Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Towards Automatically Generating Context-Specific ML Pipelines: A Case Study at adesso SE

Alexander Maisch

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dr. Justus Bogner, Robert Kasseck, M.Sc., Felix Harmsen, M.Sc.
Commenced:	January 14, 2023
Completed:	July 14, 2023

Acknowledgements

At this point, I want to thank those who supported me through the process of this master thesis, be it on a technical or an emotional level.

First of all, I would like to express my deepest appreciation to Dr. Justus Bogner for the supervision of my master thesis, providing feedback and supporting me along the way.

Furthermore, I am also grateful to Robert Kasseck and Felix Harmsen from adesso SE for assisting me in every step of the study and the creation of the paper.
Additionally, I would like to mention the participants of the study and thank them for their willingness to participate and spending of time as well as their openness about the topic.
I would also like to thank Alexander Sturm, Alexander Henka and Alexandra Voit for providing me the opportunity to conduct my study at adesso SE and managing contractual matters.

Lastly, I want to acknowledge the support and emotional assistance my family and friends have given me over the course of the master thesis.

Abstract

Machine learning pipelines are an essential component of modern data science. They play a crucial role in automating the development process of ML models. However, designing and configuring ML pipelines is a complex and time-consuming task. Furthermore, various challenges must be addressed to ensure a successful implementation.

In this case study, a comprehensive approach for generating ML pipelines automatically is proposed, aiming to alleviate the burden of manual pipeline design and enable ML practitioners to focus more on model development and analysis. The ML practitioner should be able to provide a series of configurations that are used to generate an ML pipeline which could act as a base for the further ML workflow.

For this, interviews with five experts from adesso SE were conducted to determine the components and quality requirements of ML pipelines in general as well as project specifications they might depend on. Using the results from these interviews and findings from related work, a prototypical approach was developed. In a second round of interviews with the same interviewees, the prototypical approach was evaluated using usefulness and ease of use as evaluation metrics.

The results of the case study show that the interviewees deemed such an approach to automatically generate ML pipelines useful. The reduction in time to produce first ML models at the start of a project was highlighted. Based on these results, the prototypical approach could be developed further, becoming a useful tool in the ML workflow of every ML project.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Artificial Intelligence (AI) is the new hot topic in software engineering and an incredibly fast-growing market in the last decade. In the year 2022 alone, organizations were 13% more likely to have adopted AI compared to 2021, according to the IBM Global AI Adoption Index 2022 [IBM22]. Because of the increased importance of AI to organizations, more problems in the adoption arise. Besides a lack of a clear strategy for AI, the lack of talent with the appropriate skill sets for working with AI is one of the most significant barriers organizations face in adopting AI, according to Chui and Malhotra [CM18]. Therefore, it is important to improve the productivity of those talents and make their work easier and faster. Furthermore, it is reported that 87% of data science projects never make it into production [Ven19]. This highlights the difficulties teams encounter in data science projects.
As the most prominent field of AI, Machine Learning (ML) takes a primary role in AI-related projects. Therefore, the main focus of research lies on the ML development process. One of the most important parts of the ML development process is the machine learning pipeline [DMRM19]. Using this adoption from traditional software engineering, ML practitioners are able to automate large parts of the ML workflow. But ML pipelines have different components, steps and quality requirements compared to their traditional counterpart. Additionally, setting up ML pipelines can be tedious alongside considering best practices for the development of ML models. Faster creation of ML pipelines with fewer mistakes by the ML practitioner could be achieved by automizing the process. This is the topic this paper covers. Using the context provided by the ML practitioner, an ML pipeline should be generated automatically. The goal of the approach is to speed up the creation of ML pipelines, prevent mistakes made by ML practitioners during the setup and thereby improve their productivity.

To develop and evaluate the approach, this case study was conducted at adesso SE, an IT service provider. For this, the following research questions were defined:

RQ 1: What is important for effective ML pipelines?

- RQ 1.1: Which components are important for effective ML pipelines?
- RQ 1.2: Which quality requirements are important for effective ML pipelines?

RQ 2: How much are ML pipeline steps and quality requirements influenced by the project- or user-specific context?

RQ 3: What is a feasible approach to automatically generate context-dependent, high-quality ML pipelines?

To answer the research question, the study design includes expert interviews to gather more information on ML pipelines, the development of a prototypical approach to automatically generate ML pipelines and a second round of expert interviews to evaluate the developed prototype. Using the gathered information from the first round of expert interviews and findings from the literature, an overview of ML pipelines was created which acted as a base for the consequent prototype development. In the second round of expert interviews, the resulting prototype was rated as easy to use and the approach was deemed useful. The experts made several suggestions for improving the prototype as well as a few suggestions on how the approach could work differently than the prototype did.

Chapter 2 of this paper summarizes the findings from literature concerning the ML workflow, the ML pipeline and about automatically generating ML pipelines. Chapter 3 is about the research design of the study with Chapter 4 covering the results from the study. These include the results from the first round of expert interviews in Section 4.1. Section 4.2 combines the results from Section 4.1 and the findings from literature and provides a general overview about ML pipelines and possible configurations which can influence components and steps. Both of those sections answer the research questions 1 and 2. Section 4.3 covers the development of the prototypical approach and Section 4.4 the results from the second round of expert interviews, evaluation the prototype and the approach itself. These sections cover the answers to research question 3. Lastly, Chapter 5 discusses interpretations and implications of the results as well as limitations of the study while Chapter 6 concludes the paper and gives outlook on the future direction.

Research-related artifacts are accessible via Zenodo [Mai23] while code and other artifacts are part of the corresponding GitHub repository[1].

[1]https://github.com/Scrashdemix/thesis-generating-ml-pipelines

2 Background

This section is about the background of the study's topic. It covers the basics of the machine learning workflow as well as machine learning pipelines, what they look like and what makes a high-quality ML pipeline according to academic papers. These are fundamental concepts necessary to understand this paper. Additionally, the current state-of-the-art of automatically generating ML pipelines is reviewed.

2.1 Machine Learning Workflow

The machine learning workflow is an essential part of machine learning projects. It describes how machine learning is done and how ML practitioners work on the project. The ML workflow is related to workflows or lifecycles defined for data science, such as KDD, CRISP-DM and TDSP.

Knowledge Discovery in Databases (KDD) [FPS96] from the early 90s is "the overall process of discovering useful knowledge from data. Data mining is a particular step in this process". It consists of the following nine steps that are also visualized in Figure 2.1:

1. Learning the application domain
2. Creating the target dataset
3. Data cleaning and preprocessing
4. Data reduction and projection
5. Choosing the function of data mining
6. Choosing the data mining algorithm(s)
7. Data mining
8. Interpretation
9. Using discovered knowledge

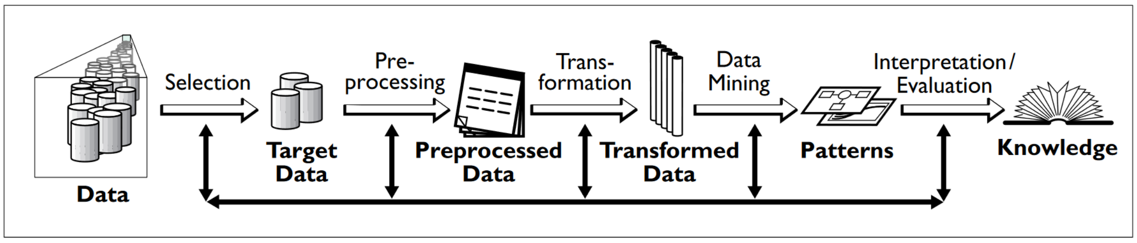

Figure 2.1: Overview of the KDD process [FPS96]

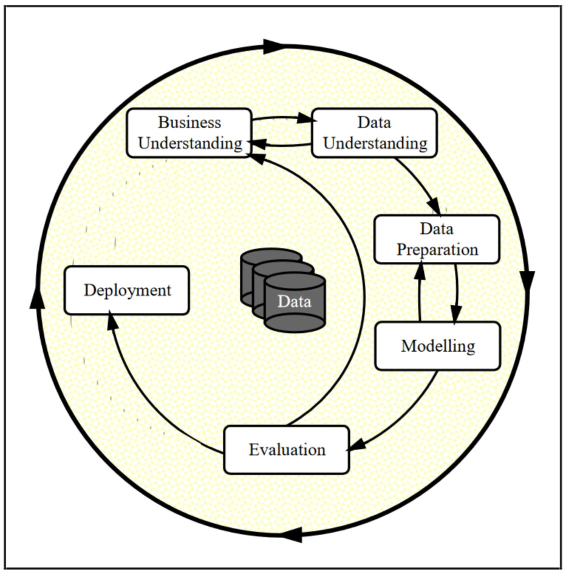

Figure 2.2: Process Diagram of CRISP-DM. Source: [Cro23]

The **Cross-industry standard process for data mining (CRISP-DM)** is a "process model for carrying out data mining projects" [WH00]. The process is useful for planning, documentation and communicating between within and outside project teams. It is visualized in Figure 2.2 and consists of the following six steps:

1. Business Understanding
2. Data Understanding
3. Data Preparation
4. Modeling
5. Evaluation
6. Deployment

The **Team Data Science Process (TDSP)** [Cor16] is “an agile, iterative data science methodology to deliver predictive analytics solutions and intelligent applications efficiently”. It was proposed by Microsoft and is visualized in Figure 2.3. It contains a variety of standardizations besides the process, like a project structure. The process consists of many steps with four major stages: Business Understanding, Data Acquisition & Understanding, Modeling and Deployment.

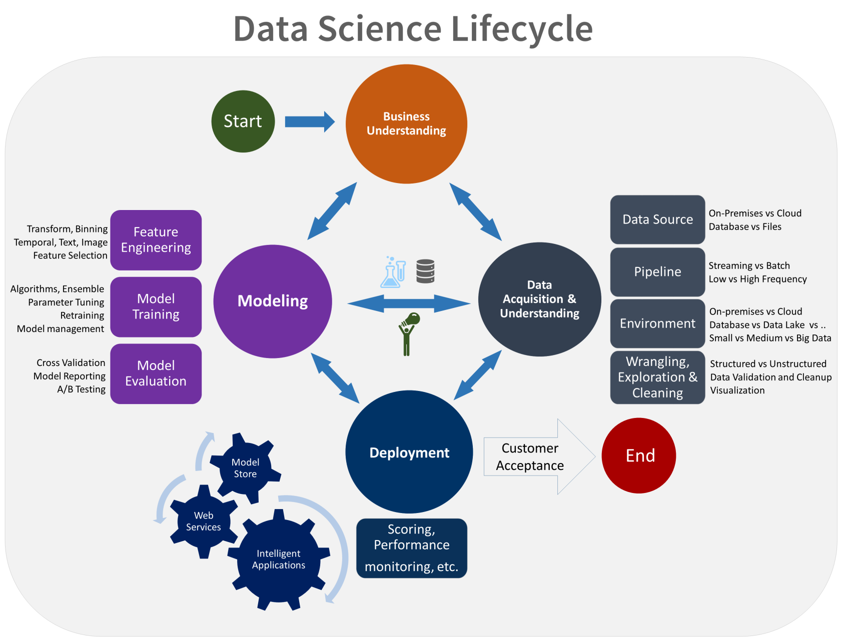

Figure 2.3: Visual representation of the TDSP lifecycle [Cor16]

These data science processes have similarities in their stages or steps. All of them start with some kind of business understanding followed by data-related steps. After this, there is a modeling process which KDD specifically targets at data mining. Lastly, the model is deployed, or in KDD's case at least used. This gives an idea of what the process of data science looks like.

To adapt these to the machine learning workflow, CRISP-DM was enhanced.
The **Cross-Industry Standard Process model for the development of Machine Learning applications with Quality assurance methodology (CRISP-ML(Q)** [SBD+21] was proposed as an answer to the challenges ML practitioner faced in ML projects as there was no established standard process model for machine learning development. The lifecycle is visualized in Figure 2.4.

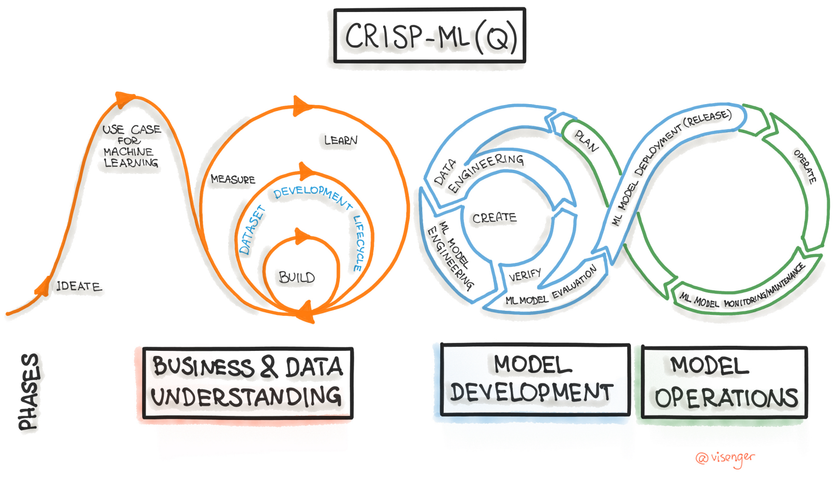

Figure 2.4: CRISP-ML(Q) lifecycle [VKB+23]

Overall, it consists of six phases:

1. Business and Data Understanding
2. Data Preparation
3. Modeling
4. Evaluation
5. Deployment
6. Monitoring and Maintenance

The Business and Understanding phase ensures the feasibility of the project. The tasks include gathering criteria for success by defining so-called “Key Performance Indicators” (KPI). Furthermore, the data should be collected and its quality verified as well as the statistical properties of the data documented.
In the data preparation phase the ML practitioner takes care of the data quality. Feature engineering as well as the standardization of the data is also part of this phase. The standardization not also includes data format but also the normalization of the features’ scale. This step needs to be applied to the test data separately with the same parameter as to the training data.
The modeling phase consists of the model selection, the specialization of the model and lastly the model training. For this, the metrics for evaluating the model should be defined in the first phase. Furthermore, the reproducibility of this phase needs to be ensured.
The fourth phase is the evaluation of the trained models, also called offline testing. It consists of the validation on a test set. Besides model performance, the robustness of the model to noisy and erroneous data should be tested. At the end, the decision whether the model should be deployed should be met automatically based on the success criteria.
The deployment of the model is the integration of the ML model into a software system. It includes the definition of the inference hardware based on the requirements of the model and the project. Additionally, the model should be evaluated in the production environment, also called online testing. The ML practitioner should provide user acceptance and usability testing as well as a fallback plan for model outages. Lastly, a deployment strategy needs to be set up for how the new model is rolled out. This should be done in an incremental way to reduce the cost in the case of erroneous deployment.
The last phase contains monitoring and maintenance of the deployed ML model. This is important because of model staleness which causes the performance of the ML models to drop. If this reduction in performance is detected by monitoring a series of metrics, it could be decided to retrain the model.

Amershi et al. [ABB+19] studied the ML workflow of a variety of software teams at Microsoft and summarized the findings in a nine-stage workflow, visualized in Figure 2.5.
It consists of the following steps: model requirements, data collection, data cleaning, data labeling, feature engineering, model training, model evaluation, model deployment and model monitoring. Furthermore, these can be categorized as data-oriented (collection, cleaning and labeling) and model-oriented (model requirements, feature engineering, training, evaluation, deployment and monitoring). The feedback loops of model evaluation and monitoring as well as the iterations of feature engineering and model training represent the non-linearity of the ML workflow. The findings provide an overall process of the common form of machine learning, supervised learning, for ML practitioners to follow in an ML project.

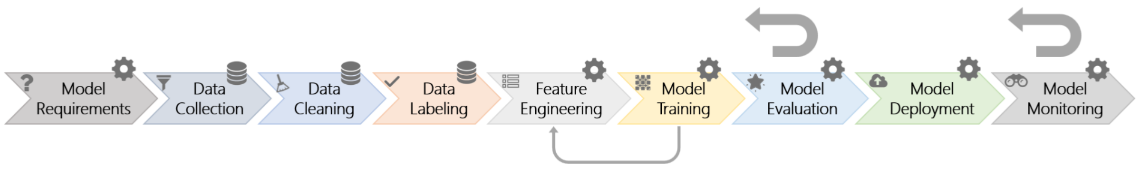

Figure 2.5: ML workflow according to Amershi et al. [ABB+19]

Haakman et al. [HCHD21] reviewed and refined the process models of CRISP-DM, TDSP and the workflow by Amershi et al [ABB+19]. Added components to these models generally include a "Feasibility Study" in which the feasibility of the project is assessed with the intention of adapting a fail-fast approach. Furthermore, "Model Scoring" is added to TDSP and the Microsoft model to the "Model Training" step. These changes are visualized in Figure 2.6. A "Documentation" step is added to all workflows as this was not featured in any of them despite the importance of documenting the model for reasons of audition. This also goes for the "Risk Assessment" which is viewed as part of the "Model Evaluation" and covers the analysis whether the model performs well enough for the given use case to be deployed. Lastly, "Model Monitoring" was added to CRISP-DM and TDSP as this was not part of their process model.

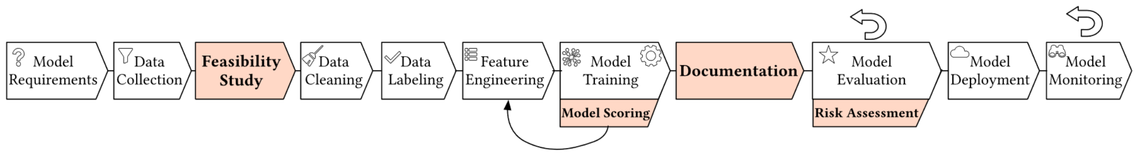

Figure 2.6: Revised ML workflow from Amershi et al. [ABB+19] by Haakman et al. [HCHD21]

Serban et al. [SBHV20] studied the adoption of software engineering practices by teams in machine learning projects, thereby providing more information on what is important in the ML workflow. The findings are summarized in Table 2.1. The training objective and corresponding practices are most important with the top two ranked practices regarding this topic, according to the study. Overall, the practices related to model training were of the most adopted with half of the practices in the category ranked in the top ten of practice adoption. Two other important steps were the versioning of data, model, configurations and training scripts as well as the writing of reusable scripts for data cleaning and merging.

Additional information can be gathered on **ML-Ops.org**, a website containing information on MLOps and the end-to-end machine learning development process created by Dr. Larysa Visengeriyeva et al. [VKB+23]. The workflow described here has similarities with other process models described earlier on. It is structured in three stages: data engineering, ML model engineering and model deployment. The data engineering stage starts with the data ingestion. This is followed by the exploration and validation as well as the data cleaning. These steps improve the overall quality of the data. Next is the labeling of the data in the case of a supervised learning method used. Lastly, the data is split for the model training into training, validation and test set. Following the data engineering is the model engineering stage. Here, a model is trained, evaluated and tested by using the test dataset. In the end, the model is packaged and prepared for deployment. Lastly, the model deployment stage consists of the model serving as well as the continuous monitoring and logging of the model performance in production.

Nr.	Title	Class	Type	Rank
1	Use Sanity Checks for All External Data Sources	Data	N	22
2	Check that Input Data is Complete, Balanced and Well Distributed	Data	N	18
3	Write Reusable Scripts for Data Cleaning and Merging	Data	N	5
4	Ensure Data Labelling is Performed in a Strictly Controlled Process	Data	N	11
5	Make Data Sets Available on Shared Infrastructure (private or public)	Data	N	14
6	Share a Clearly Defined Training Objective within the Team	Training	N	2
7	Capture the Training Objective in a Metric that is Easy to Measure and Understand	Training	N	1
8	Test all Feature Extraction Code	Training	M	23
9	Assign an Owner to Each Feature and Document its Rationale	Training	M	29
10	Actively Remove or Archive Features That are Not Used	Training	N	28
11	Peer Review Training Scripts	Training	M	20
12	Enable Parallel Training Experiments	Training	N	6
13	Automate Hyper-Parameter Optimisation and Model Selection	Training	N	26
14	Continuously Measure Model Quality and Performance	Training	N	4
15	Share Status and Outcomes of Experiments Within the Team	Training	N	7
16	Use Versioning for Data, Model, Configurations and Training Scripts	Training	M	3
17	Run Automated Regression Tests	Coding	T	27
18	Use Continuous Integration	Coding	T	16
19	Use Static Analysis to Check Code Quality	Coding	T	24
20	Automate Model Deployment	Deployment	M	15
21	Continuously Monitor the Behaviour of Deployed Models	Deployment	N	12
22	Enable Shadow Deployment	Deployment	M	25
23	Perform Checks to Detect Skews between Models	Deployment	N	17
24	Enable Automatic Roll Backs for Production Models	Deployment	M	13
25	Log Production Predictions with the Model's Version and Input Data	Deployment	M	19
26	Use A Collaborative Development Platform	Team	T	8
27	Work Against a Shared Backlog	Team	T	9
28	Communicate, Align, and Collaborate With Multidisciplinary Team Members	Team	T	10
29	Enforce Fairness and Privacy	Governance	N	21

Table 2.1: SE practices for ML according to Serban et al. [SBHV20]. They are grouped into 6 classes, together with the practice type and adoption ranks, where N – new practice, T – traditional practice, M – modified practice.

2.2 Machine Learning Pipelines

The ML workflow is often used interchangeably with ML pipelines, although there is a difference between those two concepts. The machine learning pipeline is derived from the pipeline from traditional software engineering. It is a central part of the ML workflow and can automatize many steps. It consists of a directed acyclic graph (DAG) with a series of steps or nodes that automatically execute several steps of the ML workflow.

According to **ML-Ops.org** [VKB+23], the ML workflow consists of three pipelines: the data pipeline, the machine learning pipeline and the software code pipeline.
The data pipeline handles data acquisition and preparation with the sequence of operations of data ingestion, exploration and validation, data wrangling or cleaning, data labeling and at last data splitting.
This is followed by the machine learning pipeline which consists of the steps of model training, evaluation, testing and packaging. Model training includes feature as well as model engineering. Furthermore, there are two types of model training with online and offline training where with the former the model is trained regularly when new data arrives, and the latter is trained with collected data in a batch when the model needs retraining because of decay of performance. Additionally, there is a difference in how the model makes predictions. Batch predictions are made on historical data whereby real-time predictions are made at the time the data is available.
The last part is the deployment pipeline where the trained ML model is deployed. Other than two model deployment strategies like as a Docker container or as a serverless function, there are several model serving patterns. These include Model-as-Service where the model is deployed as a web service accessible via a REST API, Model-as-Dependency where the model is integrated into the software system during the build process, the Precompute Serving Pattern where predictions of common input data are precomputed and stored inside a database, and Model-on-Demand which is similar to the Model-as-Dependency pattern with an own release cycle for the model. Another pattern is hybrid serving where every user has their own model as part of their application which sends their model changes to a server which then aggregates all the changes into a new model.
For the MLOps setup, a series of components are listed that are supposed to ease the development process of ML models. These include a source control for code, artifacts and data as well as test and build services and services for deployment. Further components are the model registry, feature store, ML metadata store and ML pipeline orchestrator.
There are a number of principles mentioned related to MLOps. Two practices that extend DevOps' continuous integration (CI) and continuous delivery (CD), are continuous training (CT), so automatic retraining in production, and continuous monitoring (CM) which includes the monitoring of production data and model performance metrics.
Another principle is the versioning of the model, training script and data. This is an important part of achieving another principle, reproducibility. This is accompanied by the principle of testing the data, model and code. The principle of tracking experiments helps with for example running multiple experiments in parallel.

According to **Google's website on MLOps** [LLC23], there are three levels of automation of MLOps, the operationalization of machine learning.
The lowest level of automation is level 0 where the whole process is manual and script-driven, visualized in Figure 2.7. There is a distinct disconnection between the machine learning and the operational part. Another characteristic of this level is the infrequent release iterations. Missing

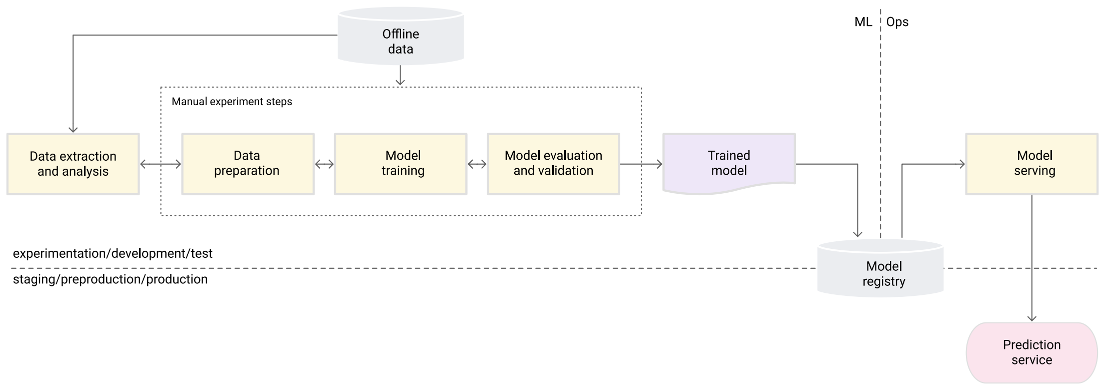

Figure 2.7: ML pipeline automization level 0 according to Google [LLC23]

components include the monitoring in production as well as frequent retraining and continuous experimenting.

The next level is MLOps level 1 where the pipeline in production is automated, visualized in Figure 2.8. This also includes the components of data and model validation, a feature store, metadata management as well as ML pipeline triggers compared to level 0. With this level of automation rapid experimentation and continuous training is possible. There is also an “experimental-operational symmetry” which means that the same pipeline is used in development and production. The project also characterizes a modular structure of code for the components and the pipelines. Another characteristic is the continuous delivery of models as well as the deployment of a training pipeline for automatic retraining. Challenges for the ML practitioner are manual testing and the manual deployment of the training pipeline.

The highest level of automation in MLOps is level 2 which adds a CI/CD automation pipeline, visualized in Figure 2.9. With this, the training pipeline can be automatically deployed and the testing of code and model. It consists of the following stages: Development and experimentation, a pipeline for continuous integration which builds source code and runs tests, a pipeline for continuous delivery which deploys the artifacts, automated triggering of the training pipeline, model continuous delivery that deploys the trained model, and the monitoring of the model performance.

In their paper, **Berthold et al.** [BBG+23] proposed a framework to represent data science patterns using data flow diagrams. Besides the “General Data Science Flow” and “Model Training and Applying” patterns which are similar to pipeline patterns and flows explored previously, the “Data Normalization” pattern explains that data normalization should only occur after splitting the training and testing data to avoid data leakage. The data science design patterns of “Cross-Validation”, “Ensembles” and the “Factoring out Transformations” are also mentioned design patterns.

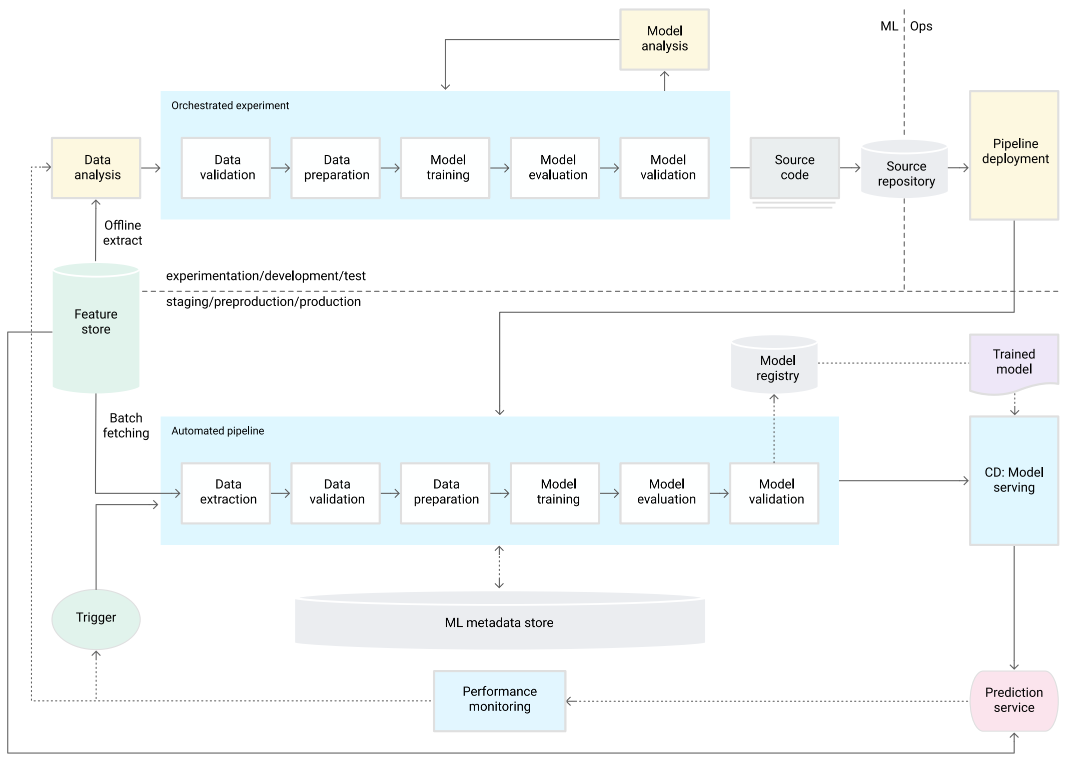

Figure 2.8: ML pipeline automization level 1 according to Google [LLC23]

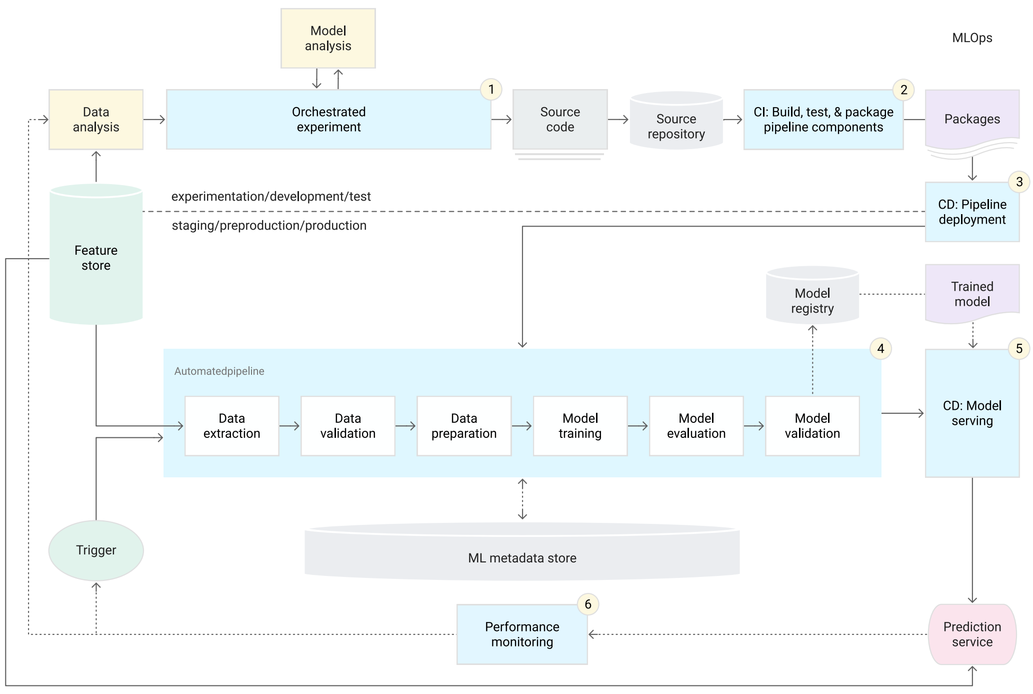

Figure 2.9: ML pipeline automization level 2 according to Google [LLC23]

2.3 Related Work

This section is about related work that is similar to the topic of this study and covers their approaches and how the approach in this study is different to theirs. Papers related to automatically generating machine learning pipelines are mostly about automated machine learning, finding the best performing combination of feature engineering steps and model algorithm.

DeepLine is such an AutoML framework proposed by Heffetz et al. [HVKR20]. It creates a grid with six columns containing data preprocessing, feature preprocessing, feature selection, feature engineering, prediction and combiner. The rows of the grid represent sub-pipelines. The cells are filled with primitives which represent pipeline steps. These are chosen by an agent which is a Deep Q Learning Network (DQN). An example of such a grid is visualized in Figure 2.10.
While DeepLine is created to engineer ML models that achieve the best possible performance, the approach introduced in this paper aims to support the development of models by the ML practitioners by introducing a way of creating ML pipelines fast.

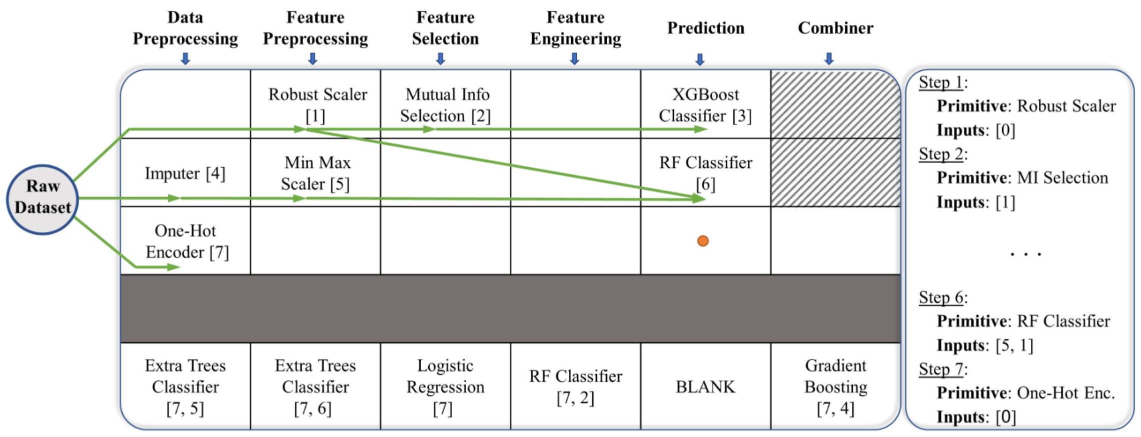

Figure 2.10: Example of the grid-world environment used in DeepLine [HVKR20]

Giovanelli et al. [GBA22] developed a method for generating preprocessing pipelines “as a step towards AutoETL”. They tried to solve the Data Pipeline Selection and Optimization problem (DPSO). For this, they first evaluated the impact finding the best pipeline and found out that no universal pipeline that works best for all considered datasets and algorithms. Additionally, they defined a method that is capable to generate the right order of a given set of transformations, “obtaining effective preprocessing pipeline prototypes” which are further optimized using Bayesian Optimization.

Berti-Equille [Ber19] proposed a method that selects the optimal sequence of data preprocessing steps, called **Learn2Clean**. Given a dataset, a ML model and a quality performance metric, a Q-Learning agent is able to maximize the quality of the ML models result. This involves the selection of data preparation algorithms, data cleaning algorithms and an ML model algorithm depending on the problem with the resulting quality metric and the state of the pipeline being used as input for another iteration of the reinforcement learning agent. The architecture of Learn2Clean is visualized in Figure 2.11.

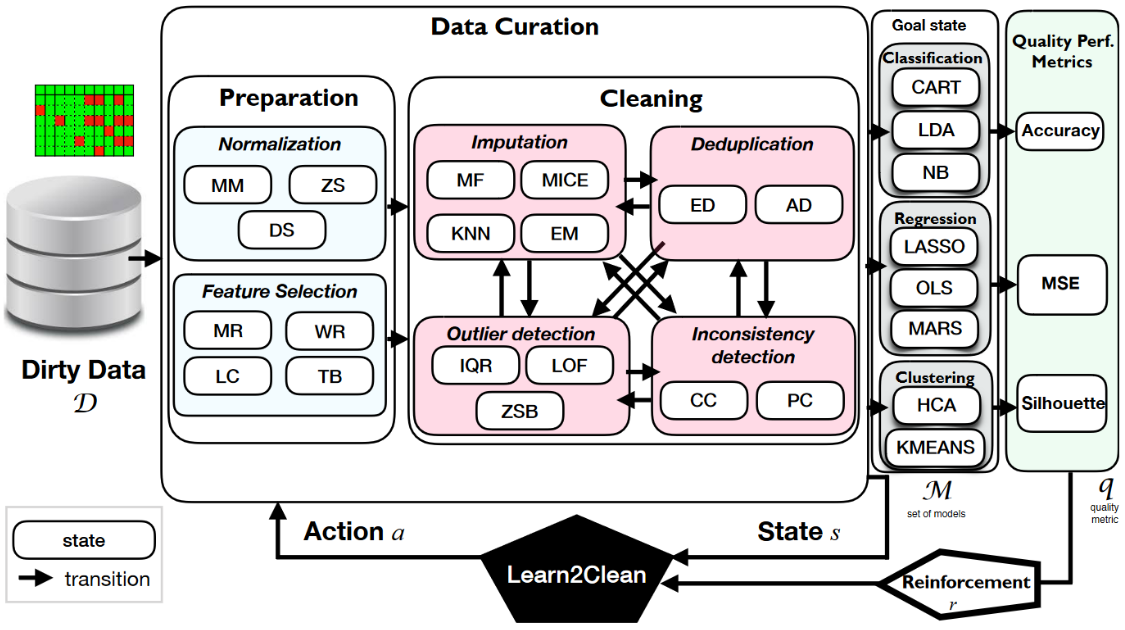

Figure 2.11: Architecture of Learn2Clean

Related work covering challenges of ML pipelines is the paper by **Steidl et al.** [SFR23]. They did a Multivocal Literature Review (MLR) on the topic of continuous development of AI models. In this paper, they proposed a pipeline comprising tasks regarding the continuous development of AI which consists of these four stages: Data Handling, Model Learning, Software Development and System Operations. Interesting for this study are the challenges they listed for each of those four phases. These include the collection and integration of data for the data handling, the model versioning for the model learning phase and the separate packaging of model and software for the software development phase. Lastly, the biggest challenge in the system operations phase was the handling of environment and infrastructure.

Further challenges regarding the ML workflow were studied by **John et al.** [JOB20]. They studied the activities and challenges data scientists face when developing machine learning and deep learning models in software-intensive embedded systems in a multiple-case study and were able to categorize those into seven different phases.
The Business Case Specification phase is similar to the Business Understanding phases of process models described previous. Challenges in this phase include the high costs of infrastructure and qualified personnel, the communication gap between the data scientists and the stakeholders and their high expectations of AI. Furthermore, the lack of data scientists as well as the need for large datasets, especially for deep learning models, are challenges to ML projects.
In the Data Exploration phase, which is equivalent to similar data-related phases in other process models of the ML workflow, there are challenges in the data concerning privacy as well as noise in the data. A further challenge is the lack of domain experts who would be able to explain what features in the data mean and how the data could be interpreted. The last challenge of this phase includes the labeling of the data which needs to be done frequently due to unlabeled data and relies often on domain experts which are not always available.
Feature Engineering includes challenges like the increase of complexity by feature engineering as well as an improper feature selection. These increase the costs and complexity the data scientist

must deal with, in the case of bad feature selection even without any additional value.
The Experimentation phase includes the challenge of the introduction of biases by the algorithm selected as well as uncertainty in the algorithm selection. By selecting algorithms well known to them, they introduce a bias based on their experience. Besides that, the data scientists are uncertain about which model algorithm they need to choose for which use case. In relation to deep learning, there are the challenges in the high complexity of deep learning models as well the need for deep DL knowledge for the data scientists to understand the DL models.
The challenges during the Development phase include how data scientists determine the final model because even in the final stages they realize that they need to go back to earlier stages. Furthermore, the model training environment and corresponding requirements as well as the increased number of hyperparameter settings of DL models are challenges data scientists face. Another challenge is the verification and validation of the models, as safety-critical products need more comprehensive validation that a sufficient degree of certainty that the system performs well can be achieved.
The Deployment phase of the project includes challenges like the lack of DL model deployment as well as integration issues. A further challenge is the deployment in internal systems instead of the customer's infrastructure. Lastly, the need for all data scientists to understand the model developed by a particular data scientist.
Operational challenges include the training-serving skew, the disparity of model performance between training and production, and end-user communication, so the communication between data scientists and end-users who do not understand the model. Further challenges are a model drift and maintaining robustness which is needed to decide whether to retrain or update the model.

D'Aloisio et al. [dDS22] studied the quality attributes of ML pipelines and proposed a new engineering approach for quality ML pipelines in their paper. More interesting for this paper though is the identified quality attributes of the ML pipelines. The basis is the abstraction of ML pipelines consisting of the stages from raw data over data preprocessing, feature engineering, model training-testing and model evaluation to model deployment and monitoring.
One of the most important quality attributes is correctness with the steps from feature engineering to model deployment and monitoring affected by this quality attribute. Another important quality attribute, fairness, affects the data preprocessing as well as model training-testing, model evaluation and model deployment and monitoring. Privacy is a quality attributes for the data preprocessing, model evaluation as well as model deployment and monitoring stage. Computational complexity affects the computationally expensive stages of feature engineering and model training-testing and lastly interpretability has an impact on the last two stages of ML pipelines.

ModelOps [Inc23] is a concept that focuses primarily on "the governance and life cycle management of a wide range of operationalized artificial intelligence (AI) and decision models [...]. Core capabilities include continuous integration/continuous delivery (CI/CD) integration, model development environments, champion-challenger testing, model versioning, model store and rollback".

The paper by **Hummer et al.** [HMR+19] introduced a framework and platform for end-to-end lifecycle management of AI application artifacts, called ModelOps. Challenges identified during a survey include the automation, quality assurance, traceability, risk management and the feedback cycle. Requirements for the proposed framework include challenges in building scalable AI operations. These consist of pluggability, reusability, flexibility, scalability, hybrid environments and fault tolerance.
The implementation of the system works with pipeline configuration as shown in Listing 2.1 where a sample configuration is visualized.

Listing 2.1 Pipeline Configuration Sample from Hummer et al. [HMR+19]

```
models:
  - name: my model 1
    type: tensorflow
    platform: wml
    training data: s3://mybucket/training_images
pipelines :
  - name: train_deploy_pipeline
    tasks :
      - name: Train Model
        type: modelops.task.TrainModel
        model: my_model_1
        harden: true
      - name: Compress Model
        type: modelops.task.CompressModel
        model: my_model_1
        output: CoreML
      - name: Deploy Model
        type: modelops.task.DeployModel
        model: my_model_1
        monitor drift: true
```

Based on the pipeline configuration as well as a task catalog, the ModelOps code generator is then able to generate artifacts/pipelines for a variety of platforms, as seen in Figure 2.12. Pipeline transformers are able to change a pipeline template to include components and steps which are added via the pipeline configuration, as visualized in Figure 2.13.

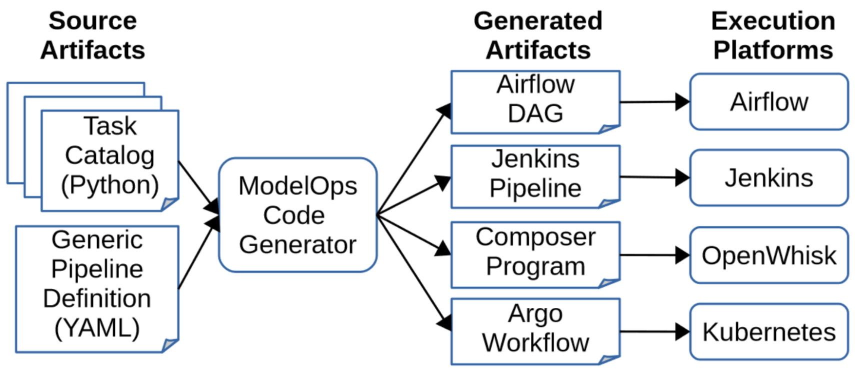

Figure 2.12: Pipeline Code Generation and Execution from Hummer et al. [HMR+19]

The generation of pipelines based on configuration files is similar to the approach of this paper but the proposed ModelOps framework is more about generating pipelines for different platforms rather than generating pipelines for the purpose of speeding up the creation of ML pipelines in general. The difference is also visible by looking at the input of the code generator. While the ModelOps framework still receives a "Task Catalog"written in Python, the approach in this paper is supposed to work without any code written by the ML practitioner besides a configuration file.

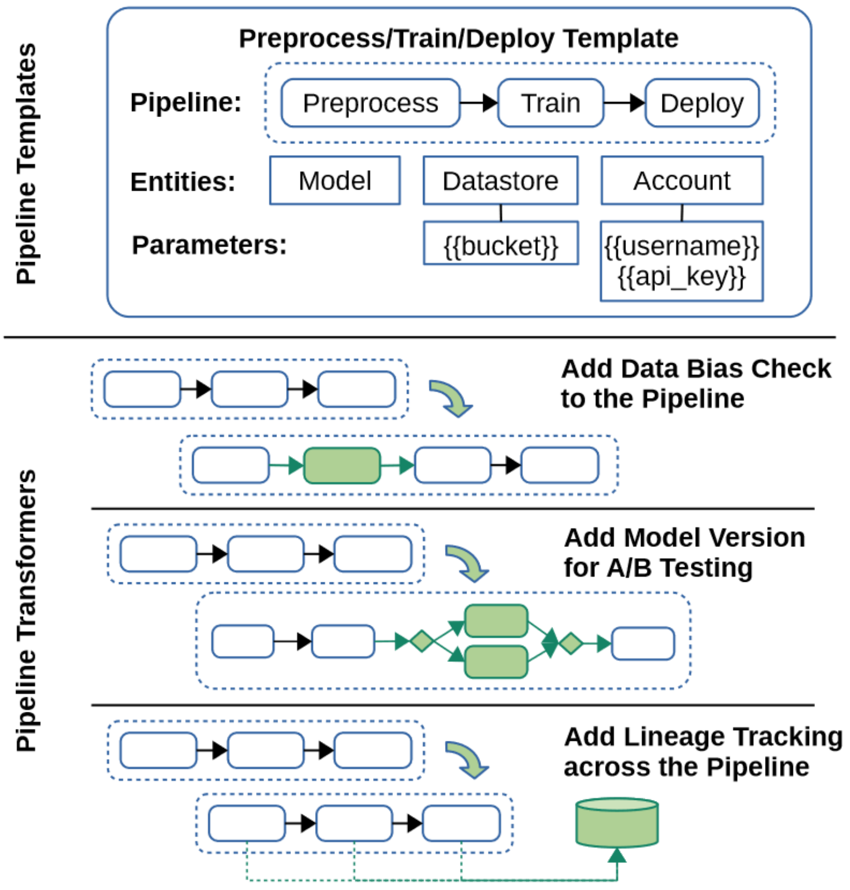

Figure 2.13: Parameterizable Pipeline Templates and Transformers from Hummer et al. [HMR+19]

3 Research Design

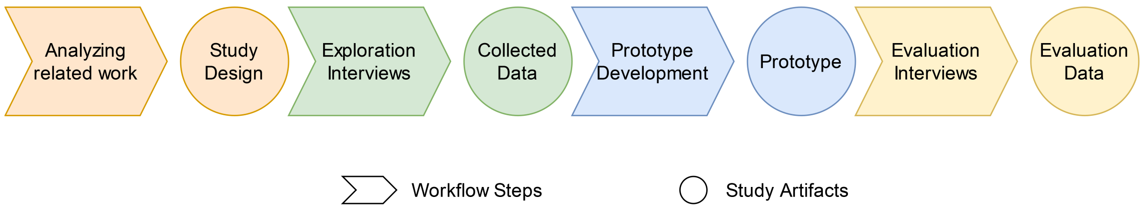

Figure 3.1: The timeline of the study

The chosen research method is a case study. This is a research method which is "conducted in order to investigate contemporary phenomena in their natural context"[RH09], thereby providing insight into how this phenomena interacts with the context. Here, the phenomena is the research topic of how ML pipelines can be automatically generated with the context being adesso SE.
The structure of the case study consisted of three phases, as visualized in Figure 3.1: an exploration phase, the development of a prototypical approach and the evaluation phase. The exploration phase included expert interviews, the exploration interviews, to investigate the topic of ML pipelines as well as their structure and components in more detail. This was followed by the development of a prototypical approach to automatically generate context-specific ML pipelines using the results from the exploration interviews. This prototype and the approach were then evaluated in the evaluation phase consisting of a second round of interviews, the evaluation interviews. This phase was also part of the case study to collect feedback on potential improvements for future development of the prototype as well as the approach itself.

Exploration Phase

In the exploration phase, the exploration interviews were conducted with 5 adesso SE employees. The goal of these interviews was to collect data related to ML pipelines.

The topic of the study was not revealed to the participants until the corresponding section during the interview. Their only knowledge about the topic of the interview was that it was related to ML pipelines. The intention behind this decision was, that the interviewees gave unfiltered views on ML pipelines without them withholding information because they valued it as less important for the approach of generating ML pipelines automatically.
The participants of the study were ML practitioners during the time of the study. Other than their role, there were no further requirements for the participants of the interviews. In more detail, the qualitative data collected from the exploration interviews included demographic data like the participant's role and years of professional experience in ML or MLOps.
The first topic-related section of questions was general questions about ML pipelines. These included quality requirements and challenges regarding ML pipelines as well as components in

general and depending on the situation and context. Additionally, dependencies of components and quality requirements on various factors were investigated. These included dependencies on the dataset, the industry domain, team expertise, type of ML (e.g., supervised learning, unsupervised learning) and type of problem (e.g., regression, classification) as well as other factors the participant could think of.
The second part of the interview was about projects related to machine learning that the ML practitioners were part of. These included the current or last ML project if the participant currently was not part of an ML project. The information gathered about the ML projects was the industry domain of the project's customer, the type of problem being addressed, the requirements and the goals of the project. Further information was gathered related to the ML pipelines used. Besides a basic description of the ML pipeline, questions about quality requirements, special components and encountered challenges during the start of the project, during development and release phases were asked. This section aimed to gather additional information about ML projects and their ML pipelines in relation to the project's context.
At this point of the interview, the interviewer presented the topic of the study to the participant. The topic was introduced with a possible use case to provide the interviewee with more insight into the topic. This use case consisted of a classification task of images of flowers. The ML practitioner wanted the project's pipeline to have the following functionalities: versioning of data, code and model as well as monitoring of performance metrics of the trained model.
Questions in the last section were about the approach of automatically generating context-specific ML pipelines. These encompassed addressed challenges and components which could be considered prime candidates for being automatically generated. The focus on the latter question laid, comparable to other automation tasks, on the potential for time reduction and error prevention. The participant was also asked about the importance of configuration input composition, which attributes the input of such an approach would need as well as what kind of input the participant preferred for this approach. Lastly, the ML practitioner was asked whether he would trust an automatically generated ML pipeline component and the reasons for the answer.
The interviews were recorded and manually transcribed afterward before the record was deleted. This transcription contained information on what the interviewees said during the interviews in the form of notes. The analysis of the interviews was conducted using open coding, summarizing the collected data and noting what information was mentioned by which interviewee or even multiple ones.

Prototype Development

Following an analysis of the results from the exploration interviews, a prototypical approach was developed to investigate the topic as well as its benefits, downsides and improvements further.
Based on the data collected, a general overview of ML pipelines was created. This included important components as well as ML pipeline steps, describing the workflow of the ML pipeline.
Based on the general overview of ML pipelines, a prototypical approach was developed. This was done using the programming language Python[VD09]. A pipeline orchestration framework was chosen for which the prototype could generate pipelines. To comply with the time limits of the study, the focus of the prototype laid on components and pipeline steps which were the most important to show the functionality of the approach.

Evaluation Phase

In the second round of interviews, the evaluation interviews, the prototypical approach was evaluated by the same participants of the exploration interviews to ensure that the analysis results from the exploration interviews matched the expected functionalities and structure of the prototype. Here, the participants were asked to evaluate the usefulness and the ease of use of the prototypical approach, according to the model Davis [Dav89] and give feedback on possible improvements in the future.
Before the interview, the participants were asked to prepare for the practical part of the interview. This involved installing the correct versions of the necessary packages for the prototype as well as creating an example ML project.
Before trying out the prototype, the participants were asked about their experience with the chosen pipeline orchestration and experiment tracking frameworks, preferably in years of professional experience. Following this, the interviewees were able to experiment with the prototype and the generated ML pipelines for 10 minutes. During this time, the interviewer first explained where the participant was able to find documentation on input and execution of the prototype and answered questions from the participants. The interviewees were then tasked to complete an exercise to get more familiar with and use more functionalities of the prototype. This included extending the given example configuration file and running the prototype with it as well as observing the resulting ML pipeline and artifacts created after a run of this pipeline. The participants had 20 minutes each to complete the exercise.
After the exercise, the participants filled out a survey about the usefulness of the approach itself, i.e. if the prototype would be a more complete product, and the ease of use of the prototype specifically, collecting quantitative data in the process. The questions and the ordinal-scaled answers of the survey originated from the work by Davis [Dav89].

Lastly, the interviewer asked about potential improvements of the approach and the prototype. First, the interviewees were asked about general improvements and whether the prototype should work differently. Following this, the interviewer asked more specific questions about improvements. Firstly, the participants should provide feedback on the interaction with the prototype. This included feedback on how the user input could be improved and what kind of user input the participant preferred and feedback on the program execution, more specifically how the generation process should be preferably started and what output format or kind of output the interviewee preferred. Questions about improvements of the resulting pipeline were asked at the end. These included questions about the structure of the resulting pipeline, namely the participants' thoughts and feedback about improvements. After that, feedback about the existing data and feature engineering algorithms was collected and which algorithms the participants wanted to be added. The interviewees were also asked to mention their opinion on a general AutoML approach as part of the ML pipeline. Following this, the interviewer asked the participants about a model deployment pipeline which was not part of the prototype. Here, the participants answered on how important such a pipeline was to them and what functionality this pipeline needs. The last question involves the participant mentioning tools and platforms the generator should provide support for.

4 Results

4.1 Exploration

This section covers the results from the first round of expert interviews, the exploration interviews, and provides answers for the research questions 1 and 2.
The exploration interviews with the 5 adesso SE employees took approximately 70 to 80 minutes, with the interview of Participant P5 taking over the full scheduled length of 90 minutes.

Demographics

All the participants called their role “data scientist”. Still, there were some differences in the tasks they were doing. While P1 mentioned the whole ML lifecycle from ETL over model engineering to model deployment, P4 specifically mentioned proof-of-concept and P5 mentioned MLOps and model deployment as their tasks. The professional experience of the participant ranged from 1.5 or 2 years to 6 or 7 years with P1, P2 and P3 with similar years of experience with 3.5 to 4.5, P4 with 1.5 or 2 years of experience while P5’s professional experience reached 6 or 7 years.
Table 4.1 summarizes the demographic information about the interviewees.

Projects

The projects which the participants were part of included a variety of use cases. Summaries of these are also included in Table 4.1.

P1’s project provided a recommendation system for a telecommunication company to recommend customers who could be contacted for call and mail campaigns. Customers got selected as potential candidates for sales for call campaigns. A positive outcome of a call was registered as a success for the recommendation system. During mail campaigns, the customers received mail newsletters with personalized content. Here, the click rate on this content was measured as a rate of success for the recommendation system. The customer provided a custom framework for deploying a production pipeline with the ML practitioner being able to configure data acquisition and data preprocessing steps. The trained ML model meanwhile was stored inside a Git repository and updated or replaced by adding it via a Git pull request. The data was stored in a data warehouse as data marts. There were two sets of data stored as data marts: the customer/profile data and historical events for each customer. A databricks workflow was scheduled from the data warehouse for preprocessing and training the model. The result was a Python package containing configurations for the customer’s custom framework, like data preprocessing steps, and the trained model. The training process ran on

Participant	Role	Years of professional experience	Project
P1	Data Scientist	3.5	Recommendation system for recommending customers for call and mail campaigns
P2	Data Scientist	3 years ML	Time series forecasting of demand for loans and subsidies
P3	Data Scientist	4.5	Automation of subsidy allowance using optical character recognition
P4	Data Scientist (especially proof-of-concept)	1.5-2	Currently not part of an ML project, before that mainly proof-of-concept ML projects
P5	Data Scientist (MLOps and model deployment)	6-7	Object recognition using open source models

Table 4.1: Demographic information about the participants and their current project

the customer's cluster infrastructure. The custom framework scored the trained model and collected data used for monitoring. This happened via Apache Hive [Apa10] tables which were automatically filled daily with data regarding the performance of the model, feature analysis and data drift.

P2 and P3 were part of different projects with the same customer, an investment and development bank. The project P2 was part of was about predicting the amount of money needed for providing loans and subsidies for the bank's customers. This project tried to solve the problem of forecasting this time series by using supervised learning. The forecasting needed to be done for multiple time periods, e.g., 1, 2, . . . months. For that, there was one model for each time period. The ML pipeline was structured as follows: the data preparation contained steps like data acquisition, the merging of the datasets, data inspection and the handling of missing values followed by data preprocessing and the train-test split. The data preparation ended with an inspection for seasonality, data leakage prevention and feature engineering. This was followed by the model training phase by the usage of grid search and hyperparameter tuning. The validation of the trained model was the last step of the ML pipeline.
Meanwhile, P3 was part of a project where the goal was the automation of subsidies allowance. For this, Azure Cognitive Services were used for optical character recognition on the loans and subsidies applications. The usage of the publicly available AI service was the only part related to machine learning. With the extracted information, it was decided whether the application could be automatically allowed or manually handled by a human person. A requirement for the program was the restriction of execution time. The pipeline of the project contained no machine learning but still handled data. Challenges were the heterogeneity of the data along with a multitude of data sources and the problem of data leakage.

General ML Pipeline Components

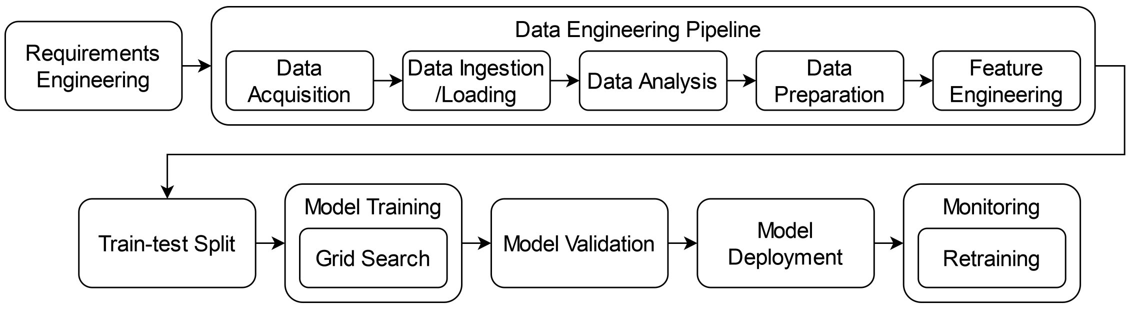

Figure 4.1: Results of the exploration interviews regarding general ML pipeline components

This section is about the components mentioned during the “General” section of the exploration interviews. A summary is visualized in Figure 4.1.

While nearly all participants started with a data pipeline as the first part of the ML workflow, P4 began with requirements engineering. This involved including the customer’s goal in the development process as well as looking at the current process which was the topic of the ML project. The emphasis on requirements engineering might stem from the fact that P4 primarily was part of proof-of-concept projects which seem to focus more on industry domain and customer satisfaction. This part often appeared in existing literature as part of the ML workflow rather than ML pipelines.

Data-related tasks were mentioned by all participants in varying frequencies and extensity. Because P3 mentioned the concept of a data engineering pipeline that fits the described tasks in extensity, the concept will work as a workflow description for those kinds of tasks for the rest of the paper. Because the tasks of a data pipeline were mentioned by all participants, this must be an essential part of an ML pipeline. The beginning task of a data pipeline was data acquisition as well as data ingestion or loading. With all participants except P4 mentioning these steps, there was no difference made between data acquisition and data ingestion or loading. Here, P1 specifically mentioned on-demand data acquisition, which provided the ML practitioner with more flexibility for pipeline execution. While data acquisition was about collecting data from various sources, this step was more part of the ML workflow than an automatically executable ML pipeline in contrast to data ingestion/loading.

All participants expressed their need for some kind of data preprocessing as part of an ML pipeline. This preprocessing contained a variety of computations. While most of the participants called these steps “data cleansing” or “data cleaning”, “data preparation” and “data preprocessing”, P3 gave more insight into what kind of computations are important. Here, combining and transforming data, handling outliers and enriching data were mentioned. Besides improving data quality, enriching the data signifies a need for a higher data quantity, especially in projects where data was scarce. Tool support for data analysis was also mentioned as a critical part of an ML pipeline. Data visualization took a major part of this phase of the ML workflow. Investigating data distribution, feature importance and correlation between features support the ML practitioner in improving the data quality and choosing the data preprocessing steps for that. Support for data visualization thereby was a crucial part of the ML workflow and could be part of an ML pipeline, either by tools visualizing the data or pipeline steps which generate data visualizations.

P3 and P4 also mentioned feature engineering as part of the data preparation phase, while P4 especially set it behind an initial model training phase as a way of improving data quality and selection. Another task of the data preparation of the ML pipeline was data quality assurance. This was important for P2 and P4. The assurance of data quality and data format is very important for practicing data science as the machine learning model can only achieve the best metrics depending on the data quality. P4 added that this task was mainly handled manually in proof-of-concept projects and automated in the latter stages.

The train-test split was mentioned as an own step of an ML pipeline by P2 in a later section and P3. By splitting the data into a dataset for training the model and one for testing the model, the ML practitioner could test how well the model works with unseen data. Interestingly, this step was mentioned as part of the data preprocessing phase by P2.

For model training, there was a kind of grid search mentioned by P2 and P3. By listing a set of machine learning algorithms as well as a set of parameters for each, it was possible to train the best model with the best-performing algorithm and hyperparameters. This can be seen as AutoML with the automatic search for the best combination of machine learning algorithm and hyperparameter. P4 has not heard about the approach but would still consider it a great help. For him, training multiple models was one of the first parts of an ML project. This might indicate that model selection and training are of lesser importance in an ML pipeline.
After the model training the best-performing model was tested using unseen data in the model validation step. This step was mentioned by all participants to various extents. For P1 scoring the model using batches of data was an important step, while for P5 using primarily publicly available models, model validation was a step for checking whether the model could be used for the project's use case. A report on model quality and accuracy was common sense among the participants. P4 also added that the kind and interpretation of metrics and metrics results depend on the data. An example was given in a classification task with 98% of the data having one label, the model could be right 98% of the time if it chose to always output that label.

At last, there was the model deployment step. While all participants except P4 mentioned this step, P1 and P2 went into a bit more detail. They mentioned the deployment of a "prediction pipeline" and a "pipeline in production". These were pipelines that validate and prepare the incoming data before they were fed to the model. Another interpretation could be the deployment of the training pipeline that could be automatically triggered to run and retrain the existing model, similar to the level 1 and 2 automization mentioned in [LLC23]. Because it was not made clear, what those terms mean exactly, the "prediction pipeline" covers the deployed pipeline which prepares the data before the model makes the prediction, while "pipeline in production" is the pipeline which covers automatic retraining for the rest of the paper. In the following, only the model is mentioned in the context of deployment but technically this can also be replaced by a prediction pipeline. Independently of whether a prediction pipeline or simply a model is deployed, the type of deployment is another matter. Several types were mentioned as examples: the model could be exported as an artifact and be part of the software directly (statically embedded), the model could be accessible via a REST API and the model could be available as a batch job. Furthermore, P1 mentioned the difference between the deployment of batch and stream processing use cases.

The deployed model should be continuously monitored as all participants except P4 added. P2 even mentioned a monitoring pipeline which not only monitored the data, looking for data and feature drift, but also checked how well the model worked in production.
Another task was monitoring the predictions by the model and its distribution as well as performance

metrics of it. These indicated when the model needed to be retrained. Retraining was another factor mentioned by nearly all participants. Metrics related to fairness and bias should also be monitored to prevent biased predictions by the model, mentioned P3. Monitoring was more part of the ML workflow than an ML pipeline though.

Even though it was not asked specifically, some technologies were mentioned. These technologies include libraries and frameworks like Pandas [McK10], TensorFlow [The15] and LGBM [KMF+17] but also tools like Kedro [The23a], Hadoop Cluster [Apa06], PySpark [Apa], Databricks [Dat13] and MLflow [The18b].

Generally, the mentioned components and steps of ML pipelines were in line with steps of the ML workflow mentioned by Amershi et al. [ABB+19]. While the model requirements step really only was mentioned by P4, there was a focus on data-related steps among all participants. These steps were split in data collection, data cleaning, data labeling and feature engineering, as described in Section 2.2. The data labeling step was not prominent in the expert interviews which might be because of it is limited to supervised learning methods. The model-related components of the model by Amershi et al. were the same as the components and steps mentioned during the interview.

Quality Requirements

Quality Requirement	Mentioned By
Testability	P3, P5
Flexibility	P2, P3
Modularity	P2, P3, P5
Pipeline Performance	P3
Good documentation	P5
Reusability	P3

Table 4.2: A summary of mentioned quality requirements

Quality requirements describe characteristics of ML pipelines that the ML practitioners saw as important for their work. There were three, several times mentioned, quality requirements with further three only mentioned by one participant each. They are summarized in Table 4.2
One mentioned quality requirement was testability. Being able to automatically test singular components with unit tests but also the interoperability between steps using integration tests, was an important requirement for ensuring the quality of the pipeline.
Another mentioned quality requirement was flexibility. Being able to change steps and the structure of an ML pipeline as well as other configurations easily, made the work for the ML practitioner easier. P2 mentioned in this context that generic tools often did not have enough functionalities for most real-life use cases.
The last of the multiple mentioned quality requirements was the structure or modularity of the ML pipeline. The project having a clear and understandable folder structure with hyperparameters extracted in one separate or multiple configuration files seemed important when it came to the structure of the ML project, as P2 and P3 mentioned. The ML pipeline as well as the project itself were supposed to have a modular structure according to P5.
Other quality requirements included pipeline performance, mentioned by P3. This meant not

the performance of the resulting model but the execution time, which should be sufficiently low depending on the amount of data and preprocessing steps. This is also mentioned in the work by d'Aloisio et al. [dDS22] where this was called "computational complexity".
Another less-mentioned quality requirement was good documentation and clean code, proposed by P5. A well-documented pipeline and project enabled the ML practitioner to faster start building the ML pipeline and doing data science with less time needed for understanding how the pipeline could be changed according to the wishes of the ML practitioner.
Finally, P3 mentioned reusability as a quality requirement. Especially, for projects with similar use cases reusing the pipeline might be a viable option for the start of the project. This speeded up the development process of new projects with the ML practitioner being able to reuse components and steps of an ML pipeline. This might also be an indicator of automatically generated ML pipelines, which could include standard components but also use-case specific ML pipeline steps.
There were also quality requirements which could be derived from functional requirements mentioned in this section. One of these functional requirements was monitoring. P1 and P4 noted that metrics depended on what is time-tested and important for the use case. This highlighted the need for good and fitting metrics for every kind of problem and use case. Choosing which metrics are best suited was a very important step as described in [SBHV20] but also a source of errors because metrics were chosen which might seem like they could measure the success rate of the model to a problem but had flaws which made them unsuitable. The corresponding quality requirement could be described as the "observability" of the ML pipeline.

Challenges

Mentioned Challenge	Mentioned by
No standard procedure for setting up a pipeline	P1, P3
Data Quality	P2, P4
Runtime Configuration	P5
Model and Code Testing	P5
Deployment	P5

Table 4.3: Mentioned challenges of ML pipelines

In summary, there were five challenges regarding ML pipelines mentioned by the participants. Two more important, mentioned by more than one participant, challenges were mentioned while there were a further three challenges that were mentioned by only one participant each. Table 4.3 provides a summary of those challenges.

These challenges included the non-existence of a standard procedure for setting up an ML pipeline. This point was mentioned by P1 and P3 with P3 especially pointing out the challenge of creating an ML pipeline from a Jupyter notebook [KRP+16] which was used for proof-of-concept, data exploration and manual experimentation. According to P3, this was because the review of such Jupyter notebooks was difficult because of the size and complexity of these notebooks. A consequence was that other developers in the team could not get any value from the notebooks because of the mentioned reasons. P1 mentioned also the creation of an ML pipeline from scratch with functionalities like monitoring and retraining.

Data quality was also mentioned as a challenge by P2 and P4. This included problems like the handling of missing values and unlabeled data. A way for addressing this challenge could be data visualization. Tools for data visualization could be a great help for assessing data quality, identifying problems and choosing a suitable solution for these, participants mentioned.
The handling of configuration was also mentioned as a challenge by P5. Here, the challenge laid less in configuring data preprocessing steps and models but in configuring runtime execution-specific configurations. Especially mentioned here was the configuration of custom Docker containers which could be used as runtime containers for individual steps inside an ML pipeline but also for model deployment inside such containers.
Related to the testability of ML pipelines mentioned as one of the quality requirements, testing code was also one of the mentioned challenges. Ensuring the correctness of the pipeline steps was a major part in improving and ensuring the quality of the ML pipeline. P5 also added in that context the testing of the model to ensure that the model was working as expected and did not just achieve good performances depending on the data and metrics used. This was supported by the work by John et al. [JOB20] that added "verification and validation" of models to the challenges during development.
Lastly, P5 mentioned the challenge of model deployment. Deployment included a variety of possible configurations for the underlying runtime execution platform, the kind of deployment and use-case-specific configurations. Examples like batch processing as possible deployment configuration were mentioned. P1 also added non-automatic deployment because of customer-specific peculiarities which was a consequence of his current project. The non-automatic deployment in this project worked via a Git pull request which needed to be accepted by the company's developers and the custom framework which only offered functionalities that could be also provided by publicly available platforms but prevented vendor lock-in for the cost of bad documentation of the framework. This challenge was also mentioned by John et al. [JOB20] who stated that integration issues of models into software-intensive systems.
Challenges during the start of P2's project were the long duration until the data from the customer was acquired as well as understanding the data. During the development phase, challenges included the requirements engineering and the lack of knowledge of non-specialist colleagues about machine learning which is supported by John et al. [JOB20] who covered the shortage in domain experts.
P3's project had to handle the challenges of the heterogeneity of the data along with a multitude of data sources and the problem of data leakage.

Contextual Factors

In terms of what quality requirements and components of an ML pipelines depended on, the dataset was one of the most important factors according to the participants. All of them mentioned dependencies that could be categorized as data quality. It was clearly stated that the data preprocessing steps were most affected by varying data quality, with more preprocessing steps needed with lower data quality. The data preprocessing steps mentioned here were handling missing values and duplicates. Data visualization and a data quality profile, so how the features were distributed and what values were missing, were helpful for identifying problems. P3 also mentioned the complexity as well as the heterogeneity and homogeneity of the data having an influence on the ML pipeline. The ML practitioners were also in contact with domain experts to have an idea of what the features but also the data points and model results meant and how they could be interpreted, according to P3. Other factors the ML pipeline might depend on are data quantity, data format and data sources.

How do ML pipelines depend on ...	Contextual Factors
Dataset	Data quality Data complexity Domain knowledge Data quantity Data format Data sources Data security
Industry Domain	Technology Stack Infrastructure Issues Regulations
Team Expertise	Focus on Implementation Available Toolset Model Validation Expectations by Domain Experts Experience
Type of ML (e.g., supervised, unsupervised, reinforcement learning)	Model Deployment (API endpoints, batch jobs) Metrics
Type of problem (e.g., regression, classification, forecasting)	Length of Feedback Loop
Other factors	Execution Speed Inference Speed Model Availability Computational Power

Table 4.4: Mentioned contextual factors of ML pipelines

Different sources of data but also different capabilities of data handling by companies might come with different formats of data. Data collection by these companies might also be lacking continuity which might motivate the ML practitioner to acquire more additional data from different sources. The last dataset-related dependency on the ML pipeline was data security. The data might be confidential and stored in the customer's infrastructure with restricted access to the data for the ML practitioners. This confidentiality might come with an additional step: anonymization. Either the data acquired from the company might already be anonymized or the data needed to be anonymized by the team of ML practitioners before it could be used for developing and training ML models.

The industry domain was mentioned as a less important dependency on ML pipelines. P1 outlined more company-specific factors with special requirements in terms of the technology stack used by the company and enforced on the project as well as missing infrastructure for providing data, model training and deployment. P2 on the other hand highlighted the regulatory side with closely regulated companies and institutions like bank and insurance companies needing to enforce more regulations on the project. Here, important factors were versioning for traceability, good data quality in general and good documentation regarding the data origin. The other participants saw little to no influence by the industry domain and more importance in the use case.

Dependencies of the quality requirements and components of an ML pipeline on the team expertise were less clear. According to P1, the software engineers were more focused on the implementation side of the project compared to scientific personnel but also saw dependencies on the available toolset. So, tools were chosen for the development that the team was familiar with and the development was faster in the case where the tools were customed by the project's customer.
P2 added that during the experimentation and prototyping phase, every ML practitioner worked more by himself than in the team, with the team expertise playing a more important role when it came to the validating and testing of the model. For P3 and P4 the expectations of the domain experts in comparison to the results of the model played a role in the project, with the model often identifying patterns that the domain experts already knew about. P4 also added that the experience of the team members in data and feature engineering let the project progress faster. P5 on the other hand differentiated that the team expertise influenced the work as a team but less the ML pipeline of the project.

The dependence of ML pipelines on the type of machine learning used was very low, according to the participants. Only P1 distinguished between supervised learning and unsupervised learning and outlined the difference in model deployment. Models trained using supervised learning were mostly deployed using API endpoints, while unsupervised learned models used batch jobs that were available on-demand. P5 also added that he saw unsupervised learning methods more used for analysis and that it needed manual validation. Different metrics were used, according to P3, but other than that there was little to no influence.

The type of problem was also a dependence which was investigated during the interviews. Here, P1 made the distinction between a recommendation system and a clustering problem. The difference described related to the length of the feedback loop of these two problems. So, the feedback loop of the clustering problem did not need to be long compared to the feedback loop of the recommendation system. This was because of the reinforcement learning approach used for this system which generally has a short feedback loop, according to P1. While P4 and P5 added that different metrics for model validation should be used, all participants saw that there was little to no influence on the ML pipeline.

Other factors which the ML pipeline might depend on, included, among other things, execution speed. This meant not only the execution speed of the data preprocessing steps (P3) but also the execution speed of the model or the response time of the service in which the model was part of, according to P1 and P3. This also involved testing the model or service under a load of requests which are expected (P3). Another factor was the availability of the deployed model which also might depend on the placement in the business cycle, according to P1. This could be categorized as a deployment configuration such that the service was available during specific times of the day, week, month or year. Lastly, P4 mentioned the need for resources the ML pipeline could use, most notably computational power. Other than that, no further factors ML pipelines could depend on were added.

Table 4.4 provides a summary on what the participants answered in this section of the interview.

Pipeline Generation

At this point in the interview, the topic of the study was revealed to the participant. To make the topic more understandable for the participants, a small use case example was added. This example was the classification of flowers based on image data with the ML practitioner wanting the following functionalities to be a part of the ML pipeline: the versioning of data, code and model as well as monitoring of model performance metrics.

The participants were asked about what challenges could be addressed by using this approach. The answers included timesaving and reduction of mistakes (P4) as well as easier first initialization of an ML pipeline with extracted hyperparameters (P3). P1 also saw the generator as a bridge between experimentation/prototyping using e.g., Jupyter notebooks, and getting the code into a pipeline.

Prime candidates for the automatic generation of ML pipelines were a set of requested features for a possible tool for the approach.
Versioning was mentioned by P2 and P5. This included the versioning of the models in a model registry, the code and, especially mentioned by P5, the data. They also added model validation consisting of the generation of a report (P2), logging and a gateway that decided whether the trained model achieved sufficient metrics to be deployed (P5). P2 also requested the data pipeline be part of the deployable artifact which closely resembled a part of the production pipeline mentioned by P1 earlier.
According to P3, the model training should consist of the aforementioned grid search approach for algorithm selection and hyperparameter tuning as a simplified version of AutoML. Additionally, data visualization was also mentioned as part of an automatically generated ML pipeline. P3 proposed data visualization using tools like PowerBI, Tableau or Qlik by using the pipeline as a data source to import the relevant data into those tools.
P4 initially had a hard time suggesting prime candidates but after hinting at data visualization, visualizations like feature importance and correlation analysis visualizations were requested by him with the intent of excluding unimportant features and automatically generating new features based on important ones. Explainability of AI was also requested by P4.
P3 insisted on a modular structure of the pipeline as well as the files of the project. This included separating the hyperparameters from the code to make it easier to change those.
One functionality request by P1 consisted of the generator being able to extend an already existing ML pipeline. For that, the existing pipeline should be used as input and then extended by one functionality, like for example a data preprocessing step.

In the following question, the participants were asked whether they would trust automatically generated components of machine learning pipelines. All participants answered the same with them being able to trust the components after some time, getting comfortable with it and confident about the correctness of the generator's output.
Regarding the generator's input, the participants made no remarks on which input format they preferred. For the configuration input the following points were mentioned:

- Model Training: Here, the grid search approach was requested. Therefore, the configurations were related to this approach.
  - Problem type: Regression, classification, . . .
    - Classification: Threshold for which model output which class is chosen

  - Model algorithms: Model algorithms to be trained using the grid search approach
  - Parameter ranges for each model algorithm
  - Features
    - Selection of features
    - Type of model features
  - Metrics for scoring the model
- Deployment:
  - Deployment type: API endpoint, batch processing, ...
  - Kind of deployment: A/B-testing, canary deployment, ...
- Monitoring: Monitoring of the deployed model(s)
  - Selection of features
  - Selection of performance metrics

P1 also added the possibility of adding a predefined feature store which stored features from previous projects to be able to reuse those features in a new project.

4.2 ML Pipeline Overview

Using the collected data from the literature as well as the exploration interviews, a general view of the components and quality requirements of ML pipelines can be created, providing an overview of answers to the research questions 1 and 2. A visualization of an overview of ML pipelines which resulted from the answers by the participants is displayed in Figure 4.2.

There can be differentiated between how many configurations a component or functionality need with configuration-independent components where less or no configuration is needed while more configuration-dependent components rely heavily on those to work as intended by the ML practitioner or even exist.
Configuration-independent components and functionalities do not require any configuration by the ML practitioner besides the notion that these components should be part of the ML project. This also means that they are independent from the use case, and the ML practitioners decide whether they want the functionality for the project or whether they would even use it. These components thereby can be initially provided by the ML platform or set up as part of the project. Those components include, for example, model registry and experiment tracker.
More configuration-dependent components need information on how the functionality should look like. These components generally are essential parts of ML pipelines and that is why they often should be part of automatically generated ML pipelines. While there are configurations that can be used as default without further configurations for some of the more configuration-dependent components, others highly depend on configurations made by the ML practitioner to work correctly or at all. What kind of configurations can be considered important to the project or single steps can be viewed in Table 4.5.

Scope		Configurations
project-wide		Type of problem (regression, classification, ...) Type of ML (supervised, unsupervised, ... learning)
step-specific	data ingestion	Location of datasets Loading configurations (e.g., credentials) Data visualization
	data joining	Kind of data (time series, images, ...)
	data preprocessing	Used algorithms and parameters
	train-test split	Ratio (optional) Feature selection Target label (in case of supervised learning)
	feature engineering	Used algorithms and parameters
	model training	General AutoML? Grid search? Then model algorithms and hyperparameters
	model evaluation	Metrics (optional) Type of problem for default metrics
	model gateway	Required model performance
	deployment pipeline	Model packaging Deployment infrastructure and environment Deployment pattern

Table 4.5: Project-wide and step-specific configurations for automatically generating ML pipelines

The ML pipeline generally consists of three pipelines. These pipelines are the data pipeline, the model pipeline and the deployment pipeline.

The data pipeline is debated to be not part of ML pipelines as this workflow often lies outside the machine learning part and can also stand by itself in non-ML-related projects and tasks. Still, as seen by the results of the exploration interviews, data preparation and data preprocessing are an essential part of machine learning and the data pipeline should therefore be part of an ML pipeline. It is supposed to load data from different data sources, join them in a meaningful way and bring it in a fitting format before preparing it for the machine learning process. The preprocessing part of the data pipeline has a very large intersection with the feature engineering part. Generally, the data pipeline tries to improve the quality of the data, while feature engineering steps prepare the data for a machine learning model. Therefore, the data preprocessing steps of the data pipeline can contain steps like handling missing values and outliers. Data visualization should also be part of the data pipeline to enable the ML practitioner to inspect the distribution and correlation of the features and recognize patterns in the data. The data pipeline contains the steps data ingestion, data join and all of the data preprocessing steps as well as data versioning.
While the data ingestion exhibits a high configuration dependence because the datasets to be ingested need to be defined by the ML practitioner, the configuration dependence of the data joining step is not so clear. Although a general data joining could be applied to some datasets, this does not apply for complex use cases and low-quality datasets. Except for configurations regarding the format of the dataset (e.g., time-series), there are no obvious configurations for this step, leaving the work to

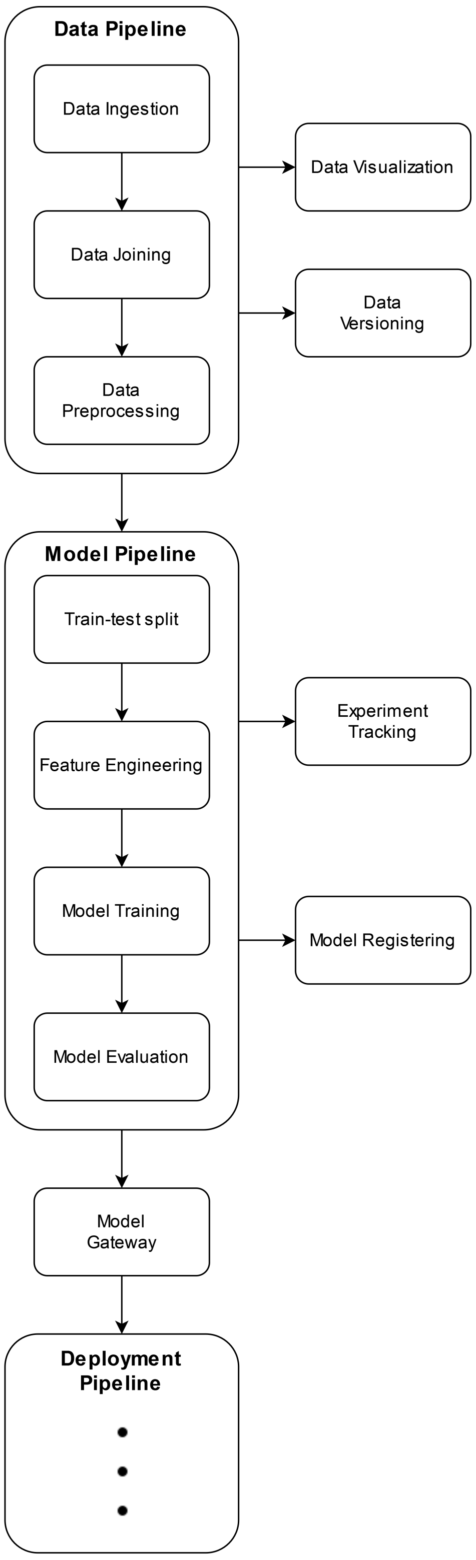

Figure 4.2: Overview of ML Pipelines

ML practitioners to define the joining of the dataset manually or to some automized approach that analyses the data before joining the datasets in a meaningful way. The data preprocessing steps on the other hand are one of the most configuration-dependent steps of ML pipelines. The steps to improve data quality are optional components that only are generated if stated by the configuration using data preprocessing algorithms and according parameters. While, depending on the use case, the data visualization needs to be configured to function in a meaningful way, simple visualizations of the data are independent from the use case and thereby a more medium configuration-dependent functionality. The configuration dependence of the data versioning components on the other hand relies heavily on the implementation by the developer of such an approach. This could be for example defined as default for all datasets, even all data artifacts that result from the execution of a pipeline step or those between single pipelines. Which configurations the ML practitioner that uses the generator could make thereby also relies on the implementation with the possibility to define versioning for the raw dataset or no possibility to add the functionality altogether.

Following the data pipeline is the model pipeline which covers the preparation for, model training itself and model evaluation. As the participants said during the exploration interviews, one important data preparation step is the train-test split which splits the dataset into multiple datasets with a defined ratio. The most common approach is to split the data into 2 datasets, one training dataset for training the model and one testing dataset for testing how well the model performs with unseen data. Another approach adds a validation dataset which is used for stopping the training process to avoid overfitting. The train-test split is a necessary step in the ML pipelines making it rather configuration-independent. The ML practitioner could be given the possibility to configure the train-test ratio as well as the inclusion of a validation dataset. Additionally, the feature selection could be implemented in this step as well, allowing more possibility for configuration by the ML practitioner.
The feature engineering is supposed to prepare the data and make it easier for the model to learn patterns in the data. This could include feature scaling using min-max normalization or z-score normalization. One important topic to keep in mind is to fit the scaling algorithm to the training dataset and transform the test and validation dataset with the same scaler. This prevents data leakage because otherwise, the normalization on the training dataset includes information about the test data. This measure is described in [SBD+21] and [BBG+23].

The model training utilizes AutoML to find the best combination of model algorithm and hyperparameters. This can be the grid search approach mentioned in the exploration interviews but also a more general AutoML approach provided by libraries and frameworks can be a solution. The model with the best metrics during training gets validated in the model evaluation/validation step. Here, the model performs predictions on the previously unseen test data. Metrics collected during the step not only include performance metrics like accuracy (depending on the kind of ML problem) but also metrics like feature importance. If the resulting metrics are better than a defined threshold, the model is saved in a model registry and deployed using the deployment pipeline. Whether this is the case, is checked using a gateway.

In a traditional approach, a model algorithm is chosen or even implemented and then trained with defined hyperparameters which is configuration-dependent. The approach resulting from the exploration interviews using grid search to find the best combination of model algorithm and hyperparameters needs the ML practitioner to configure a series of model algorithms as well as suitable possible hyperparameters. Because of that, this approach is also quite configuration-dependent. Another, less configuration-dependent, approach is using a more general AutoML

method which automatically finds the best combination of model algorithm and hyperparameters. Depending on the implementation, the ML practitioner could not need to configure the model training due to a complete reliance on AutoML.
Even though there are default metrics for the most common kind of problems, depending on the use case other metrics are needed to evaluate the model, also during model training where the grid search approach or AutoML is used. Because these metrics are optional, the model evaluation is rather configuration-independent.

In between the model and the deployment pipeline is the model gateway, a single configuration-dependent component that decides whether the newly trained model is deployed. The conditions on how this decision is made need to be configured by the ML practitioner. A more complex approach is the possibility to deploy the newly trained model in the case that it achieves a better performance than the currently deployed model.

The deployment pipeline is one of the most complex parts of the ML pipeline. Many different configurations need to be made and platforms to be considered. Possible configurations include the pattern used for serving the model like Model-as-Service and Model-as-Dependency [VKB+23], the infrastructure the model needs to be deployed on as well as less machine learning related configurations like availability and scaling.

4.3 Generation Approach

Using the results from the exploration interviews, a prototypical approach was developed to demonstrate the overarching goal of the study to automatically generate context-specific ML pipelines. The rest of the paper a differentiation is made between the "Generator" and the "Prototype". The program developed during the study is identified as the prototype while the generator means a program that solves the same problem as the prototype tries to solve but includes more functionalities and might work in a different way.

4.3.1 Kubeflow

Initially, Kubeflow [The18a] was the platform of choice. Kubeflow is an end-to-end platform for ML projects on Kubernetes [The14]. It supports a variety of features including Jupyter Notebooks, Model Serving and Pipelines. The Kubeflow Pipeline (KFP) module offers "a platform for building and deploying portable, scalable machine learning (ML) workflows based on Docker containers" [Mer14].

In KFP, an ML workflow is a pipeline consisting of multiple nodes which can run in exclusive Docker containers. Using the KFP SDK, a development kit for Kubeflow pipelines, the developer can easily create pipelines using a Python decorator. Nodes can also be easily created by supplying a Python function that is executed inside the Docker container upon the execution of the node during a pipeline run.

One of the initial reasons for using Kubeflow as the prototype platform was the product's completeness. Many functionalities mentioned in the literature regarding data science are provided by Kubeflow and few other tools were needed to be added. Another reason was the flexibility of the deployment of Kubeflow. Kubeflow can be deployed using solely a Kubernetes cluster which can run locally as well as in the cloud. These deployment options give the developer more flexibility and enhance scalability, which is one important factor in data processing as well as model training.

The reasons for abandoning Kubeflow as the platform for the prototypical approach were insufficient documentation as well as problems with adding more tools to a project. Another reason was the poor extensibility regarding other third-party tools as well as the insufficient documentation and management of artifacts between pipeline nodes.

4.3.2 Kedro and MLflow

During the exploration interviews, the tool combination of Kedro and MLflow was mentioned.

Kedro [The23a] is a pipeline orchestration framework using Python specialized for machine learning with an emphasis on "maintainable and modular data science code". Kedro creates a new project using a folder structure, separating configuration files, data files and code. The last is organized in a folder structure with every major pipeline having its own folder, including a `nodes.py` file that includes the Python functions for the nodes and a `pipeline.py` file that covers the declaration and definition of the pipeline. A data catalog allows for declaring datasets as well as configuring the loading process and other configurations like data versioning of datasets. The definition of pipelines consists of a series of nodes along with their respective code, name, input and output. The code of a node is simply a Python function. Using Kedro's pipeline visualization plugin Kedro-viz [WbK+21], the Kedro pipeline can be visualized via the browser in a clear interface.

MLflow [The18b] is a platform "to manage the ML lifecycle". It provides features like experiment tracking, reproducibility, a model registry and model deployment. Providing a modern graphical user interface that is accessible via the browser, the developer can view tracked experiments, details and metadata of each experiment as well as models and their deployment stage and is able to change this by click. Using the Kedro-MLflow plugin [The23b], the developer is able to use Kedro and MLflow in unison and no further configurations are needed to be made.

One of the main reasons for choosing the tool combination of Kedro and MLflow is that it already provides a variety of configuration-independent features, like pipeline visualization and experiment tracking. Another reason is the simplicity of setting up a project with the developer just needing to execute a handful of shell commands. With a range of plugins for Kedro, it is possible to deploy the Kedro pipeline on other platforms and infrastructure. The detailed documentation of the Kedro and MLflow APIs allows the developer to gain more insight into how the platform and tools work and thereby avoid possible errors during the development of the ML project. Because both frameworks only depend on configuration files and the filesystem, the generation process of a pipeline consisted of the generation of Python files.

4.3.3 Functionalities

The more configuration-independent components of the ML pipeline were provided by Kedro and MLflow. Besides Kedro as the pipeline orchestration tool providing the visualization of the pipeline, it also provides data versioning in a limited way. Especially the versioning of the local raw datasets needs to be set up manually besides adding the versioning to Kedro's data catalog. Because of the lack of value to the purpose of the study, data versioning is not supported by the prototype. For this prototypical approach, MLflow covers the functionalities of tracking experiments as well as the model registry. This leaves data visualization, for which the tool Sweetviz [Ber22] was used. Sweetviz is an open-source Python library that generates visualizations using self-contained HTML applications containing information about a dataset. This information includes the number of data points, distribution of values and number of missing values as well as statistical information about each feature of the dataset. Additionally, plots of the data feature regarding its values as well as tables about the correlations between features are generated. Besides those three tools, the Python library scikit-learn [PVG+11] is used, especially for splitting the data in training and test dataset as well as model training. Therefore, the model produced by the ML pipeline is of scikit-learn's format.

The generated ML pipeline contains the following steps:

- Data Pipeline
  - Join Datasets
  - Cut Outliers (optional)
- Model Pipeline
  - Train Test Split
  - Scaling (optional)
    - Scale X Train
    - Scale X Test
  - Model Training
  - Model Evaluation

Because of the immense complexity of it, the deployment pipeline was excluded from the functionalities of the prototype. For the prototype to function correctly, there was no installation of Python packages necessary. During the development of the prototype, Python version 3.10.2 was used but further tests revealed that other sub-versions of 3.10.x could work similarly. For the resulting ML pipeline to function correctly, a series of Python packages needed to be installed. The repository for this study includes the file `requirements.txt` which additionally covers these packages as well as their exact versions. The repository contains all the necessary files of the prototype inside the folder `prototype`. Besides Python files like `main.py`, the entry point of the program, the prototype contains `README.md` covering the usage of the prototype and `requirements.txt` listing all necessary Python package versions needed for the ML pipeline to function correctly inside the Kedro project. Furthermore, the folder `generator` contains internal code and the "data" folder contains two example datasets in `CSV` format as well as documentation on the input configuration inside `config_template.md` and an example configuration file in `template.yml`. This example configuration file contains the configuration for an example classification project using the two mentioned example datasets alongside the target label as well as a collection of model algorithms and hyperparameters for model training and further minor configurations.

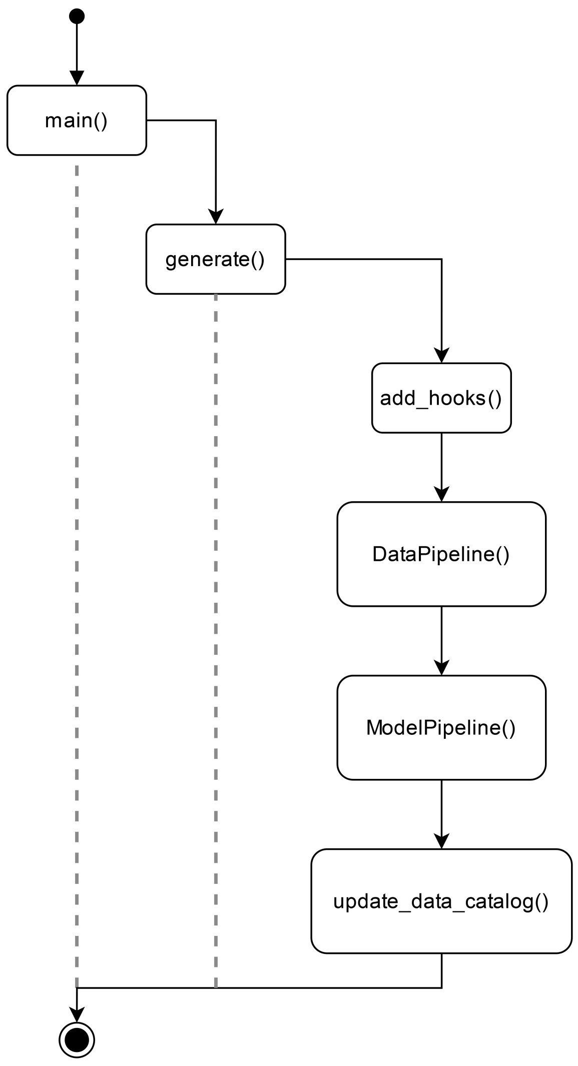

Figure 4.3: Overall program flow

4.3.4 Architecture

Because Kedro projects are based on Python files, the generation process consists of creating and writing Python code into those files. These include a folder for each pipeline as well as Python functions inside `nodes.py` and the pipeline creation inside `pipeline.py` as well as the declaration of the dataset inside the data catalog and the `hooks.py` and `settings.py` for Kedro's hook system. The entry point for the program is the file `main.py` which is executed using the command `python`. Here, the program arguments are parsed and the configuration file is read. Following this, the generator is started using the `generate` function. In `generator.py`, the configuration from the configuration file gets enriched and all pipeline generation is coordinated. Firstly, the layer of the datasets declared in the configuration file is set to `raw` which is important for Kedro-viz for visualizing the resulting ML pipeline in a more structured fashion. The program flow is visualized in Figure 4.3.

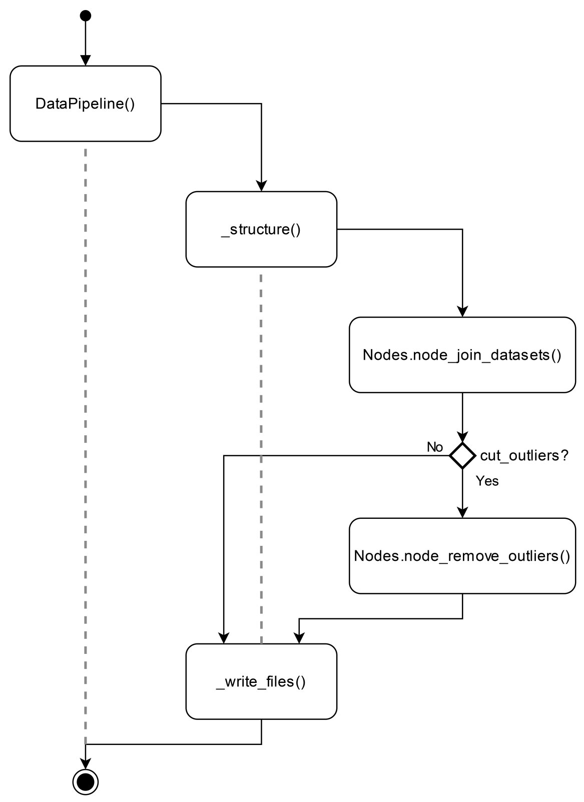

Figure 4.4: Program flow of the creation of the data pipeline

Kedro Hooks

After that, the Kedro hooks are written into the files. For this, the file `hook_writer.py` declares functions for writing to the projects `hooks.py` and `settings.py` as well as for generating the code for the data visualization. The last one is done using the function `add_hook_data_visualizer`, which returns code for the hook in the form of a Python string. This function is executed when the visualization is found in the generator internal configuration and returned code is given to the file writing function. With this structuring of functions, it is simple to add new functionalities as hooks by declaring code inside a function and then declaring when the hook is added to the resulting project. The file also contains the functionality of logging the artifact after model training. This functionality was replaced by MLflow's `autolog` functionality.

Pipelines

After the hooks, the pipelines are created as visualized in Figure 4.4 for the data pipeline and in Figure 4.5 for the model pipeline. For that, every pipeline has a dedicated class that handles the structure of the data pipeline as well as the writing into the files while holding information regarding intermediate datasets between nodes. A class `Nodes` is part of every pipeline class to encapsulate the code and information needed for adding new nodes to the pipeline in an easy manner. This class holds several functions that return the code for the pipeline nodes as well as information on that node to register it as part of the Kedro pipeline. Besides the code which is written in the pipeline's

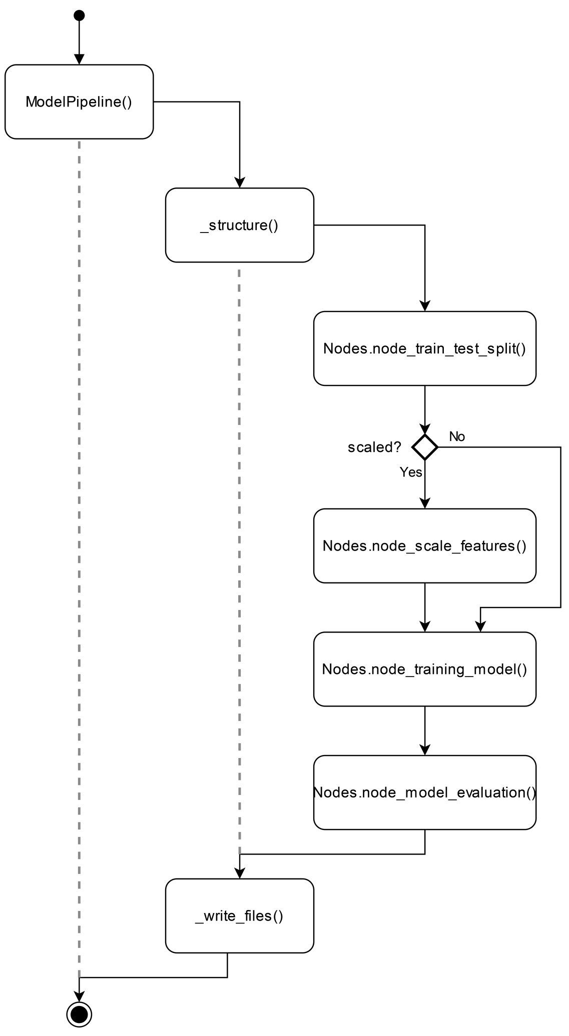
ModelPipeline()
_structure()
Nodes.node_train_test_split()
scaled?
No
Yes
Nodes.node_scale_features()
Nodes.node_training_model()
Nodes.node_model_evaluation()
_write_files()

Figure 4.5: Program flow of the creation of the model pipeline

nodes.py file, the functions return the name of the generated function, the name of the resulting node as well as the names of the input and output datasets as a list each.
During the object creation of the pipeline's class, the function _structure is called which executes the functions for the necessary steps and adds the information about the node and the output dataset to a list each which is then used by the function _write_files which writes the code for the nodes in the corresponding file and the information on all nodes in the file pipeline.py.

While the nodes.py can be generated by writing the nodes' code one after another into the file, generating the file pipeline.py is more complex. So, the code for the list of nodes is created by filling in the necessary information for the Kedro pipeline in a template. This code containing a list of node information is then inserted in the code for Kedro's function create_pipeline which generates the pipeline. This pipeline can then be automatically found by the Kedro project without further registration.
Because Python lists of names can be easier represented as a string, all the input and output datasets are represented as lists, even if the list only holds one single dataset. This is because a Python string object is only printed without quotation marks, which would be a syntax error in the generated

Listing 4.1 Generation of the nodes.py for the model pipeline

```
    def _write_files(self):
        nodes_list = self.additional_nodes
        # write nodes.py
        with open(self.model_pipeline_dir.joinpath('nodes.py'), 'w') as node_file:
            for node in nodes_list:
                node_file.write(node['code'])
        # write pipeline.py
        write_pipeline_file(self.model_pipeline_dir, nodes_list)
        # write pipeline specific params
        write_parameters_file(self.root_dir, self.name, self.params)
```

Listing 4.2 Code snippet of the generation of the pipeline.py file for any pipeline

```
def write_pipeline_file(pipeline_dir: Path, nodes_list: list) -> None:
    func_str = ', '.join([n['func'] for n in nodes_list])
    nodes_code = ''
    with open(pipeline_dir.joinpath('pipeline.py'), 'w') as file:
        for node in nodes_list:
            nodes_code += f'''
            node(
                func={node['func']},
                inputs={str(node['inputs'])},
                outputs={str(node['outputs'])},
                name='{node['name']}',
            ),'''
        # write into file using nodes_code
```

Python file. This is not the case for lists that include quotation marks for each string object inside the list. Furthermore, only the return of the node's function needs to be encapsulated inside a Python list. Other than that, there is no further difference.
The generation of the `nodes.py` file for the model pipeline is shown in Listing 4.1. The generation of the corresponding file for the data pipeline works in a similar way. How the composition of Python code for the `pipeline.py` file is constructed, is displayed in Listing 4.2.

Data Pipeline

For the data pipeline, the nodes include the node for joining all datasets and the optional node for removing outliers from the data. The joining of all datasets requires the names of all datasets defined in the configuration file. The joining itself is done by using the function `reduce` of the Python package `functools` which applies the `merge` function of the Python package Pandas [McK10] to the list of datasets from left to right, as displayed in Listing 4.3.

Listing 4.3 Joining of a list of datasets

```
def join_datasets(*datasets) -> pd.DataFrame:
    return [reduce(lambda left, right: pd.merge(left, right), datasets)]
```

Listing 4.4 Code for generating outlier removal using the 'percentile' option

```
df = df[
    (df['{feature_name}'] < df['{feature_name}'].quantile(0.95)) &
    (df['{feature_name}'] > df['{feature_name}'].quantile(0.05))
    ]
```

This is a general joining of datasets with the `merge` function handling all of the necessary logic. This approach might not work completely or not as wished by the ML practitioner in more complex scenarios.

For removing outliers, the code generated highly depends on which outlier detection algorithm was chosen for what features. For every feature for which outlier removal is defined in the configuration file, there is a line of code removing these outliers from the dataset.
For the outlier removal using `percentile` which removes the 5^{th} and 95^{th} percentile of this feature's data points, the function `quantile` from pandas' `DataFrame` class is used. With this, two boundaries can be defined where only the data points within the boundaries are selected. How the code generation and the resulting code for the option `percentile` looks like, is displayed in Listing 4.4.

For the outlier removal using `iqr`, the function `quantile` is also used to calculate the interquartile range. In the second line of code, only the data points are selected which are within the boundaries of 1.5 times the calculated interquartile range above the 1^{st} and below the 3^{rd} quartile. For the outlier removal using `zscore`, the Z-score [Abd07] is calculated for each data point. This involves calculating the difference between the data point and the mean divided by the standard deviation. The resulting Z-score needs to be below or equal the value of 3 for the data point to be included in the updated dataset with data points with a Z-score above the value of 3 to be excluded as an outlier. How the resulting data pipeline looks like in Kedro is visualized in Figure A.1.

Model Pipeline

The model pipeline is a highly parameterized part of the generated ML pipeline. Many nodes contain the same code independent of the configuration provided by the ML practitioner. The provided configurations of these steps are included as parameters, extracted in separate files.
In Kedro, parameters are organized in dedicated YAML files. The defined parameters can then be used as input of a node during the pipeline's declaration with the `params:` prefix added. Inside the node's function, the parameters can then be used as Python dictionaries. Instead of using the default `conf/base/parameters.yml` file for managing all parameters across the whole project, this prototype creates an additional folder called `parameters` which holds a parameter file for each pipeline where necessary.

The train-test split is the first step of the model pipeline, splitting the dataset which was preprocessed by the data pipeline into two datasets, each containing two parts. These include a training dataset, one part without the target label `X_train` and one only the target label `y_train`, and a similar test dataset with `X_test` and `y_test`, respectively. The parameters regarding this step are called `split_options` and contain the ratio of the size of the training dataset to the test dataset as well as the name of the target label and the name of the features selected for model training. The train-test-split step first encodes string-type features using scikit-Learn's `OrdinalEncoder`, giving each different string value an integer value. This newly generated feature is then used to replace the old feature containing the string values.
Following that, the target label is retrieved from the parameters and the corresponding feature is split from the rest of the dataset. Then, the features for the model training are selected based on the parameters retrieved from the configurations file. The train-test split itself is done using scikit-Learn's `train_test_split` function. Additional to the ratio of the two dataset sizes, in case of a classification problem, the `train_test_split` function's `stratify` parameter is set to the variable's name for the data containing the target label, `y`. This way, the data is split in a stratified fashion, using the data containing the target label as class labels. Imbalanced data thereby has a similar ratio of the class label in both training and test datasets, preserving the ratio from the original dataset in both even after the split.

After the train-test split, there can be feature scaling steps. This depends on whether the ML practitioner added the scaling of at least one feature in the configuration file. If that is the case, two steps are added between the train-test split and the model training. The first step is called `Scale_X_train` and scales the corresponding features in the training set. This is done using scikit-Learn's `MinMaxScaler` which fits to and then transforms the training data. The scaler object is then part of the output of the node such that it can be used to transform the test dataset in the step `Scale_X_test`. The feature scaling is done in two separate steps with the same scaler to avoid data leakage [BBG+23][SBD+21]. If the scaler object would be fit to the whole dataset before the train-test split, there might be information about the test dataset in the training dataset.

The model training step is another highly parameterized ML pipeline component. For implementing the grid search approach mentioned by the participants during the exploration interviews, scikit-Learn's `GridSearchCV` was used. This class takes a model estimator and a grid of parameters and find the best parameters for this model. For testing out different model algorithms too, a few changes needed to be made. The estimator was replaced by a scikit-Learn `Pipeline`, containing a single step called `clf` which creates a `DummyEstimator` object. This way, the model algorithm can be part of the parameter grid of the `GridSearchCV`. The parameter grid consists of a list of model algorithms with their corresponding parameter values. Every list item contains an initialized object of the model algorithm with the key `clf` and a list of values for the parameters which the model should be trained with, having the key `clf [parameter_name]` with `[parameter_name]` being the name of the parameter according to scikit-learn's documentation of the model algorithm. After the execution of the grid search with the training dataset, the best-performing model is returned. For tracking the model training, MLflow's `autolog` function for scikit-learn was executed at the beginning of the ML pipeline step. This automatically logs parameters of and model metrics during the training process.

The last step of the model pipeline is evaluating the model which shows the best performance during the training process. Besides the model from the model training step, this step needs the test dataset along with the corresponding labels and metrics which were defined in the configuration file by the

ML practitioner. For this step, MLflow's `evaluate` function is used. Using this approach, common metrics are tracked depending on the machine learning problem by adding what type of model, regressor or classifier, the model to be tested is. At the end, the additional metrics which were defined in the configuration file by the ML practitioner are logged.

How the resulting model pipeline including and excluding the nodes representing parameters looks like in Kedro is visualized in Figure A.3 and Figure A.2, respectively.

Data Catalog

At the end of the generation process, the data catalog is generated. For this, the file `conf/base/catalog.yml` in the Kedro project is filled with objects describing information on datasets. These include datasets added by the ML practitioner in the configuration file but also intermediate datasets. The dataset `Joined dataset` stores the resulting dataset after the step which joins all of the datasets from the configuration file with the layer `raw`. This dataset is stored as a `CSV` file and gets added the layer information `intermediate` such that the dataset is visualized below the `raw` datasets. In the next layer (`primary`), there is the dataset resulting from the removing outlier step if this step is part of the generated ML pipeline. Unlike the dataset before, this dataset is only of type `memory` which is an identifier for this dataset being of Kedro's `MemoryDataSet` which loads and stores data from/to an in-memory Python object. Therefore, the dataset is not stored permanently and the steps to create this dataset need to be run again if the ML practitioner wants to do so. The datasets created from the train-test split step, namely `X_train`, `y_train`, `X_test` and `y_test`, are part of the layer `feature` and also are `MemoryDataSet`. As are the resulting datasets from the optional feature scaling steps, the scaler object and the scaled datasets `X_train_scaled` and `X_test_scaled` which are part of the `model_input` layer. The final dataset listed in the data catalog is the model, stored as a file in `pickle` format within the `model` layer.

Configuration File

Using the configuration file, the ML practitioner is able to provide the necessary information to the generator for generating the ML pipeline. This YAML file contains basic information about the project, the data and many configuration-dependent components of an ML pipeline. A short documentation on the exact syntax of the configuration file can be found in the project's repository. All components are mandatory unless specified otherwise.

The datasets are declared in a similar way to Kedro's data catalog, as specified in Listing 4.5. The `datasets`-list contains dictionaries containing the name, one of the supported types of the dataset as well as the file path and other arguments for loading, saving the data as well as credentials. The name is a string for identifying the dataset inside the Kedro project. The type of the dataset describes the format of the file containing the dataset. For this prototype, the file formats `csv`, `json`, `xls` and `parquet` are supported. The file path indicates where the data file can be found and loaded from. This can be a local file path but also an URL pointing to a remote file.

Another project configuration is the `problem` which is a string describing the problem the project tries to solve. This prototype accepts one of `regression` and `classification` as the configuration for this key. This is specified in Listing 4.6.

Listing 4.5 Specification of the dataset configuration

```
datasets:
  - name: <name>
    type: 'csv' | 'json' | 'xls' | 'parquet'
    filepath: <path to file/remote>
    # [Optional:]
    load_args:
      <Arguments according to kedro's specification>
    save_args:
      <Arguments according to kedro's specification>
    credentials:
      <Credentials needed for accessing the dataset>
  - name: ...
    ...
```

Listing 4.6 Specification of the problem configuration

```
problem: 'regression' | 'classification'  # Type of problem
```

This prototypical approach generates ML pipelines for projects trying to solve the problem using supervised learning methods. For this method, one or multiple target labels should be defined. This is done by the `target_label` key in the configuration file as displayed in Listing 4.7. The value of the key should be the same as the name of the label in the data.

The `visualization`-key is an optional configuration for adding data visualization using Sweetviz, as specified in Listing 4.8. The default value is `false`, so the resulting ML pipeline does not initially support data visualization. With the value `true` Sweetviz is added as a Kedro hook to generate HTML files during an ML pipeline run.

The train-test split used for splitting the dataset in a training and testing dataset is described in the configuration file with the key `train-test-split` as displayed in Listing 4.9. This is a dictionary containing optional information like the size of the training dataset as a ratio to the complete dataset and the same configuration for the test dataset.

One optional configuration is the `features`-list. This describes the feature selection and engineering steps which should be part of the ML pipeline. For that, every feature which should be used for model training needs to be listed as well as the preprocessing steps which should be performed on this feature like displayed in Listing 4.10. If there are no features with preprocessing configured, no

Listing 4.7 Specification of the target label configuration

```
target_label: <name of the feature which should be used as target variable>
```

Listing 4.8 Specification of the configuration for data visualization using Sweetviz

```
visualization: true  # [Optional:] data visualization using sweetviz (html report gets
generated in data/08_reporting)
```

Listing 4.9 Specification for the train-test split configuration

```
train-test-split:
  train-ratio: <Value between 0 and 1>  # [Optional: ] Ratio of training data from the
original data
  test-ratio: <Value between 0 and 1>  # [Optional: ] Ratio of test data from the original
data
```

preprocessing steps are added to the resulting ML pipeline. The data preprocessing algorithms available are the removal of outliers and min-max normalization. For outlier removal, one of the following algorithms can be chosen:

- `percentile`: The 5^{th} and the 95^{th} percentile of the data is removed.
- `iqr`: Data points outside of the 1.5-times the interquartile range below and above the 1^{st} and 3^{rd} quartile are removed.
- `zscore`: The Z-score for each data point is calculated and data points with a Z-score above 3 are removed.

For min-max normalization the key `scaled` is used which can be set to `true` while the default value is `false`. If the `features`-section is not part of the configuration file, all the available features are used for model training and no preprocessing steps are generated.

Listing 4.10 Specification of the configuration for feature selection and engineering

```
# [Optional:] list of feature names which are used for training all models
# If none, all available features are used
features:
    - <name of feature>:
# [Optional:] whether this feature shall be scaled using scikit-learn's MinMaxScaler
        scaled: true
# [Optional:] whether outliers will be dropped from the dataframe.
# The value determines the algorithm chosen for detecting outliers.
# 'percentile' removes the 5th and 95th percentile of the data,
# 'iqr' removes data points which are outside of 1.5-times the interquartile range below/above
 the 1st and 3rd quartile,
# 'zscore' removes data points with a z-score above 3
        cut_outliers: percentile | iqr | zscore
    - ...
```

Model training is configured using the `training`-keyword like specified in Listing 4.11. Because grid search is used to find the best combination of model algorithm and hyperparameters, model training needs a series of model algorithms and hyperparameters for each. This list inside the `training`-dictionary has the key `model`. A list item contains the name of the algorithm and parameters which fit the algorithm. All the algorithms that can be chosen are implemented by scikit-learn. The following algorithms are available:

- `tree.DecisionTreeClassifier`
- `tree.DecisionTreeRegressor`
- `linear_model.LinearRegression`
- `linear_model.LogisticRegression`
- `linear_model.PassiveAggressiveClassifier`
- `linear_model.RidgeClassifier`
- `linear_model.SGDClassifier`
- `linear_model.Ridge`

Generally, the identifiers for model algorithms are similar to the identifiers used by scikit-learn. The first part before the dot identifies the scikit-learn's module while the second part identifies the model algorithm. The parameters for each model algorithm is supposed to have the same name as described in scikit- learn's documentation. Each of the parameters can have a list of values that will be tested during the model training step. These values should be valid for the associated parameter according to scikit- learn's documentation. If a parameter is not listed, the default value is used according to scikit-learn's documentation.

Listing 4.11 Specification for configuration regarding model training

```
training:
  models:
    - algorithm: <model algorithm>  # key for sklearn (Example: 'linear_model.LinearRegression
')
      parameters:
        <name of parameter>:  # List of values or min/max
    - ...
```

For model evaluation, the `metrics`-list can be defined which contains the metrics the model should be evaluated with additionally to common metrics depending on the problem of the project. This is specified in Listing 4.12. Available metrics and the corresponding keywords can be found using the scikit-learn's documentation on metrics and scoring (`https://scikit-learn.org/stable/modules/model_evaluation.html`).

Listing 4.12 Specification for the metrics configuration

```
# String id or list of those of the corresponding sklearn metric
metrics: <string id / list of string ids>
```

4.4 Evaluation

This section covers the second round of expert interviews for evaluating the prototypical approach. Those interviews lasted approximately between 55 and 70 minutes.

Firstly, they were asked about their experience with Kedro and MLflow, preferably in years of professional experience. While P2 said, he had approximately one year of experience with Kedro and P3 used it “a while ago”, the others had no experience with Kedro. The participants were a bit more familiar with MLflow. P3 stated one year of professional experience while P2 “used it before”. P1 had between 1 and 2 years of professional experience while the participants P4 and P5 stating no experience with MLflow. After this, the interviewer sent the participants the directory `prototype` in ZIP format and noted that the two example datasets in the `data` folder need to be moved to the example project’s folder for raw data. Additionally, the interviewer explained where to find the documentation on how to use the prototype and how to configure the prototype’s input in the configuration file.
During the play-around phase, the participants used the example configuration file to try out the prototype. Generally, the participants looked inside the example configuration file shortly, before executing the prototype.
Following the play-around phase, the interviewer tasked the participant with the exercise to further expose him to the functionalities of the prototype and possible options of configuration for the configuration file. In short, the interviewees needed to extend the example configuration file by adding the `features` section and note the removal of outliers and min-max normalization to at least one feature before executing the prototype and later the ML pipeline inside the Kedro project.

Identified Weaknesses

Identified Weakness	Problematic for
Complexity of Directory Paths	All participants
Feature Selection	All participants
Data Preprocessing & Feature Engineering	All participants
Target Label is part of feature list	All participants
YAML syntax (indentation)	P1, P2
YAML syntax (keywords)	P4

Table 4.6: Identified weaknesses and problems of the interviewees

There were a few problems and mentioned weaknesses of the approach during the second round of the interviews, summarized in Table 4.6. The execution of the prototype was errorless except for the execution by P3 where a problem with a Python package was stated. The reason for this problem was not found, although the participant was able to execute the prototype after creating a new Python virtual environment [VD09]. Another problem with executing the prototype, especially of P2 and P5, was that the participants seemed to struggle with directory paths to the configuration file and the root directory of the Kedro project which were needed to be added to execute the prototype. For P4, the frequency of switching between the prototype and the example project in the terminal was also cumbersome. To solve the exercise, all participants tried to apply the data preprocessing and feature engineering steps to all features at once. The interviewer noted that this is not possible and referred to the documentation. After this, the participants understood that they needed to list all features for model training if outlier removal and/or feature scaling should be added to at least one feature. The interviewees indicated this as tedious, especially for data with a large number of features. The participants proposed listing only features which should be preprocessed as well as an exclusion of unwanted features for adding data preprocessing and feature engineering steps easier. For this, they looked inside the dataset files to copy the available features and paste them into the configuration file. This meant that the interviewees also copied the target label and pasted it into the list of features, which would have resulted in an error upon execution of the prototype. The target label should be separately defined in the configuration file instead of being part of the feature list. The interviewer noted this mistake during the adoption of the features to the YAML's syntax. The YAML syntax of the `features` section was another common mistake by the participants. P1 and P2 added the `cut_outliers` and `scaled` keys one indentation less than needed. The list item of this section is another dictionary though with the feature's name as key. To simplify this, the indentation could be removed by adding a new key, e.g. `name`, to the list item such that all the configurations in this section have the same indentation. The current syntax is displayed in Listing 4.13 while the proposed syntax is specified in Listing 4.14.

Listing 4.13 The configuration syntax for adding feature preprocessing to the ML pipeline

```
features:
  - feature_name:
      cut_outliers:iqr
      scaled: true
```

Listing 4.14 The proposed configuration syntax for adding feature preprocessing to the ML pipeline

```
features:
  - name: feature_name
    cut_outliers:iqr
    scaled: true
```

The last problem during the interview of P4 was an unidentified syntax error of the key `cut_outliers` in the configuration file. This caused the prototype to ignore the outlier removal of the feature. Because this was the only feature which outlier removal should be applied, the generated ML pipeline lacked the corresponding steps. Critically, the prototype did not throw any output related to the typo, thereby indicating an errorless execution, which was not the case. This could be addressed by automatically checking the syntax of the configuration file by the prototype before generating the ML pipeline.

Questions

There were a number of questions by the participants during the play-around phase and the exercise. These may indicate which steps and components the participants were most curious about. P1 asked about how the joining of datasets works, especially in case of heterogeneous data or different timeframes of time series data. The participant also asked about whether multiclass classification is supported and whether the resulting pipeline supported big data and distributed computing. Default values of the data preprocessing algorithms were also unclear, according to P1. P2 asked whether the absence of output by the prototype is a good or a bad sign regarding success of the prototype's execution. P3 and P4 asked about how the prototype can be used, indicating that the documentation was insufficient. This could be improved by describing the prototype's purpose of automatically generating ML pipelines as well as how the user is able to achieve this.

Advantages

Advantages	Mentioned by
Prototype easy to use	P3
Fast creation of first ML pipeline	P5
Easy to learn	Observed among all participants
Good structure of the ML pipeline	P2
Data visualization helpful	P1
MLflow's report helpful	P3

Table 4.7: Most mentioned advantages

There were many advantages of the approach noted by the participants, summarized in Table 4.7. P1 mentioned that the configuration file is easily created and the approach is an easy entry for creating ML pipelines. P5 also noted that the approach is "straight-forward" for ML pipeline creation and good for a project's start and the creation of first ML models, which was also supported by P4. The creation of first results in just a few minutes was also highlighted by P5. This was complemented by P3 who added that the prototype is easy to use, with only a few commands needed for the creation of a functioning ML pipeline. Additionally, the participants remembered how to use the prototype during the exercise, indicating that the usage of the prototype is easy to learn. P1 mentioned that the prototype was good for local development with small datasets but probably was too simplistic for datasets of low quality. Meanwhile, P2 looked at the generated code of the ML pipeline, indicating that the code as well as the structure of the pipeline seem correct. Especially, the separate scaling of the training and test dataset using the same scaler object was noted as correct by P2. The data visualization of the original and intermediate datasets was noted as helpful for the testing of manually programmed data preprocessing steps by P1, indicating that the data visualization of the generated ML pipeline is similar to the data visualization on the platform "Dataiku". P2 and P3 mentioned an alternative approach to the approach using Sweetviz. Instead of using the library for automatically generating HTML reports, the execution of Jupyter notebooks could be triggered, creating custom visualizations. This could be extended by data visualization tools like Pandas Profiler [YDa23] which was mentioned by P2. Lastly, P3 noted that the plots generated by MLflow's `evaluate` and displayed in the results of a pipeline run in MLflow's UI are useful.

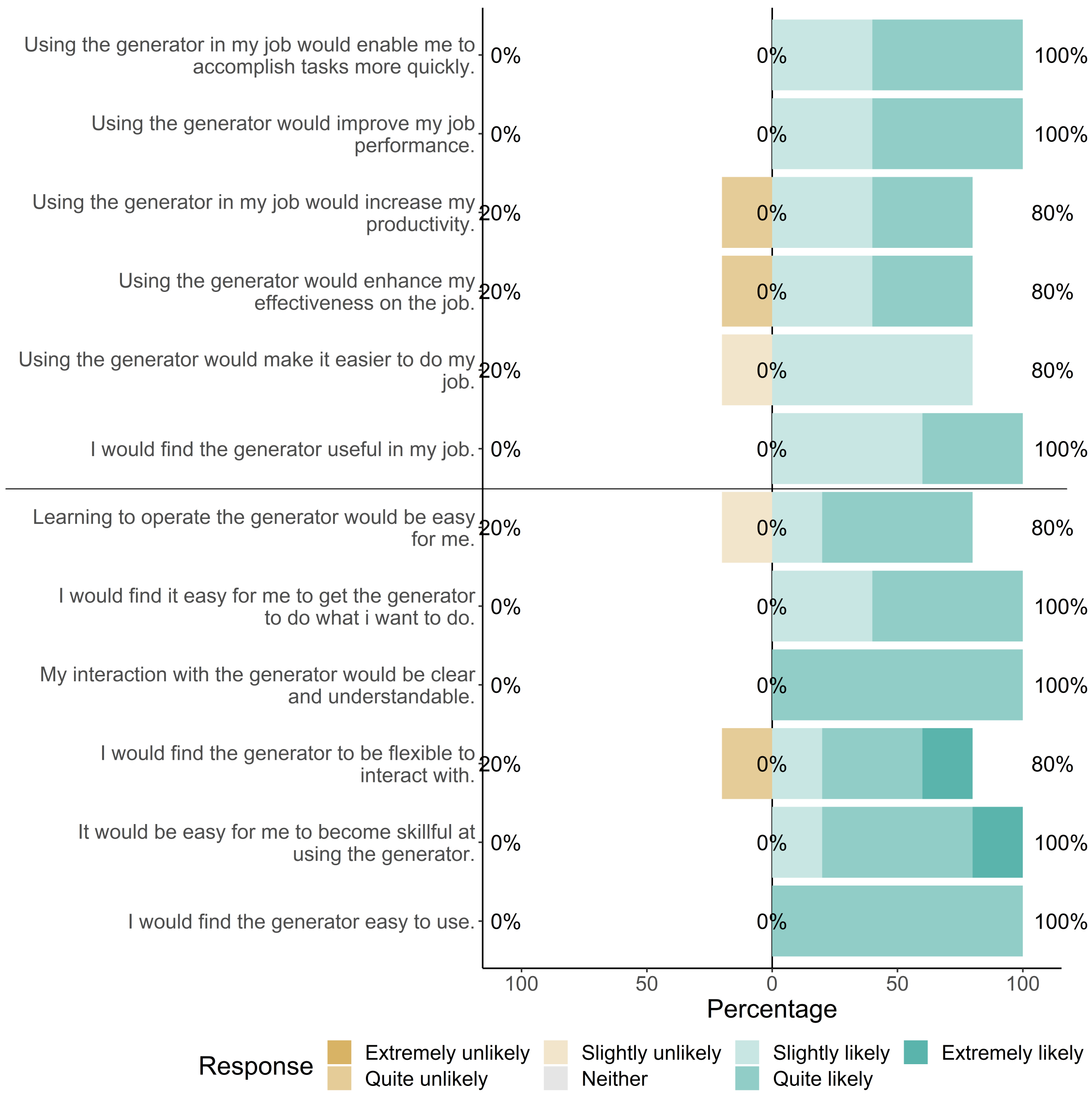

Figure 4.6: Survey results regarding usefulness (top) and ease of use (bottom)

Usefulness and Ease of Use

As mentioned in the research design, the usefulness of the generator and the ease of use of the prototype were rated by the participants in a survey. The results of the survey were generally positive as visualized in Figure 4.6. Except for five answers of “quite unlikely” or “slightly unlikely” and two answers of “extremely likely”, all questions were answered “quite likely” or “slightly likely”. So, the approach was generally valued as useful while the prototype was deemed easy to use.

Improvements

There are a variety of improvements for the prototype and the approach mentioned by the participants.

As mentioned above, the prototype was described as good for simple use cases but there was no guarantee for it to work well for more complex ones. This was added by P5, but also by P2 in relation to the joining of the datasets.
Another possible improvement mentioned above was the feature selection and feature engineering in the configuration file. All participants made suggestions like the user being able to apply preprocessing algorithms to all features as well as the exclusion of features that are not supposed to be used for model training.
The support of more preprocessing algorithms was another topic mentioned by multiple interviewees. P4 missed the inclusion of algorithms that are able to handle missing values in the data and added feature generation as a possible functionality. P3 added that there are too many algorithms that could be or needed to be provided by the generator. Which algorithm used was mainly dependent on the data. To cover all wanted algorithms as part of the ML pipeline, the preprocessing steps could be extracted from the experimentation phase, P3 suggested. P2 also mentioned that data preprocessing might be a too complex topic for automatic generation but suggested that an AI could support this process.
P1 mentioned that the support for distributed computation would be useful, distributing data preprocessing and model training steps across multiple processing units.
The documentation could also be improved which was a conclusion from the play-around phase and the exercise but was also mentioned by P2. One part mentioned in particular was that the participants were not aware of the configurations in regard to the train-test split, model training and model evaluation could easily be changed later on by changing the parameters of the respective nodes without executing the prototype again. This might be because this fact is not mentioned in the documentation of the prototype.
Suggestions regarding the frequent switching between the prototype and the Kedro project as well as the struggles with complex directory paths for the execution of the prototype included moving the prototype inside the Kedro project. All participants suggested this improvement. Another way of making the prototype part of a Kedro project could be creating a plugin for Kedro to provide the prototype's functionality.
Support for other kinds of problems and datasets was also requested by P5. Examples included kinds of datasets like images, text and videos besides the current focus on tabular data.
Additionally, P1 added that a prediction pipeline could be the output of the ML pipeline instead of the trained model. This would include the data preprocessing steps being applied to incoming data before the prediction by the model.
Asked about improvements regarding the input of the prototype, the participants were split on two options. The current approach with the configuration file was described as clear and understandable if the documentation was good but might have boundaries when it came to complex use cases according to P2 and P4. P1 and P5 preferred a graphical user interface. While P3 was not against using a configuration file but proposed a web application for changing the configuration file alongside a potential AI recommending changes to the ML pipeline.
Regarding the structure of the resulting ML pipeline, the participants were mostly content. P3 mentioned that he would have expected the feature scaling steps to be part of the data pipeline instead of the model pipeline. After the interviewer noted that this would result in data leakage

because information about the test dataset would end up in the training dataset, P3 agreed.
Furthermore, the interviewees were asked whether they would like a more general AutoML instead of the grid search approach used in the prototype. All participants agreed that a general AutoML approach would be able to replace the current grid search approach. P3 added that AutoML and the grid search approach could be used together with the grid search approach refining the hyperparameters found by the AutoML component.
Asked about the deployment pipeline, which was not part of the prototype's functionality, the participants acknowledged that model deployment was a difficult topic with many possible options for configuration. According to P5, these configurations for example included the definition of environment variables, runtimes, scripts, models, inferencing, inference speed, availability and scaling. P3 suggested that running the model inside a Docker container could be considered the standard and proposed that configurations for creating the Docker image could be collected during the pipeline run. Still, the deployment pipeline was described as a necessary part of the ML pipeline except for initial testing as P4 noted.
Additionally, a list of platforms and tools was mentioned for which the prototype could provide support. While P2 thought that the combination of Kedro and MLflow was good enough, P4 and P5 requested support for the machine learning platforms of cloud providers like Amazon Web Services (AWS) and Azure.
Regarding model deployment, P3 mentioned tools automating this task could be involved. On the other hand, P1 noted that the platform and tools depend on the project's requirements, especially the amount of data is handled and whether existing infrastructure was available and how it looked like.
Besides these improvements, some participants mentioned ideas about improving the approach. Because the prototype only created new ML pipelines, P2 mentioned that existing changes by the ML practitioner would be overwritten in case of another prototype execution. To prevent this, the participant suggested the functionality of extending existing ML pipelines by tracking existing code and adding new code snippets to the corresponding files. P2 compared this to code inserts by Git.
Another idea by P3 involved the ML practitioner only working with the prototype instead of Kedro. This consisted of the user only configuring the configuration file or some kind of other input without the need for the user to have skills and experience with machine learning in general. For this, Kedro could be the underlying pipeline orchestration tool with which the user did not work anymore.

5 Discussion

The aim of this study was to evaluate an approach of automatically generating ML pipelines. Answers for research question 1.1 regarding important components of effective ML pipelines were comparable to findings from existing literature, with components and steps as part of three sub-pipelines: the data pipeline, model pipeline and deployment pipeline. The most important quality requirements of effective ML pipelines, answering research question 1.2, include testability of single components and the interoperability between steps, flexibility of changing the ML pipeline and a modular structure of the ML pipeline and associated files. Answers to research question 2 are contextual factors influencing ML pipelines including data quality and regulatory requirements. The results from the evaluation interviews indicate that the approach proposed in this paper is viewed as a feasible approach to automatically generate context-specific ML pipelines (RQ 3).

The participants liked the fast and simple creation of ML pipelines through this approach. This shows that there is a need for setting up ML pipelines fast such that the ML practitioners are able to focus on the process of developing good performing ML models. The reduction in time is a common benefit of automation of specific processes which was clearly achieved. Another common benefit includes the reduction of mistakes. That this was achieved to some degree was shown by the comments by one of the participants regarding the separate data normalization. This topic was covered by Davis [Dav89] and would have resulted in data leakage if it was not applied that way.

Another point was the fast creation of models, which is similar to the "Feasibility Study" mentioned by Haakman et al. [HCHD21] for "failing fast" if machine learning is not suitable for the project's use case. This is especially important for the start of a project which may indicate where the approach is most useful in the ML workflow. The placement in the ML workflow might also support the standardization of ML projects which could be an additional goal of the approach. The study also showed what configurations are needed for generating ML pipelines. These include project-related configuration options (e.g., ML problem or target label) but also configurations regarding single steps (e.g., dataset and model training) as well as optional configuration regarding feature engineering and data visualization.
These configurations can be categorized differently as well. Besides necessary configurations like, e.g., the ML problem and the datasets, there are optional configurations for steps which were generated regardless and only influence generated Kedro parameters. For those steps like the train-test split, there are default values for the configurations set by the library used to implement those steps. Besides the train-test split where scikit-learn provides the implementation, MLflow's "evaluate" works similarly for the model evaluation.

The study also discovered weaknesses of this approach using a YAML file for configuration because of problems with syntax and semantics which could be traced back to insufficient documentation of the configuration input. Although some participants liked the compactness of the configuration file, complex use cases and complex configuration of pipeline steps might cause problems in the future. Limitations regarding the approach include that the joining of data might get complex, especially for some datasets. This might be less of a problem if the ML practitioner is able to make changes to the ML pipeline after its generation, but the pipeline might not function without. This results in a question of how much of these steps can be abstracted. For simple datasets, the joining might be straightforward but for complex ones this might not even be possible without interference by the ML practitioner. Generally, the components that might suffer by an increase of complexity of the use case depend heavily on the quality of the data, like the data joining and data preprocessing steps as part of the data pipeline, as well as domain knowledge in case of the feature engineering steps. Analyzing the data automatically before generating the datasets joining and data preprocessing nodes might be one way to abstract these steps further and prevent errors. How this data analysis might look like and how great the potential impact is, could be the goal of another research project. Also, depending on the future support for complex tasks, the complexity of the configuration file might get out of hand. While this was viewed as one reason for the prototype's easy use, this could become a strain on the ease of use for the ML practitioner, which is why this approach using a configuration file might be limited to generating basic ML pipelines with general solutions for possibly complex problems like the data joining.

Another interpretation of the results indicates that the iterative development of ML models is more important for the ML workflow than highlighted in the existing literature. This is based on an improvement suggestion for the prototype that includes the extension of existing ML pipelines. With this, the prototype should be able to add and remove new steps as well as adapt existing steps without changing custom programmed steps and steps that are not affected by the change in configuration of the input file. This mostly includes data preprocessing steps. One reason for this focus is the widespread adoption of AutoML. All participants specified AutoML as their primary approach to model engineering with the mentioned grid search being some kind of brute force approach to AutoML. This decreases the challenges of model training but also the importance of it, shifting the focus to data engineering. This is also the reason for the composition of the ML pipelines which were generated by the prototype. While many literatures acknowledge the importance of data preprocessing to the ML workflow, few modularize these steps into their own pipeline, a data pipeline. In papers where this modularization is applied, the pipeline concerned with model engineering is often still called machine learning pipeline. With the increased usage of AutoML and the resulting focus on the data engineering and the data pipeline, there is less focus on the ML pipeline. This is why in the generated ML pipeline part concerned with model engineering is called "Model Pipeline". Together with the data pipeline and the deployment pipeline, the model pipeline builds the base for the machine learning pipeline.

Even though the approach was deemed useful by the participants, it can be refined to improve the transition from experimentation to pipeline further. The repetitive process of experimentation followed by the effort of transferring the data preprocessing steps into a pipeline might not be fully solved by such an approach. For this, the add-on "Kale" for the platform "Kubeflow" might be an inspiration. The ML practitioner is able to annotate cells inside a Jupyter Notebook and "Kale" is then able to generate a Kubeflow Pipeline. This makes the transition from experimentation to packaging the code into a pipeline much easier and faster. Still, the ML practitioner needs to write all of the code himself. To be able to adapt the ML pipeline more easily, the ML practitioner would need to experiment with different data preprocessing steps inside a Jupyter Notebook which might be solved by generating a Jupyter Notebook from a pipeline and back into a pipeline after the changes by the ML practitioner. One difficulty of this approach is that errors are not automatically prevented but need to be detected and at least notified to the user or remedied automatically.

The study also highlighted that there are similarities in terms of some ML pipeline steps that make them prime candidates to be automatically generated. Those include for example the model training or engineering step that only involves the execution of some AutoML algorithm. Other areas of ML pipelines are key areas in the ML workflow. These areas like feature engineering steps change often during iterations of the ML workflow and play a role in improving performance of newly trained models. That is why, the prototype is not able to provide such a functionality as it only generates ML pipelines from scratch. By adding the functionality to extend existing ML pipelines, the prototype would be able to help the ML practitioner in changing the ML pipeline quickly. Another potential improvement that could address this issue includes the usage of AI as part of the experimentation process. With an AI for analyzing the data and recommending pipeline steps and structure, the experimentation could be changed from a manual process to an automatic one. This could involve an AutoETL approach which analyzes data and feature importance as well as selects features and generates new ones. Depending on how well the AI would be able to grasp knowledge about the domain, the process might be automatable completely.

The prototype partially fulfills the quality requirements gathered during the exploration interviews. By generating ML pipelines for the framework Kedro, the prototype enables the ML practitioner to adapt the resulting ML pipeline after the generation which fulfills the flexibility quality requirement. Kedro also provides a modular structure with pipelines, nodes and hyperparameters in separate files in a modular manner. This was also mentioned as an important quality requirement during the exploration interviews. The framework also covers the quality requirement of reusability by providing the functionality to package the pipeline and make it possible to reuse it in other Kedro projects. Other quality requirements are not fulfilled by the prototype. Those include the testability, pipeline performance and good documentation. While the evaluation interviews showed that the last one was not achieved, the others were not focused on during the development of the prototype. Good execution performance of the ML pipeline could be achieved by, for example, adding the functionality of parallel model training to the prototype. Testability on the other hand might be difficult to fulfill. Automatically generating tests for the ML pipeline might be simple to implement for components which are similar in different ML pipelines, but other steps might not. Automatically generating tests for feature engineering steps, especially integration tests, could be difficult to achieve. This could also be the focus of another research study.

Lastly, while automatically generating a deployment pipeline was not implemented in the prototype, this is still an important component of ML pipelines. Therefore, there are more information on components and configurations to be gathered. How the deployment pipeline generally could be split into separate parts, might be helpful in getting more insight into this component. This could also involve the separation into components or steps that are commonly used which might be an entry into how this pipeline could be automatically generated. Also those components or steps should be able to generate artifacts for, otherwise these might not be automatically generated. One of those commonly used steps might be to package the ML model into a Docker container. Creating a Docker image for the model to be deployed into could be a pipeline step that could be automatically generated.

Threats to Validity

There are several threats to the validity of this study. A threat to the conclusion validity is the researcher bias. Because the observation, the comments by the participants and the analysis of the qualitative data were carried out by a single researcher, there is the possibility that the results would differ if the study would have been carried out by a different researcher. The extent to which the results would differ though is probably relatively limited.

The biggest threat to external validity is the type of this study, a case study. Using the prototype at another company and use case outside of the given exercise probably would have resulted in more errors and other problems the interviewees would have addressed.

Threats to internal validity includes the difference between what was said by the interviewees and what the interviewer understood during the interviews. This was reduced by the interviewees reviewing what the interviewer transcribed after the corresponding interview and correcting potential misunderstandings. With the participants gaining more experience using the prototype, they would have been able to provide better feedback. This is another threat to internal validity. Because all interviewees as well as the interviewer were employees of the adesso SE during the time of the study, the interviewees might give more positive feedback than without the relationship. Another threat to internal validity is the interaction between interviewer and interviewee. Because of the help the interviewer provided during the exercise of the evaluation interviews by answering questions, bringing the participant back on track and highlighting mistakes for example in the syntax of the configuration file, the interviewer helped curtain participants more than others. Lastly, the Hawthorne effect is another threat to the external validity of the study. Because the experimentation with the prototype took place during the interview, there could have been different results if the participants were tasked with the exercise outside the interview setting.

6 Conclusion

This case study proposed an approach to automatically generate ML pipelines depending on context provided by ML practitioners. Based on a qualitative analysis of results from expert interviews at adesso SE, important components and quality requirements of effective ML pipelines were identified. Furthermore, project- and user-specific context that influences ML pipeline steps and quality requirements were determined. Based on the results of the analysis, a prototypical approach was developed which was evaluated by experts. The quantitative and qualitative analysis of those evaluations shows that the approach is deemed useful and easy to use by ML practitioners. Additionally, it was described as good for the start of an ML project.

For the productive use in real-life ML projects, the prototype needs to be developed further with more added functionalities. These might include the handling of complex use cases and datasets, most notably low-quality datasets and different data formats. Future research projects might study the tools and visualizations that are important for ML practitioners to develop ML models depending on the project's context. Additionally, the possibilities of ML model deployment and how those could work would be an interesting topic for a research project.

With these results, this case study fills the research gap of the automatic generation of ML pipelines for the purpose of supporting and increasing the speed of how ML practitioners work while existing literature focus on automatically generating ML models or ML pipelines for the purpose of improving the model's performance.

Bibliography

[ABB+19] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, T. Zimmermann. “Software engineering for machine learning: A case study”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pp. 291–300 (cit. on pp. 20, 21, 39).

[Abd07] H. Abdi. “Z-scores”. In: *Encyclopedia of measurement and statistics* 3 (2007), pp. 1055–1058 (cit. on p. 56).

[Apa] Apache Software Foundation. *PySpark*. URL: https://spark.apache.org/docs/latest/api/python/index.html (cit. on p. 39).

[Apa06] Apache Software Foundation. *Hadoop*. Apr. 1, 2006. URL: https://hadoop.apache.org/ (cit. on p. 39).

[Apa10] Apache Software Foundation. *Hive*. Oct. 1, 2010. URL: https://hive.apache.org/ (cit. on p. 36).

[BBG+23] M. R. Berthold, D. Brookhart, S. Gerber, S. Hayasaka, M. Widmann. “Towards Data Science Design Patterns”. In: *Advances in Intelligent Data Analysis XXI: 21st International Symposium on Intelligent Data Analysis, IDA 2023, Louvain-la-Neuve, Belgium, April 12–14, 2023, Proceedings*. Springer. 2023, pp. 55–64 (cit. on pp. 24, 48, 57).

[Ber19] L. Berti-Equille. “Learn2clean: Optimizing the sequence of tasks for web data preparation”. In: *The World Wide Web Conference*. 2019, pp. 2580–2586 (cit. on p. 26).

[Ber22] F. Bertrand. *Sweetviz*. Version 2.1.4. 2022 (cit. on p. 51).

[CM18] M. Chui, S. Malhotra. *AI adoption advances, but foundational barriers remain*. https://www.mckinsey.com/featured-insights/artificial-intelligence/ai-adoption-advances-but-foundational-barriers-remain. 2018 (cit. on p. 15).

[Cor16] M. Corporation. *What is the Team Data Science Process?* https://learn.microsoft.com/en-us/azure/architecture/data-science-process/overview. [Online; accessed 29-June-2023]. 2016 (cit. on pp. 18, 19).

[Cro23] Cross-industry standard process for data mining. *Cross-industry standard process for data mining — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-June-2023]. 2023. URL: https://en.wikipedia.org/wiki/Cross-industry_standard_process_for_data_mining (cit. on p. 18).

[Dat13] Databricks, Inc. *Databricks*. 2013. URL: https://www.databricks.com/ (cit. on p. 39).

[Dav89] F. D. Davis. “Perceived usefulness, perceived ease of use, and user acceptance of information technology”. In: *MIS quarterly* (1989), pp. 319–340 (cit. on pp. 33, 69).

[dDS22] G. d'Aloisio, A. Di Marco, G. Stilo. "Modeling Quality and Machine Learning Pipelines through Extended Feature Models". In: *arXiv preprint arXiv:2207.07528* (2022) (cit. on pp. 28, 40).

[DMRM19] B. Derakhshan, A. R. Mahdiraji, T. Rabl, V. Markl. "Continuous Deployment of Machine Learning Pipelines." In: *EDBT*. 2019, pp. 397–408 (cit. on p. 15).

[FPS96] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth. "The KDD process for extracting useful knowledge from volumes of data". In: *Communications of the ACM* 39.11 (1996), pp. 27–34 (cit. on pp. 17, 18).

[GBA22] J. Giovanelli, B. Bilalli, A. Abelló. "Data pre-processing pipeline generation for AutoETL". In: *Information Systems* 108 (2022), p. 101957 (cit. on p. 26).

[HCHD21] M. Haakman, L. Cruz, H. Huijgens, A. van Deursen. "AI lifecycle models need to be revised: An exploratory study in Fintech". In: *Empirical Software Engineering* 26 (2021), pp. 1–29 (cit. on pp. 21, 69).

[HMR+19] W. Hummer, V. Muthusamy, T. Rausch, P. Dube, K. El Maghraoui, A. Murthi, P. Oum. "Modelops: Cloud-based lifecycle management for reliable and trusted ai". In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2019, pp. 113–120 (cit. on pp. 28–30).

[HVKR20] Y. Heffetz, R. Vainshtein, G. Katz, L. Rokach. "Deepline: Automl tool for pipelines generation using deep reinforcement learning and hierarchical actions filtering". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 2103–2113 (cit. on p. 26).

[IBM22] IBM. *IBM Global AI Adoption Index 2022*. https://www.ibm.com/watson/resources/ai-adoption. 2022 (cit. on p. 15).

[Inc23] G. Inc. *Gartner Glossary: ModelOps*. Gartner Inc., 2023. URL: https://www.gartner.com/en/information-technology/glossary/modelops (cit. on p. 28).

[JOB20] M. M. John, H. H. Olsson, J. Bosch. "Developing ml/dl models: A design framework". In: *Proceedings of the International Conference on Software and System Processes*. 2020, pp. 1–10 (cit. on pp. 27, 41).

[KMF+17] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu. "Lightgbm: A highly efficient gradient boosting decision tree". In: *Advances in neural information processing systems* 30 (2017), pp. 3146–3154 (cit. on p. 39).

[KRP+16] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing. "Jupyter Notebooks – a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides, B. Schmidt. IOS Press. 2016, pp. 87–90 (cit. on p. 40).

[LLC23] G. LLC. *MLOps: Continuous delivery and automation pipelines in machine learning*. Google LLC, 2023. URL: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning (cit. on pp. 23–25, 38).

[Mai23] A. D. Maisch. *Towards Automatically Generating Context-Specific ML Pipelines: A Case Study at adesso SE*. July 2023. DOI: 10.5281/zenodo.8128447. URL: https://doi.org/10.5281/zenodo.8128447 (cit. on p. 16).

[McK10] W. McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. Ed. by S. van der Walt, J. Millman. 2010, pp. 51–56 (cit. on pp. 39, 55).

[Mer14] D. Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2 (cit. on p. 49).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. “Scikit-learn: Machine learning in Python”. In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830 (cit. on p. 51).

[RH09] P. Runeson, M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical software engineering* 14 (2009), pp. 131–164 (cit. on p. 31).

[SBD+21] S. Studer, T. B. Bui, C. Drescher, A. Hanuschkin, L. Winkler, S. Peters, K.-R. Müller. “Towards CRISP-ML (Q): a machine learning process model with quality assurance methodology”. In: *Machine learning and knowledge extraction* 3.2 (2021), pp. 392–413 (cit. on pp. 19, 48, 57).

[SBHV20] A. Serban, K. van der Blom, H. Hoos, J. Visser. “Adoption and effects of software engineering best practices in machine learning”. In: *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2020, pp. 1–12 (cit. on pp. 21, 22, 40).

[SFR23] M. Steidl, M. Felderer, R. Ramler. “The pipeline for the continuous development of artificial intelligence models—Current state of research and practice”. In: *Journal of Systems and Software* (2023), p. 111615 (cit. on p. 27).

[The14] The Kubernetes Authors. *Kubenetes*. The Linux Foundation, Sept. 9, 2014. URL: https://kubernetes.io/ (cit. on p. 49).

[The15] The TensorFlow development team. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on p. 39).

[The18a] The Kubeflow Authors. *Kubeflow*. Apr. 5, 2018. URL: https://www.kubeflow.org/ (cit. on p. 49).

[The18b] The mlflow development team. *MLflow*. LF Projects, LLC, June 5, 2018. URL: https://www.databricks.com/ (cit. on pp. 39, 50).

[The23a] The Kedro development team. *Kedro*. Version 0.18.11. July 2023. URL: https://github.com/kedro-org/kedro (cit. on pp. 39, 50).

[The23b] The kedro-mlflow development team. *kedro-mlflow*. Version 0.11.8. 2023. URL: https://github.com/Galileo-Galilei/kedro-mlflow (cit. on p. 50).

[VD09] G. Van Rossum, F. L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697 (cit. on pp. 32, 63).

[Ven19] VentureBeat. *Why do 87% of data science projects never make it into production?* https://venturebeat.com/ai/why-do-87-of-data-science-projects-never-make-it-into-production/. 2019 (cit. on p. 15).

[VKB+23] L. Visengeriyeva, A. Kammer, I. Bär, A. Kniesz, M. Plöd. *MLOps*. innoQ Deutschland GmbH, 2023. URL: https://ml-ops.org/ (cit. on pp. 19, 21, 23, 49).

[WbK+21] R. Westenra, S. bot, K. Kunii, bru5, S. Wong, L. Hoang, G. Wrigley, D. Deriabin, L. Bălan, A. Ivaniuk, rashidakanchwala, Z. Patel, M. Theisen, N. Khan, W. Walker, Y. Dada, A. B. Potje, T. Nguyen, Y. Minami, O. Kelleghan. *quantumblacklabs/kedro-viz:* version 3.11.0. Apr. 2021. DOI: 10.5281/zenodo.4701066. URL: https://doi.org/10.5281/zenodo.4701066 (cit. on pp. 50, 79–81).

[WH00] R. Wirth, J. Hipp. "CRISP-DM: Towards a standard process model for data mining". In: *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*. Vol. 1. Manchester. 2000, pp. 29–39 (cit. on p. 18).

[YDa23] YData Labs Inc. *ydata-profiling: Exploratory Data Analysis for Python*. 2023. URL: https://github.com/ydataai/ydata-profiling (cit. on p. 64).

All links were last followed on July 06, 2023.

A Kedro Pipelines

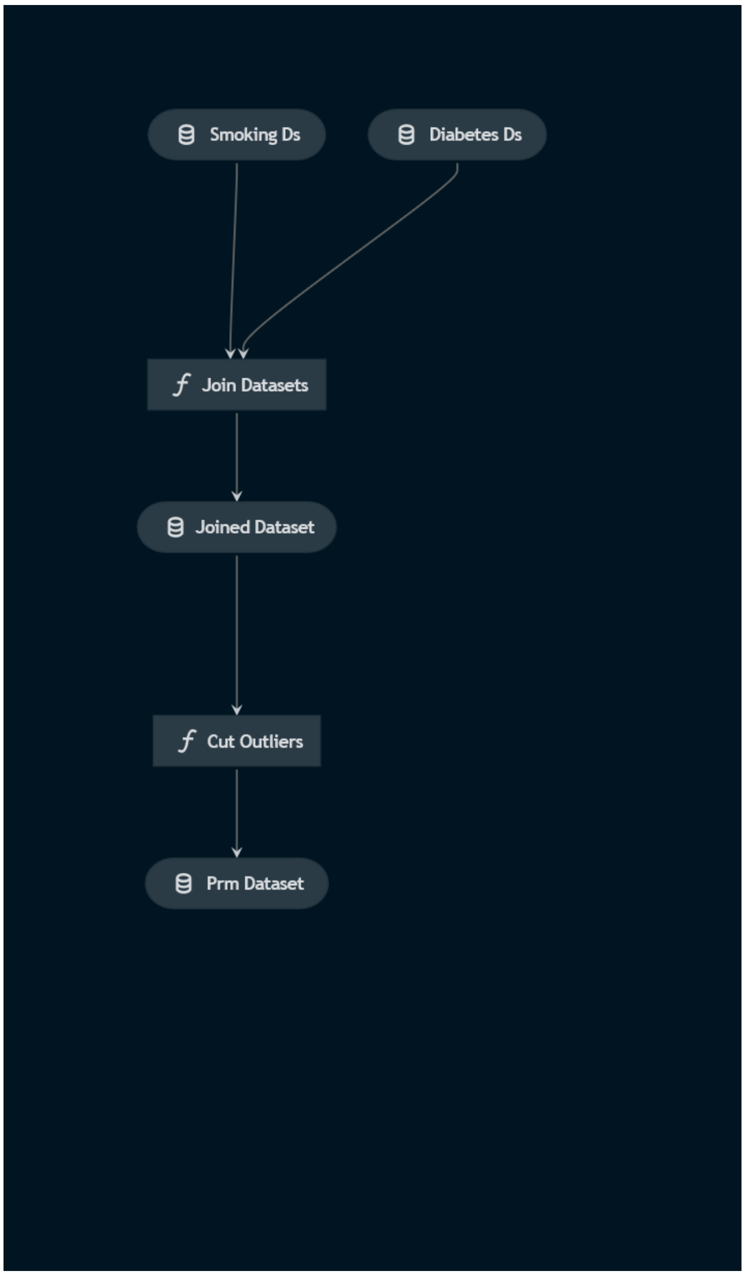

Figure A.1: Visualization of the data pipeline of a generated ML pipeline using Kedro-viz [WbK+21]

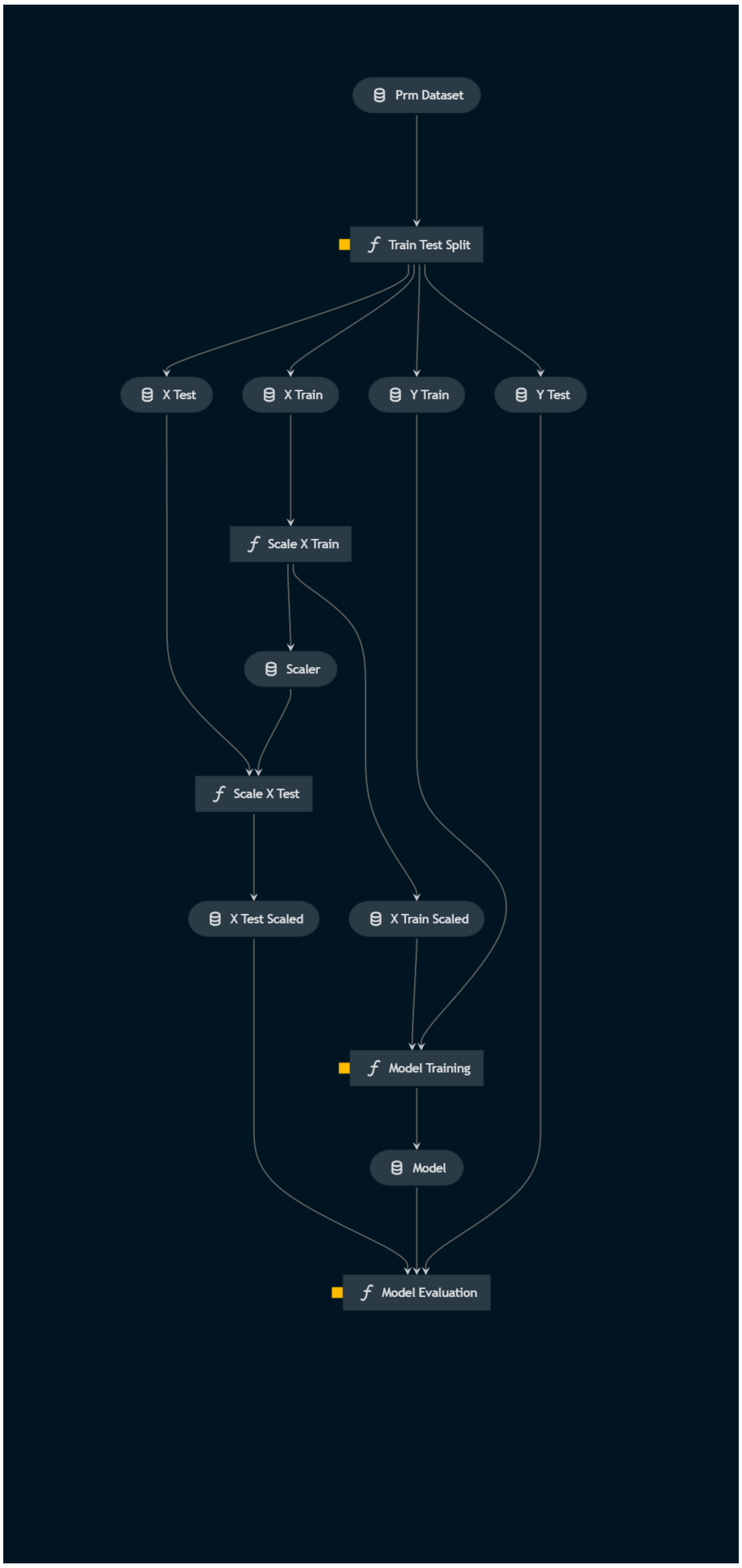

Figure A.2: Visualization of the model pipeline of a generated ML pipeline using Kedro-viz [WbK+21]

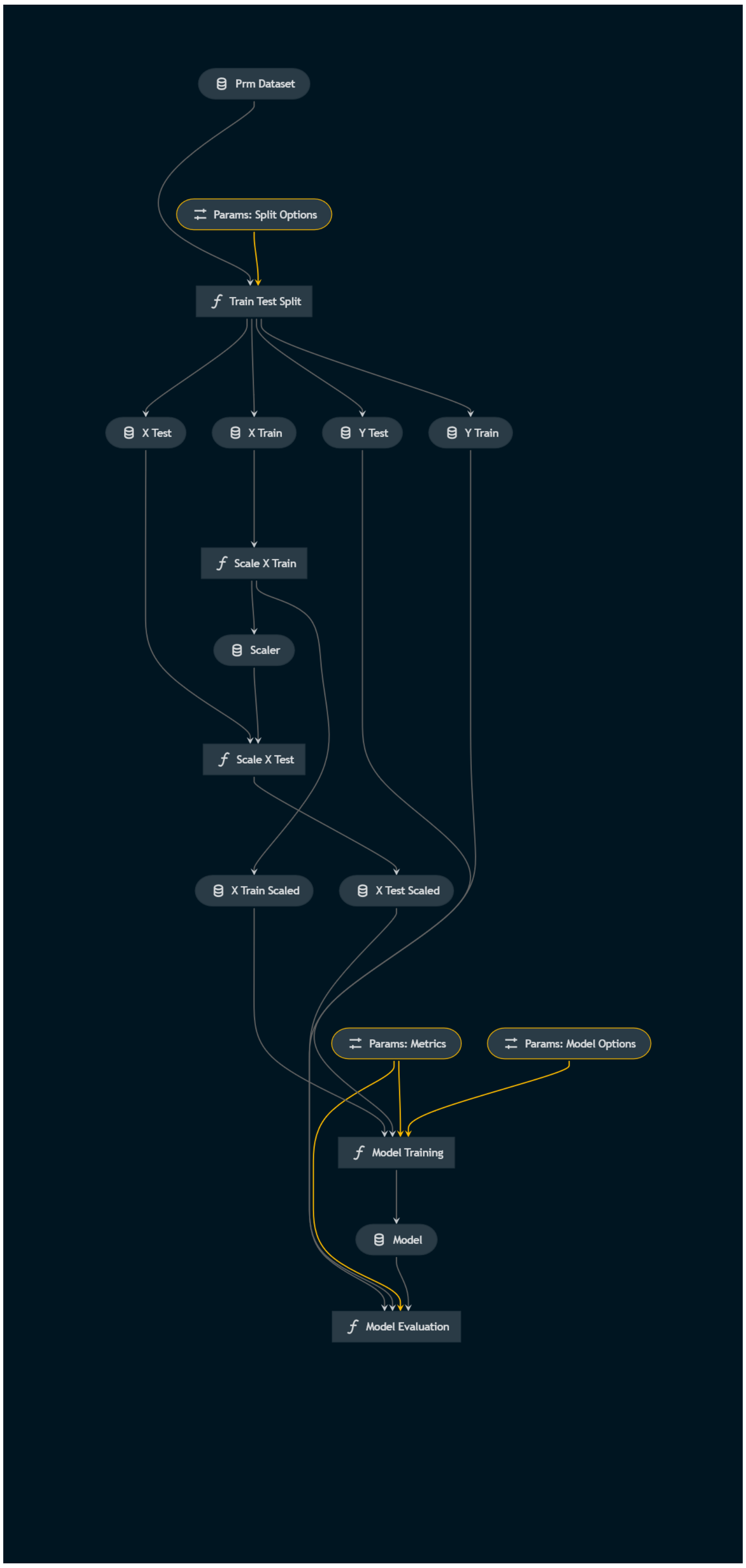

Figure A.3: Visualization of the model pipeline including parameter nodes using Kedro-viz [WbK+21]