5` in the case of `x = 5`). When the variable is a function argument (e.g. `(a)` in the case of `f(a=1)`), the utility module looks for the expressions this argument gets assigned in calls to the function. If a variable is assigned an estimator and a learning function is called on this variable, the rule of this checker is violated and an issue message is raised.

Limitations

This checker has both methodological limitations and technical limitations.

The methodology of enforcing the use of Pipelines in the `scikit-learn` library surely prevents a lot of data leakage due to a leaky validation strategy, but as discussed above, data leakage can also be the result of leaky predictors. Pipelines do not prevent leaky predictors in any way. Preventing data leakage is still an open problem in research [48, 21], so it is expected this checker does not solve the entire problem at once.

The implementation of the checker also has some limitations. As no type checking can be used to determine whether a variable holds a `scikit-learn` estimator, evaluating this is limited by the implementation of the AssignUtil module. It checks for direct assignments in parent nodes, but the assignment may also be done, for example, in function definitions in other modules. These will not be found, so the checker will not raise an issue message when the variable on which a learning function is called is assigned to an estimator there.

3.5 Evaluation

This section describes the evaluation of the prevalence of violations of the implemented linter and the precision of the checkers, for which we run `dslinter` on a large number of open source scripts used for data processing and machine learning.

3.5.1 Evaluation Setup

To evaluate the linter, we collected the 1000 most popular notebooks from Kaggle[27]. Kaggle is an online platform with a large community of data scientists and machine learning practitioners, where they can build models in a web-based environment and enter competitions to solve real-world data science challenges. Kaggle has currently over 3 million registered users and 200,000 public notebooks[28]. This makes it an ideal source for analyzing real-world source code from many different data scientists.

Dataset

The notebooks are retrieved from Kaggle sorted by 'hotness', which means that those notebooks are *"scoring highly in things like upvotes and views, or are "all-time"*

[27] `https://kaggle.com`

[28] Jupyter notebooks, usually called just 'notebooks', consist of a sequence of cells that contain code, visualizations, or text to perform data analysis and machine learning.

greats that have been consistently popular on the platform for a long time" [20]. We have published the list of all collected notebooks online[29], so that the evaluation could be replicated in a future study.

The 1000 collected notebooks correspond to 226,129 lines of Python code, with a median of 163 code lines per notebook. Of those 1000 notebooks, 656 have associated tags with them. The five most popular tags are *data visualization* (230x), *beginner* (203x), *eda*[30] (193x), *gpu* (181x), and *tutorial* (152x). A total of 655 notebooks are linked to a competition.

Evaluation Process

An overview of the process to collect and process notebooks is explained in this paragraph and can be seen in Figure 3.3. To collect the notebooks, the Kaggle API[31] is used. First, the names of 1000 kernels, which are either a script or a notebook, are retrieved sorted by 'hotness'. Then, the kernels themselves are pulled from Kaggle. Notebooks are converted to scripts with `nbconvert`[32]. Finally, `Pylint` is run on all scripts with the default checkers disabled and the `dslinter` plugin and its checkers enabled. The reporting functionality of `Pylint` is used to create an overview of the aggregated results.

The report generated by `Pylint` shows the number of violations of each checker in the collected notebooks. A maximum of 30 violations per checker is inspected manually to determine the number of true positives of each checker.

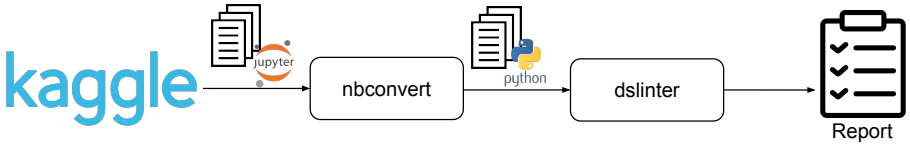

Figure 3.3: Evaluation process of collecting and processing notebooks.

3.5.2 Evaluation Results

Among the 1000 collected notebooks, the checkers from `dslinter` found 2,664 violations. An overview of the occurrences of all violations can be seen in Table 3.2.

The violations that occur most often (1876 times) in the collected notebooks are from the *Data Leakage Checker* regarding not using `scikit-learn` estimators inside pipelines, which can introduce data leakage between training and test data. Manual inspection of 30 random lines containing violations shows support for the reasoning behind the checker. There are some instances (4 in the sample of 30) where standardization and normalization are done before splitting the data into train and test splits. This introduces data leakage because scaling factors are calculated on the full distribution of data. The correct way to do this is to compute the scaling factors on the training data, and perform these factors on both the training and test data. A `scikit-learn`

[29] `https://doi.org/10.5281/zenodo.3909595`
[30] *eda* stands for 'exploratory data analysis'.
[31] `https://github.com/Kaggle/kaggle-api`
[32] `https://github.com/jupyter/nbconvert`

Table 3.2: Violations found in collected notebooks.

Checker	Occurrences	True Positives
Data Leakage Checker	1876	250* **
Hyperparameter Checker	734	611*
Unassigned DataFrame Checker	25	6
Import Checker	14	14
DataFrame Iteration Checker	8	8
NaN Equality Checker	7	7

**Estimated based on 30 random violations.*
***Note that the technical number of true positives is 1876, as the checker checks whether a `scikit-learn` pipeline is used.*

pipeline would have taken care of this. If we assume the same ratio (4/30) occurs in all found violations, 250 of the 1876 violations contain true data leakage. In other instances (3 in the sample of 30), violations of the checker are found where no actual data leakage occurred, because the data scientist handled the correct separation of the training and test set manually before standardization and normalization. However, the code would have been much cleaner if a `scikit-learn` pipeline was used. The remaining manually inspected lines contained calls to estimators without any obvious standardization or normalization of the data, although data leakage could still have been introduced earlier in the script because no pipeline is used.

The second most occurring violation, with 734 occurrences of the *Hyperparameter Checker*, is not defining the hyperparameters of learning algorithms. Manual inspection of 30 random violations shows that some are false positives (5 in the sample of 30), where the learning function is indeed called without defining hyperparameters, but are defined later using parameter search strategies. If we assume the same ratio (25/30) occurs in all found violations, then in 611 of the 734 violations hyperparameters are truly not defined for used learning algorithms. When the option *strict-hyperparameters* is turned on, which means that all parameters of learning algorithms have to be set, 1287 violations are found. Inspection of the notebooks shows that only 17 calls to learning algorithms defined all parameters. This means that most scripts could be improved for better reproducibility.

The 25 violations of the *Unassigned DataFrame Checker* were few enough to manually inspect them all. As Kaggle is also used to educate other people about data science and machine learning, 17 violations are made purposely to show the result of an operation, without the need of using the result for further calculations. Two violations are thrown by calls to libraries imported earlier in the notebook. That means only six violations are true positives and introduced a bug in the script.

There are in total 14 violations found related to import naming conventions by the *Import Checker*. While these violations do not make the code less accurate, readability could be improved if the community conventions are used. The low amount of violations of this category indicates that indeed most data scientists use the same import naming.

Eight violations of the *DataFrame Iteration Checker* are found in the collected notebooks. In most instances, the built-in functions of the `pandas` library could have

been used instead. This results in huge runtime reductions on large datasets, as shown in Subsection 3.4.2.

Finally, seven violations of the *NaN Equality Checker* are found. As comparing a value to `numpy.nan` does always return **`False`** (as `numpy.nan != numpy.nan`), this can result in serious bugs in those scripts.

3.6 Discussion

In this section the interpretation of the results, the implications and limitations of the study, and recommendations for future research are discussed.

The results show that static code analysis can detect and thus prevent probable bugs in data science code. All checkers of the implemented linter found violations in the collected notebooks. Most notable is the large number of violations of the *Data Leakage Checker*, which indicates there are many scripts with possible data leakage and thus over-optimistic results.

It is important to note that the relevance of `dslinter` could be higher than the numbers suggest. Even though the linter has not found certain violations in currently available versions of source code, it does not mean it was never there and the data scientists never struggled with it. A data scientist could have noticed a fault in the program, but not be able to find the origin of the fault for a long time. The linter could have saved the data scientist a lot of time by pointing to the source of the problem right away.

3.6.1 Implications

Data scientists should be aware of the faults and best practices highlighted in this research and should try to prevent these faults from occurring in their code. This will result in more accurate scoring metrics during training, better optimization of hyperparameters, cleaner code, and for these reasons a better functioning program. Data scientists could run linters on their code to be able to address possible faults quickly, even if their projects are still in the experimental phase. When Python, `pandas`, and `scikit-learn` are the language and libraries of choice, `Pylint` extended with `dslinter` could be set up to automatically lint the code while developing data science and machine learning projects.

The implication of this research for software engineers is the notion that general programming language linters can successfully be extended for specific use cases. It is not needed to develop a linter from scratch, as new plugins for existing linters can be developed for the situation the software engineer is working in. The implemented linter of this study can also be extended, as we made it publicly available on GitHub[33] under the GNU General Public License.

The implications for the accountable person of a machine learning application within an organization such as ING, for example the product owner, are that `dslinter` offers first steps to automated validation. It could, for example, be used in the continuous integration pipeline for machine learning applications to ensure and gain confidence that the system is able to accomplish its intended goals.

[33] `https://github.com/MarkHaakman/dslinter`

3.6.2 Limitations

There are some limitations with taking Kaggle as a source for the evaluation of the implemented linter. Kaggle is a platform where individual users can upload their notebooks to share them publicly and enter competitions. This differs from the industry, where code is often peer-reviewed. Furthermore, the skill of the users of Kaggle is not known. Therefore, it is unclear how generalizable the evaluation results are to the industry and larger data science projects with multiple data scientists.

Although the implemented linter has a statement test coverage of 89%, the linter could contain faults itself. This means there could be fewer or more actual violations to the checkers in the collected notebooks.

The checkers themselves can also contain methodological faults, which result in false positives with respect to the faults we claim each checker prevents. Besides the whitelisted functions of the *Unassigned DataFrame Checker*, there could be more operations that do not lead to faults when the result is not assigned to a variable. Regarding the *DataFrame Iteration Checker*, iterating through a DataFrame is sometimes unavoidable and not always wrong. Also, when the *Data Leakage Checker* reports a violation as an estimator is not used in a pipeline, it does not mean there is guaranteed data leakage.

The last limitation of the results is that linter only tells us about a selected number of possible faults while using the `pandas` and `scikit-learn` libraries of the Python programming language. There are probably many faults not covered within the use of these libraries, but also in other libraries such as `TensorFlow` and `Spark`, and other programming languages such as Java and R.

3.7 Conclusions

The goal of this study is to reduce bugs and improve the code quality of machine learning applications. Therefore, we developed a static code analysis tool, `dslinter`, consisting of six checkers that help prevent data leakage, not assigning the result of a DataFrame operation, comparing objects with `nan`, unnecessary iterations over DataFrames, and enforce import conventions and defining hyperparameters for learning algorithms. The evaluation of the linter on 1000 collected notebooks from Kaggle shows that static code analysis can detect and thus prevent bugs in data science code. All checkers of the implemented linter found violations, with most notable a large amount of possible data leakage and calls to learning algorithms where hyperparameters are not defined. Our research helps machine learning and data science practitioners to be aware of some possible faults and best practices in their code and helps them by providing the linter so static code analysis can be run automatically on their projects. We show software engineers that linters can successfully be extended for specific use cases and we encourage them to extend the linter developed in this research.

Chapter 4

Conclusions and Future Work

This chapter gives an overview of the project's conclusions and research directions for future work.

4.1 Conclusions

In this thesis, we studied the machine learning lifecycle and used static code analysis to improve the code quality of machine learning applications.

The goal of our first study, as described in Chapter 2, was to understand the evolution of machine learning development and how state-of-the-art lifecycle models fit the current needs of the fintech industry. To that end, we conducted a case study with seventeen machine learning practitioners at the fintech company ING. Our key findings are: 1) CRISP-DM and TDSP are largely accurate; but 2) there are crucial steps missing from the fintech perspective, including feasibility study, documentation, model evaluation, and model monitoring; in particular, 3) the key challenges comprise model governance and technology access. This research helps practitioners fine-tune their approach to machine learning development to fit fintech use cases. Additionally, it guides educators in defining learning objectives that meet the current needs in the industry.

As the importance of the reliability of machine learning applications became clear in the case study, we performed a second study with the goal of reducing bugs and improving the code quality of machine learning applications. This study is described in Chapter 3. We developed a static code analysis tool, `dslinter`, consisting of six checkers that help prevent data leakage, not assigning the result of a DataFrame operation, comparing objects with `nan`, unnecessary iterations over DataFrames, and enforce import conventions and defining hyperparameters for learning algorithms. The evaluation of the linter on 1000 collected notebooks from Kaggle shows that static code analysis can detect and thus prevent bugs in data science code. All checkers of the implemented linter found violations, with most notable a large amount of possible data leakage and calls to learning algorithms where hyperparameters are not defined. Our research helps machine learning and data science practitioners to be aware of some possible faults and best practices in their code and helps them by providing the linter so static code analysis can be run automatically on their projects. We show software engineers that linters can successfully be extended for specific use cases and we en-

courage them to extend the linter developed in this research.

4.2 Future work

Both studies performed in this thesis call for future research directions.

The case study could be replicated at other organizations to validate the generalizability of the results. These other organizations could be of a different industry (e.g., health), different types (e.g., data-driven by nature), different sizes, or could be even other banks or fintech companies.

Future work could also tackle the challenges found in the case study and thereby reducing bottlenecks in the machine learning lifecycle. Automation support should be developed for exploratory data analysis and tracing documentation back to the codebase and vice versa. Software testing needs to be extended and adapted machine learning software and holistic monitoring solutions should be created that can scale to different models in organizations.

The created static code analysis tool could be extended to support more data science and machine learning libraries. Extra checkers could also be added to support detecting more probable bugs and adherence to more best practices. The inspiration for these checkers could come from close collaboration with data science teams.

Lastly, user studies could be performed to evaluate the perceived usefulness of the linter in the machine learning lifecycle. This can also reveal parts of the tool that should be improved to detect probable bugs and adherence to best practices more effectively.

Bibliography

[1] ISO/IEC/IEEE 15288: 2015. Systems and software engineering–system life cycle processes. 2015.

[2] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupre, Vadim Eksarevskiy, Senja Filipi, Tom Finley, et al. Machine learning at microsoft with ML.NET. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2448–2458, 2019.

[3] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: a case study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 291–300. IEEE Press, 2019.

[4] Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. Software engineering challenges of deep learning. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 50–59. IEEE, 2018.

[5] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg. Static code analysis to detect software security vulnerabilities-does experience matter? In *2009 International Conference on Availability, Reliability and Security*, pages 804–810. IEEE, 2009.

[6] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481. IEEE, 2016.

[7] Lucas Bernardi, Themistoklis Mavridis, and Pablo Estevez. 150 successful machine learning models: 6 lessons learned at booking.com. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1743–1751. ACM, 2019.

[8] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. The ML test score: A rubric for ml production readiness and technical debt reduction. In

2017 IEEE International Conference on Big Data (Big Data), pages 1123–1132. IEEE, 2017.

[9] Pearl Brereton, Barbara A Kitchenham, David Budgen, and Zhi Li. Using a protocol template for case study planning. In *EASE*, volume 8, pages 41–48. Citeseer, 2008.

[10] Pandas development team. Working with missing data, 2020. URL https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html.

[11] Frank A Fasick. Some uses of untranscribed tape recordings in survey research. *The Public Opinion Quarterly*, 41(4):549–552, 1977.

[12] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering*, 20(3):611–639, 2015.

[13] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015.

[14] Mark Haakman and Luís Cruz. Machine learning behind the scenes: An exploratory study in fintech - case study protocol, 2020. URL https://1drv.ms/b/s!AuvX_CBP4YARcBBYdYAOG8qLGIk.

[15] Elizabeth J Halcomb and Patricia M Davidson. Is verbatim transcription of interview data always necessary? *Applied nursing research*, 19(1):38–42, 2006.

[16] Charles Hill, Rachel Bellamy, Thomas Erickson, and Margaret Burnett. Trials and tribulations of developers of intelligent systems: A field study. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 162–170. IEEE, 2016.

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

[18] ING. ING at a glance, 2019. URL https://www.ing.com/About-us/Profile/ING-at-a-glance.htm.

[19] Kaggle. State of data science and machine learning 2019, 2019. URL https://www.kaggle.com/kaggle-survey-2019.

[20] Kaggle. Notebooks documentation, 2020. URL https://www.kaggle.com/docs/notebooks.

[21] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):1–21, 2012.

[22] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. Data scientists in software teams: State of the art and challenges. *IEEE Transactions on Software Engineering*, 44(11):1024–1038, 2017.

[23] Henrik Kniberg and Anders Ivarsson. Scaling agile @ Spotify with tribes, squads, chapters and guilds, 2012. URL `https://blog.crisp.se/2012/11/14/henrikkniberg/scaling-agile-at-spotify`.

[24] Jimmy Lin and Dmitriy Ryaboy. Scaling big data mining infrastructure: the Twitter experience. *Acm SIGKDD Explorations Newsletter*, 14(2):6–19, 2013.

[25] Y.S. Lincoln and E.G. Guba. Naturalistic inquiry. *Newbury Park, CA: SAGE.*, 1985.

[26] Gonzalo Mariscal, Oscar Marban, and Covadonga Fernandez. A survey of data mining and knowledge discovery process models and methodologies. *The Knowledge Engineering Review*, 25(2):137–166, 2010.

[27] Fernando Martínez-Plumed, Lidia Contreras-Ochando, Cèsar Ferri, Peter Flach, José Hernández-Orallo, Meelis Kull, Nicolas Lachiche, and María José Ramírez-Quintana. CASP-DM: Context aware standard process for data mining. *arXiv preprint arXiv:1709.09003*, 2017.

[28] Fernando Martínez-Plumed, Lidia Contreras-Ochando, Cèsar Ferri, José Hernández Orallo, Meelis Kull, Nicolas Lachiche, Maréa José Ramírez Quintana, and Peter A Flach. CRISP-DM twenty years later: From data mining processes to data science trajectories. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[29] Joseph Maxwell. Understanding and validity in qualitative research. *Harvard educational review*, 62(3):279–301, 1992.

[30] Carver Mead and Mohammed Ismail. Analog VLSI implementation of neural systems. 1989.

[31] Tim Menzies. The five laws of SE for AI. *IEEE Software*, 37(1):81–85, 2019.

[32] MicrosoftDocs. Team data science process documentation, 2020. URL `https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/`.

[33] Terence R Mitchell, Leigh Thompson, Erika Peterson, and Randy Cronk. Temporal adjustments in the evaluation of events: The "rosy view". *Journal of experimental social psychology*, 33(4):421–448, 1997.

[34] Steve Moyle and Alípio Jorge. RAMSYS - a methodology for supporting rapid remote collaborative data mining projects. In *ECML/PKDD01 Workshop: Integrating Aspects of Data Mining, Decision Support and Meta-learning (IDDM-2001)*, volume 64, 2001.

[35] Andreas C Müller, Sarah Guido, et al. *Introduction to machine learning with Python: a guide for data scientists*. " O'Reilly Media, Inc.", 2016.

[36] Katerina Nerush. Clean code in Jupyter notebooks, 2016. URL `https://www.slideshare.net/katenerush/clean-code-in-jupyter-notebooks`.

[37] Basarab Nicolescu and Atila Ertas, editors. *Transdisciplinary theory and practice*. TheATLAS, 2013.

[38] Robert Nisbet, Gary Miner, and Ken Yale. *Handbook of Statistical Analysis and Data Mining Applications*. Academic Press, an imprint of Elsevier, London, United Kingdom, 2018. ISBN 978-0124166325.

[39] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research*, 20(53):1–32, 2019.

[40] JB Rollins. Foundational methodology for data science. *Domino Data Lab, Inc., Whitepaper*, 2015.

[41] Michael J Ryan and Louis S Wheatcraft. On the use of the terms verification and validation. In *INCOSE International Symposium*, volume 27, pages 1277–1290. Wiley Online Library, 2017.

[42] Patrick Schueffel. Taming the beast: a scientific definition of fintech. *Journal of Innovation Management*, 4(4):32–54, 2016.

[43] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*, pages 2503–2511, 2015.

[44] Colin Shearer. The CRISP-DM model: the new blueprint for data mining. *Journal of data warehousing*, 5(4):13–22, 2000.

[45] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, pages 120–131, 2016.

[46] Anselm Strauss and Juliet Corbin. *Basics of qualitative research*. Sage publications, 1990.

[47] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. The adoption of javascript linters in practice: A case study on ESlint. *IEEE Transactions on Software Engineering*, 2018.

[48] Feifei Tu, Jiaxin Zhu, Qimu Zheng, and Minghui Zhou. Be careful of when: an empirical study on time-related misuse of issue tracking data. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 307–318, 2018.

[49] Jan N van Rijn and Frank Hutter. Hyperparameter importance across datasets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2367–2376, 2018.

[50] Jiawei Wang, Li Li, and Andreas Zeller. Better code, better sharing: On the need of analyzing Jupyter notebooks. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2020.

[51] Yingxu Wang. Cognitive informatics: A new transdisciplinary research field. *Brain and Mind*, 4(2):115–127, 2003. doi: 10.1023/A:1025419826662. URL https://doi.org/10.1023/A:1025419826662.

[52] Tom Wengraf. *Qualitative research interviewing: Biographic narrative and semi-structured methods*. Sage, 2001.

[53] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[54] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data mining with big data. *IEEE transactions on knowledge and data engineering*, 26(1):97–107, 2013.

[55] Robert K Yin. *Case study research and applications: Design and methods*. Sage publications, 2017.

[56] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344. IEEE, 2017.

[57] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 2020.

Appendix A

Glossary

In this appendix an overview of frequently used terms in this thesis is given.

CRISP-DM (Cross-Industry Standard Process for Data Mining): *de facto* process for developing data mining and knowledge discovery projects [44]. It breaks down a project in six phases: business understanding, data understanding, data preparation, modeling, evaluation, and deployment.

Checker *(in the context of Chapter 3)*: a piece of software that traverses the abstract syntax tree of some source code looking for specific, predefined types of probable bugs, best practices, or conventions.

DataFrame: a two-dimensional data structure used in the Python library *pandas*.

dslinter: a plugin for the Python static code analysis tool *pylint*, developed in the study of Chapter 3 with the aim of finding probable bugs and enforcing best practices in source code of machine learning applications.

Fintech: a financial industry that applies technology to improve financial activities [42].

Fix *(in the context of Chapter 3)*: a change in source code which will resolve the problem found by a checker of a static code analysis tool.

Linter: a static code analysis tool that is used to flag issues in software source code. It shows the source code line an issue occurs in and the name of the issue.

Machine learning lifecycle: the process of developing, training, and serving machine learning applications.

Static code analysis: analysis of the source code of software that is performed without executing the program it analyzes.

TDSP (Team Data Science Process): a modern industry process for developing data mining and knowledge discovery projects [32], which has at a high level much in common with CRISP-DM. It includes four major stages for business understanding, data acquisition, modeling, and deployment.

Violation *(in the context of Chapter 3)*: a piece of source code that contains a problem according to one of the checkers of a linter.