FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Tackling Version Management and Reproducibility in MLOps

Priscilla Dias Melin

Master in Software Engineering

Supervisor: André Restivo, Assistant Professor

Second Supervisor: Carlos Soares, Associate Professor

July 24, 2023

Tackling Version Management and Reproducibility in MLOps

Priscilla Dias Melin

Master in Software Engineering

Approved in oral examination by the committee:

Chair: João Carlos Pascoal Faria
External Examiner: Paulo Jorge de Sousa Azevedo
Supervisor: André Monteiro de Oliveira Restivo

July 24, 2023

Resumo

A crescente adoção de soluções baseadas em machine learning (ML) exige avanços na aplicação das melhores práticas para manter estes sistemas em produção. Operações de machine learning (MLOps) incorporam princípios de automação ao desenvolvimento de modelos, promovendo entrega, monitoramento e treinamento contínuos. Devido a vários fatores, como a necessidade de otimizações derivadas de mudanças nas necessidades de negócios, espera-se que os cientistas de dados criem vários experimentos para desenvolver um modelo ou preditor que atenda satisfatoriamente aos principais desafios de um dado problema.

A execução de cada experimento gera metadados, como configurações de ambiente, versões de código-fonte e dados, métricas e configurações de hiperparâmetros. Esses metadados são conhecidos como artefatos ou ativos de ML. O rastreamento e o gerenciamento dessas informações são necessários para alcançar a rastreabilidade do processo de geração dos modelos, facilitando a recriação do ambiente em que determinada versão de um modelo foi originada. Vincular informações de experimentos, modelos, conjuntos de dados, configurações e alterações de código requer organização, rastreamento, manutenção e controle de versão adequados.

Este trabalho investigou as melhores práticas, problemas atuais e desafios relacionados ao gerenciamento e controle de versão de artefatos. Usando ferramentas existentes, desenvolvemos um estudo de caso que permitiu o desenvolvimento e operacionalização de um fluxo de trabalho de ML com foco na gestão dos artefatos gerados, aplicando princípios de MLOps para facilitar o rastreamento e a reprodutibilidade do modelo.

A abordagem baseada em estudo de caso contribuiu para coletar conhecimento empírico sobre ferramentas com foco em gerenciamento de versões, o que nos permitiu criar tutoriais sobre casos de uso comuns. Cenários cobrindo preparação de dados, geração de modelo, comparação entre versões de modelo, implantação, monitoramento, depuração e retreinamento demonstraram que o estudo de caso não apresentou problemas na recriação do ambiente de desenvolvimento de uma determinada versão de modelo. Por fim, este trabalho fornece orientação para quem deseja automatizar e operar fluxos de trabalho de ML com controle de versão de artefatos como requisito central.

Keywords: Mlops, Machine Learning Engineering, ML Versioning, Reproducibility.

Abstract

The growing adoption of machine learning solutions requires advancements in applying best practices to maintain artificial intelligence systems in production. Machine Learning Operations (MLOps) incorporates DevOps principles into machine learning development, promoting automation, continuous delivery, monitoring, and training capabilities. Due to multiple factors, such as the experimental nature of the machine learning process or the need for model optimizations derived from changes in business needs, data scientists are expected to create multiple experiments to develop a model or predictor that satisfactorily addresses the main challenges of a given problem.

Every experiment training run generates metadata, such as environment configuration, source code, data versions, metrics, and hyperparameters configurations. This metadata is known as ML artifacts or assets. Tracking and managing this information is necessary to achieve ML traceability and model reproducibility by facilitating the recreation of the environment that originated a given model version. Linking information from experiments, models, datasets, configurations, and code changes requires proper organization, tracking, maintenance, and version control.

This work investigated the best practices, current issues, and open challenges related to artifact versioning and management. Using existing tools, we developed a case study that enabled the development and operationalization of an ML workflow focusing on data, models, code, and configuration versioning, applying principles of MLOps to facilitate reproducibility by creating traceability between ML artifacts.

The case study approach contributed to collecting empirical knowledge on MLOps tools focusing on versioning capabilities, which enabled us to create tutorials on common use cases. Scenarios covering data preparation, model generation, comparison between model versions, deployment, monitoring, debugging, and retraining demonstrated that the case study presented no issues in recreating the development environment of a given model version. Finally, this work provides guidance for those who want to automate and operate ML workflows with artifact versioning as a central requirement.

Keywords: Mlops, Machine Learning Engineering, ML Versioning, Reproducibility.

Acknowledgments

To my advisor, Professor André Restivo, and my co-advisor — Professor Carlos Soares, thank you for the guidance, suggestions, and incentive while pursuing this research topic.

To Rafael, thank you for being an outstanding partner. I could not have done this work without your support. I love you always.

Last but not least, thanks to my family and friends. Especially the fantastic women that were part of my support network during this time. Rosangela, thanks for always believing in me. Natália, thanks for the listening and encouragement. Mariza, thanks for the conversations and laughters. Giulia, thanks for being the kind of person that makes one believe that everything is possible as long as one has support, love, and friendship.

You are all inspiring and dear to me. Thank you for everything.

Priscilla

"The greatest gifts we can give our children are the roots of responsibility and the wings of independence."

Maria Montessori

Contents

List of Figures

List of Tables

Abreviaturas e Símbolos

AI	Artificial Intelligence
CD	Continuous Delivery
CD4ML	Continuous Delivery for Machine Learning
CI	Continuous Integration
CT	Continuous Training
CM	Continuous Monitoring
DevOps	Software Development and Operations
DVC	Data Version Control
GTO	Git Tag Ops
KPIs	Key Performance Indicators
ML	Machine Learning
MLOps	Machine Learning Operations
SE	Software Engineering
SDLC	Software Development Life Cycle
SLR	Systematic Literature Review

Chapter 1

Introduction

The prospect of artificial intelligence (AI) adoption keeps continuing on a steady rise. The intensive development of this field in the past years has contributed to growing the adoption of machine learning (ML) solutions to attend a diverse range of businesses, particularly in contact-center automation, service-operations optimization, and AI-based enhancement products [35]. Once machine learning solutions are mature enough and organizations recognize them as core to advance and sustain their business goals, a need for sophistication takes place. Maintaining artificial intelligence systems in production enhances the system's overall complexity, requiring tooling advancements and the application of best practices to build and maintain the infrastructure that supports the execution of ML pipelines.

1.1 Context

Machine Learning Operations (MLOps) is an approach to streamline the process of taking machine learning models to production, based on a set of practices derived from the DevOps – a combination between Software Development and Operations – mindset. The concept behind this practice involves collaboration, quality assurance, and automation. The fact that machine learning systems deploy models (a mixture of code and data) into production instead of purely code is one aspect that differentiates DevOps from MLOps. Principles such as Continuous Integration (CI), Continuous Delivery (CD), Continuous Training (CT), and Continuous Monitoring (CM) are process automation mechanisms that enable the retraining of models combining machine learning development with deployment and monitoring processes frequently using cloud resources.

Many artifacts are produced throughout the development and execution of machine learning pipelines due to the constant inflow of new data and the inherent need for experimentation in this field. Data distribution changes coming from new data or changes in prediction targets may indicate changes in model performance [33]. For this reason, data is constantly gathered, analyzed, and monitored to determine if a model needs to be retrained or eventually optimized. New data, models, and code versions are produced if that is the case. Changes in the code or the input data can

possibly lead to model optimizations, reaffirming the need for synchronized artifacts versioning [25].

1.2 Motivation

ML systems propose additional complexities if compared to traditional systems. These complexities include different levels of dependencies, additional model artifacts, more data process steps, and more complex relationships between them [52]. Changes in business requirements, platforms, libraries, middleware underneath deployed applications, and compatibility issues related to pipeline integration and dependencies [29] can increase the complexity in the maintenance of deployed systems leading to challenges in artifact and change management.

Challenges in the integration between ML frameworks may also impose difficulties in gathering a holistic view of the lineage between data, code, models, and configurations. Furthermore, many model variants may coexist, including those personalized for subsets of users [25]. Hidden technical debts increases as applications evolve, leading to the behavior of applications deviating from initial expectations. Some of this problem's sources are data dependency, model complexity, reproducibility, testing, monitoring, and external response changes [36]. Also, maintainability costs tend to rise as actions to test, update, monitor, and maintain applications in production become more complex.

1.3 Problem Statement and Goals

Research database searches conducted on January 2023 demonstrated that many studies on MLOps have focused on architectural design, focusing on components, activities, and challenges. However, there is still room to advance regarding specific guidance implementation or instructions for managing ML artifacts focusing on versioning issues. The need for ML artifact management and lineage is mentioned as prerequisites to attain model reproducibility. However, most studies still need to provide codified architectural knowledge, techniques, patterns, or documentation surrounding artifact versioning and management.

Versioning management as a core design principle of MLOps is necessary to achieve lineage between code, configurations, data, and models and to enable model reproducibility. Experiments, datasets, configurations, code, and models must be stored and versioned to enable model maintenance and comparison between experiments and facilitate roll-back operations to a prior high-performer model whenever necessary. For debugging purposes, it is essential to understand which libraries, configurations, or code changes are related to each model version. Suppose these artifacts are not properly versioned and linked together. In that case, it can be hard to determine if the new version of the solution is better than the previous one, since it would be hard to achieve model reproducibility and traceability.

This study will revise the state-of-the-art related to artifact versioning and management, identifying different approaches, best practices, and open challenges. The principles, components, and

tools identified in that phase will be applied in a case study built around a model pipeline developed for this project. Automated infrastructure will be integrated to enable basic MLOps features, such as automation of CI/CD workflows and continuous monitoring. We will then simulate usage scenarios to assess if the selection of existing tools to assist in artifact versioning and management facilitated model tracking and reproducibility goals. Finally, this dissertation aims to contribute to the machine learning engineering field, specifically focusing on versioning issues to facilitate the reproducibility of models by tracking and linking information from experiments, models, datasets, configurations, and code changes.

1.4 Structure

In addition to the introduction, this document is structured as follows: Chapter 2 provides background knowledge about machine learning, DevOps, and their particularities, followed by an overview of the MLOps lifecycle. It also describes the methodology used to conduct a Systematic Literature Review (SLR) on the topic with a state-of-the-art description. Chapter 3 indicates the study scope, the hypothesis, the methodology, and general goals. It also describes the implementation and architecture developed for the case study, while Chapter 4 presents the validation process by describing and implementing scenarios. Finally, Chapter 5 presents the conclusions.

Chapter 2

Literature Review

This chapter begins with an overview of MLOps, including information on its main concepts and lifecycle. A description of a known approach to implementing ML systems is also presented to exemplify how the main concepts are related to each other. Next, a Systematic Literature Review (SLR) in search of state-of-the-art for Machine Learning Operations (MLOps) concerning versioning issues is presented, indicating tools, best practices, and open challenges in this field.

2.1 An Overview of MLOps

This section introduces the field of MACHINE LEARNING ENGINEERING by explaining the particularities and intersections between two different areas: Machine Learning (ML) and Software Development and Operations (DevOps). This interdisciplinary field between data science, software engineering and operations emerges with the need to integrate AI Software with broader systems while considering infrastructure and monitoring capabilities. Its practical application is complex, and its study is necessary to advance machine learning applications' success rate in production environments.

2.1.1 Machine Learning

Machine Learning is a branch of computer science which uses computational algorithms and data to imitate human intelligence. Given data, several operations are carried out to prepare a dataset. An algorithm obtains a model from the dataset capable of producing an estimate about a pattern in the data. The model is then evaluated to assess if the goals are met. Data scientists are expected to create multiple experiments in different stages of data manipulation to develop a model or predictor that satisfactorily addresses the problem's main challenges and accurately presents results based on the business and data mining goals. This process is highly experimental and often time-consuming. It involves different stages with back-and-forth interactions between the domain experts and ML specialists. AI Software may evolve more frequently [34] since it has

Algorithm type	Name	MLOps considerations
Linear	Linear regression	There is a tendency for overfitting.
	Logistic regression	There is a tendency for overfitting.
Tree-based	Decision tree	Can be unstable—small changes in data can lead to a large change in the structure of the optimal decision tree.
	Random forest	Predictions can be difficult to understand, which is challenging from a Responsible AI perspective. Random forest models can also be relatively slow to output predictions, which can present challenges for applications.
	Gradient boosting	Like random forest, predictions can be difficult to understand. Also, a small change in the feature or training set can create radical changes in the model.
Deep learning	Neural networks	In terms of interpretability, deep learning models are almost impossible to understand. Deep learning algorithms, including neural networks, are also extremely slow to train and require a lot of power (and data). Is it worth the resources, or would a simpler model work just as well?

Table 2.1: Algorithm selection and MLOps challenges (reproduced from [47]).

a data-centric model development process based on non-linear characteristics with several feedback loops [18, 52]. The relationship between algorithm selection and their impact on MLOps is illustrated in Table 2.1.

After developing a model, experts must maintain and eventually update it until goals are met or as new challenges emerge from business rule changes. Many process methodologies applied to data science are available regardless of ML teams' traditional, agile, or hybrid approach. As seen in Figure 2.1, even though agile approaches are gaining track on the market, a recent poll conducted in 2022 identified that CRISP-DM remains the most popular framework for executing data science projects.

2.1.1.1 CRISP-DM methodology

According to this methodology, there are six phases in the development of ML pipelines that are parts of an iterative process.

1. **Business Understanding**: This phase encompasses all the work between the domain expert and the data scientist. The goal is to define and translate the problem into a data analytics problem.

2. **Data Understanding**: This phase involves collecting the necessary data and data visualization to summarize it better so it can be later used to test the hypothesis, gain initial insights and detect possible data quality problems.

3. **Data Preparation**: This segment encompasses different processes, such as data transformation and cleaning. This pre-processing step usually includes data cleaning, augmenting,

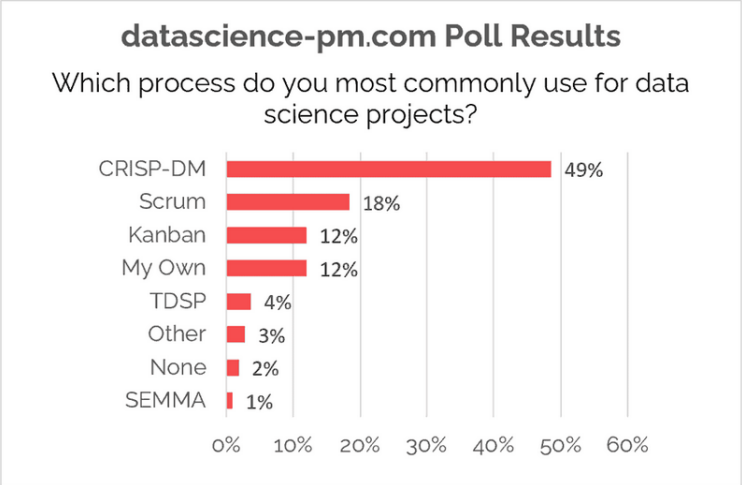

Figure 2.1: datascience-pm.com Poll Results [7]

labeling, imputation of missing values, and encoding operations. At the end of this phase, data must be consistent and versioned so that all transformations to the data must be traced back if needed, just like software is versioned due to code changes [17].

4. **Modeling**: In this phase, the data scientist experiments with different machine-learning techniques to determine the most appropriate one to tackle the problem. In order to achieve the best results, each chosen algorithm is trained with different hyperparameters and datasets to produce a model. Each iteration in this process produces an **experiment**. This element represents the connection between data, models, configurations, and results.

5. **Evaluation**: In modeling, the evaluation is done according to data mining goals. When these are achieved, the evaluation phase is done to assess the business goals.

6. **Deployment**: This phase involves the integration of the solution in the business process. It is also necessary to plan monitoring and maintenance strategies.

2.1.1.2 Agile methodologies

There are many options for agile frameworks that can be applied to managing data science projects. This methodology is best applied in projects with fast-paced and changing requirements. Frequent releases promote feedback loops to build the project's outcomes, and the requirements are continuously reviewed over the development process. Table 2.2 summarizes the most cited processes indicated on the poll results.

Approach	Description	Strengths	Challenges	Best For...
Kanban	Visualize flow. Minimize work-in-progress.	• Simple • Combines well with other frameworks • Maximizes throughput • Minimizes waste	• Least definitive • Lots of ambiguity	• Starting with a solid core set of principles and building a framework on top of it
Scrum	Well-known Agile approach focused on fixed-length iterations	• Quick, incremental value focus • Well-defined feedback loop • Strong team focus	• Time-boxing can be restrictive • Often poorly implemented • Management might get in the way	• Agile teams who need discipline provided by fixed time cycles • Radical innovation cultures
Data Driven Scrum	Agile framework specifically designed for data science teams	• Most of same benefits of Scrum and Kanban •Caters to experimentation • Relaxes Scrum pain points	• Not as vetted as Scrum • Adds challenges of managing concurrent iterations	• Teams with strong experimental culture • Data science teams that struggled with Scrum

Table 2.2: Agile Frameworks for Data Science (reproduced from [4])

2.1.2 Software Development & Operations (DevOps) applied to ML

DevOps is a set of practices derived from the combination of Software Development and Operation stages within the Software Development Life Cycle (SDLC). Collaboration, quality assurance, and automation are pillars of the DevOps mindset. Principles such as **Continuous Integration(CI)** and **Continuous Delivery(CD)** are process automation mechanisms that enable the retraining of models using new data with minimum effort. With CI, developers continuously integrate and merge development code. Tests and quality checks are automatically applied before any change enters the production environment. The model version is fully built, packaged, and validated with CD. Other principles rely on test automation, fast feedback loops, fault injection into the production environment to test how the system deals with fails in a planned manner, and small batch sizes deliver with less work in progress and faster detection of errors [31].

With the support of DevOps applied to ML, data scientists can test new approaches and perform experiments around the different segments of the pipeline, while automatically deploying new models and comparing results. To fully take advantage of automated learning algorithms, it is necessary that the infrastructure that supports the execution of ML pipelines should be automated in such a way that the development and operations activities can be fully merged into a continuous process. This approach enables the monitoring of problematic areas and the identification of bottlenecks, enabling continuous improvement of results. Promoting the observability of the pipeline execution through a processing engine where it is possible to measure performance, cost, and accuracy consistently is necessary.

Methodologies such as CRISP-DM can be established as the standard process model for

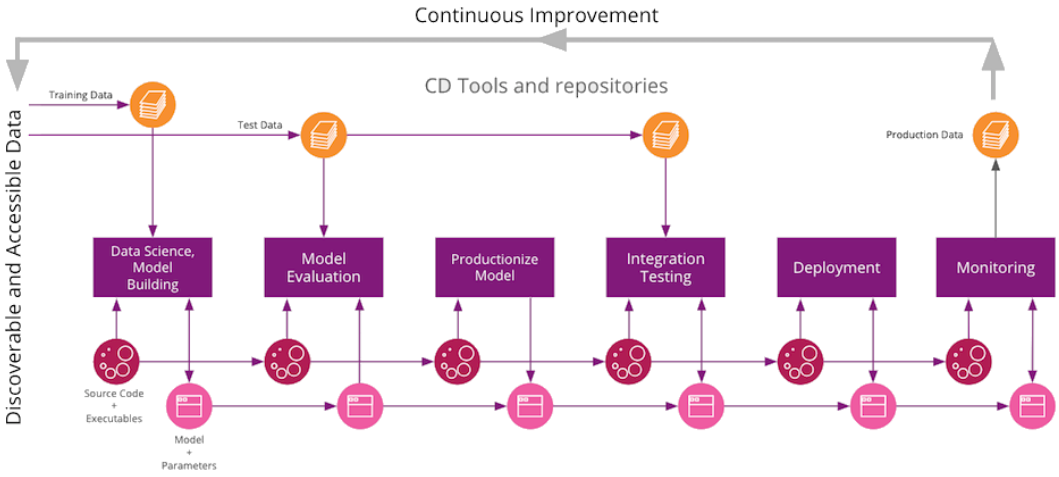

Figure 2.2: Continuous Delivery for Machine Learning (reproduced from [41]).

machine learning development. This process usually encompasses the following steps: 1) Data transportation, 2) Data Transformation, 3) Continuous ML (re) training, 4) Continuous ML (Re) Deployment, (5) Output production/presentation to the end user [46]. One known approach to MLOps is the implementation proposed by ThoughtWorks [41]. A cross-functional team produces machine learning applications based on code, data, and models in small, safe increments that can be reproduced and reliably released anytime in short adaptation cycles.

The workflow starts when data scientists have access to data. They will then perform several pre-processing data steps until data can be split for training and testing purposes. Model selection, experimentation with hyperparameters, evaluation, and comparison will occur until a good model is found. Automated testing will then run on the model. If tests pass, the model should be adapted to the production environment, and new integration tests should be performed. The model is packaged, wrapped, and integrated into a deployable artifact. At this stage, the ML model is ready to migrate from the testing environment to the production environment. Once the model is successfully integrated and deployed, its performance or potential biases should be constantly evaluated through monitoring and redeployed through the practices of CI and CD.

2.1.3 The MLOps lifecycle

MLOps is the standardization and streamlining of machine learning life cycle management [47]. It is composed of multiple integrated processes that work iteratively, being data and model management the basis to enable this structure. Figure 2.3 shows how these processes are integrated.

ML Development The experimental nature of the model development process produces multiple artifacts in each model iteration. For this reason, experiment versioning and management is essential in MLOps architecture. Whenever the data scientist thinks an experiment is showing signals of promising results, it can be interesting to save this information. Experiment management helps select feasible models to be deployed in a production environment since it enables the

Figure 2.3: MLOps Lifecycle (reproduced from [1])

experiment's metrics comparison. Previously saved experiments can also be later used as an initial point to try different experiments. Every experiment training run generates a set of metadata information, such as:

- **E**nvironment configurations versions
- **S**ource code versions
- **D**ata versions
- **M**etrics
- **H**yperparameters configurations

Figure 2.4 shows a set of artifacts that are frequently produced throughout the experiment runs. Examples include datasets, models, environment resources (Docker containers, Conda environment), software resources (notebooks, scripts, and source code in general), execution data (results and execution metadata), and metadata (experiment tracking and environment dependencies such as environment variables, python libraries and their versions, and hardware details). All these artifacts need to be simultaneously versioned. Tracking and managing this metadata is necessary to achieve traceability. Experiment tracking tools are essential to ensure appropriate version management. These tools are expected to provide information on the data, feature selection, model parameters, and performance metrics related to an experiment.

Data and Model Management One of MLOps goals is to reuse and provide high-quality data to the different phases of MLOps environments. Having a feature and dataset unified repository helps

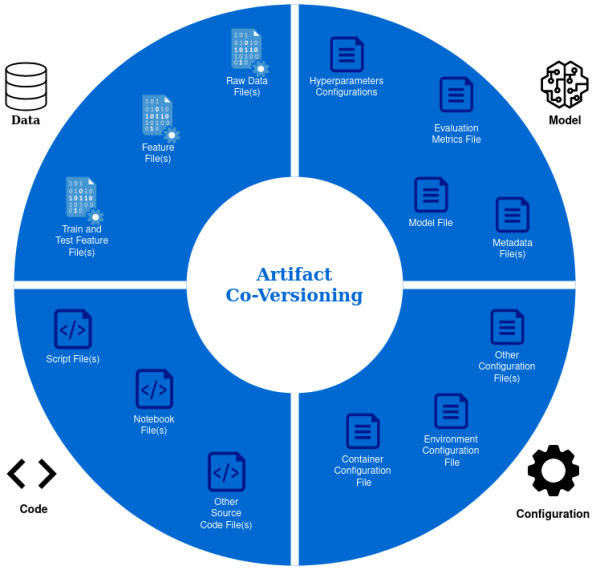

Figure 2.4: Artifact Co-Versioning.

to accomplish data management, reducing wasted time on data exploration and helping to prevent data inconsistencies. Once the model version is generated, it should be stored in a convenient place to be possibly used.

Training Operationalization In this phase, the process of packaging, testing, and deploying training pipelines is automated. For this purpose, a CI/CD pipeline is set up to re-create the model, ensuring that all tests and checks have been met. The main goals are early detection of bugs or conflicts in development code and enabling better collaboration between contributors. Several transformations can be performed on the model to make it compatible with the production environment. If that is the case, the validation must be done in an environment as close to the production environment as possible. In the context of CI/CD, an ML artifact is a testable and deployable project bundle that must be built [47]. Different elements are part of that bundle, such as code, hyperparameters and configurations, training and validation data, trained model, environment variables, and libraries.

Continuous Training Continuous training concerns the re-execution of the ML training pipeline in a similar matter of regular ML engineering process, with key differences:

- **D**ata is extracted according to the most recent data update.
- **D**ata and model validation are essential. Predefined metrics and checks should be included.
- **M**odel is stored in the model registry.

Retraining is needed for models that produce a non-deterministic output since model degradation can emerge over time, either due to data drift – changes in data distribution of the environment

– or concept drift – changes in the business requirements [36]. Another aspect that should be considered when retraining models is: 1. how accurate the model needs to be ;2. how easy it is to build and deploy a better model [47]. There can be many paths to refining a model. It can be retraining the model with the latest data, adding new features, or developing new algorithms. The next step is to evaluate the metrics and compare several candidate models trained with the latest data with the deployed model using the same development dataset.

Model Deployment In the beginning, applications were deployed by operations teams responsible for managing expensive and heavy-weight application servers. Once the management of physical machines was replaced by virtual machines running on the cloud, companies experienced cost reductions, and much of the complexity related to managing or acquiring computational resources were abstracted to third-party service providers.

The need for agility, scalability, and effectiveness in developing and deploying software applications led to the advance of microservices architecture over monolithic design. In this context, loosely coupled services can be deployed independently, and a deployment based on containerization became a popular option over virtualized development based on virtual machines. Clouds use virtualization mechanisms to provide abstraction and encapsulation features. It also enables increased application availability and improved responsiveness, enabling the automation of resource provisioning, monitoring, and maintenance while allowing resources to be cached and reused [21].

Under these circumstances, containerization strategies began to gain popularity. A *container* is a virtualization mechanism that works on the operating system level and consists of one or more processes running in isolation. While virtual machines virtualize physical hardware at the operating system level, containers virtualize the operating system level itself [30]. Each container is an instance of a service and has a fixed amount of CPU and memory resources that can be specified upon creation. Containerization technologies help to ensure consistency in dependency versions across different environments due to their encapsulation capacity. This feature is important since the new model version will run on a target environment in this phase. Usually, the model pipeline output consists in a set of code plus data artifacts, which can have version dependencies on runtimes. This environment should reflect the development environment, including the versions of software and operating systems in use, since a mismatch can cause differences in model predictions [47].

Prediction Serving It concerns serving the deployed model for inference. The model starts to accept prediction requests and to serve responses with predictions. A data feedback loop is usually needed to feed the development environment for future model improvements. Prediction logs can also be stored for continuous monitoring related tasks.

Continuous Monitoring There are risks inherently involved in using ML models. A major goal is to mitigate those risks by continuously monitoring a model's effectiveness and efficiency in production. One way to detect if a model is performing well is by accessing and measuring the

Key Performance Indicators (KPIs). These reflect business objectives translated into performance targets, considering the technical infrastructure requirements and cost constraints [47]. Another way is applying maintenance measures. These include resource monitoring related to IT metrics, such as CPU, memory, and network usage. On top of that, a health check mechanism capable of querying the model in production at a fixed interval and logging its results can also be implemented to monitor the model's latency and availability.

Continuous monitoring should also have a mechanism to detect model degradation. Input drift monitoring compares if the results from recent requests to the deployed model significantly differ from those captured in the training data. If that is the case, then there is a high probability that the model will not accurately reflect the newer data. In this scenario, reactive monitoring can be implemented through an automated pipeline that triggers warnings whenever model performance degradation is detected. This mechanism can be used to notify the data scientist of the new data changes that can trigger retraining or rollout strategies. Another fundamental goal of continuous monitoring is to capture the emerging patterns in new data to promote a data feedback loop, feeding the model development environment with information collected in the production environment to further model improvement.

2.2 Systematic Literature Review

The SLR describes the state-of-the-art for MLOps concerning version management issues. First, the methodology is presented in Section 2.2.1. In Section 2.2.2, results are presented and analysed, followed by a discussion of the main findings and the related work (Section 2.2.3).

2.2.1 Methodology

This section highlights the method in which search engines were queried with the relevant terms to answer each research question, describing how results were filtered using an iterative strategy.

2.2.1.1 Research Method

This study aims to provide an overview of the studies conducted in the realm of MLOps, focusing on versioning issues. More specifically, the main objective is to investigate practices, techniques, tools, and challenges related to the application of artifact versioning in MLOps workflows. The following research questions were developed to achieve this objective:

- **RQ1**: How is versioning applied to machine learning artifacts?
- **RQ2**: Which tools apply versioning?
- **RQ3**: What are the open challenges in this field?

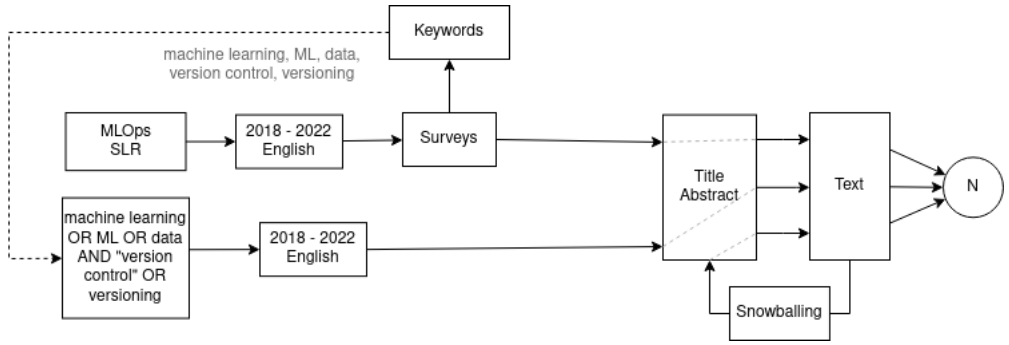

Figure 2.5: Search Process.

Research Process The search results show that MLOps has recently gained traction in research papers. There are a few references dating from before 2018. The first objective was to acquire an overview of this topic. Search strings were composed to access surveys and systematic literature reviews. Keywords were chosen from these surveys, and the research scope was reduced to issues surrounding version control and machine learning versioning. Google Scholar and Scopus were the research databases chosen to conduct this review. These searches were conducted using the title, abstract, and keywords scope. During this search, inclusion/exclusion criteria were applied to filter the studies explicitly addressing the specified topics. Also, duplicate citations were disregarded. The search process is shown in Figure 2.5.

Application of inclusion and exclusion criteria The results were refined and restricted to the period between 2018 and 2022. On Scopus, the subject area was restricted to Computer Science. Regarding the document type, the result was restricted to conference papers, articles, books, and journal papers written in English.

Abstract and text screen criteria For each remaining paper, the following criteria were applied:

- Does the context match with DevOps applied to ML pipelines?
- Does the paper mention versioning issues?
- Does the content address at least one research question?

Final pool of the primary studies Results from both search engines were later screened through a brief review of the title and abstracts to ensure that at least one research question was addressed. The search process ended when it was identified that the last ten results per page were unrelated to this research's scope. Based on the findings, a shortlist with 53 studies was selected. Then a final selection was made considering the entire article to ensure the article was interesting for this study. The snowballing technique was later conducted as a search strategy for identifying important articles related to these topics. On the final pool, the shortlist was restricted to 30 papers.

2.2.2 Result Analysis

2.2.2.1 RQ1: How is versioning applied to machine learning artifacts?

Machine learning systems have particularities compared to other types of systems. The relationship between AI and non-AI components can involve dependencies, and the architecture of such systems requires the implementation of the multi-level co-versioning principle. This concept encompasses not only the management of multiple versions of AI and non-AI components but also the management of artifacts within an AI component, like different versions of data, model, code, and configurations in such a way that traceability from data to model to ML component can be accomplished [33]. Different artifacts can be versioned throughout the execution of machine learning pipelines. Properly managing these artifacts requires architectural thinking in developing model pipelines to support co-versioning with version control and storage tools. The following subsections summarize the best versioning practices for data, model and configurations.

Data Versioning Proper artifact management requires tools that enable version control. While there are very well-designed technologies to version code, the same is not true for data [18]. Traditionally version control systems like Git are not suited for large files [19, 49, 26] or ML projects since it does not allow tracking of changes in data, hyper-parameters, model artifacts, or model dependencies [28]. Some tools extend git features and adopt git-style alike approaches by tracking immutable files like Data Version Control (DVC) [8]. Others adopt a snapshot-based approach by manually creating snapshots of each individual artifact [42] while time-difference-based approaches can also be an option [40, 38]. ML versioning pattern states that model structure, code, training dataset, and training system must be versioned preferentially in a two-layer mechanism, with code and metadata stored in code storage and large datasets and models stored in data storage [50]. The input data should ideally be archived in immutable storage. If that is not the case, then recoding the origin of the data with any meta information required to reconstruct its historical record is necessary. To this end, hashes of the original training dataset can be stored and version controlled along with the method for calculating them [19]. The actual dataset is usually stored in local data repositories, while remote solutions store only a hash of the current version due to the massive amount of data [40]. Features should also be immutable and versioned. They can be registered as a central repository for commonly used features in a feature store. Later, they can be used for model development or inference when deploying a model. In case of a bug fix, a new feature can be created instead of updated [43]. The feature system can often have an online database to serve features with low latency and an offline database to serve features with normal latency [32].

Model Versioning Model development should start as soon as that is enough data to learn. Containerization can create reusable, composable, and potentially shareable pipeline components [22]. Once the model workflow is done, the pipelines can be tested and integrated into the beginning of the model experimentation process. Models should be immutable, and any update should generate

a new version. Versioning is essential not only for model identification but also for model dependency tracking and management. Comprehending the dependencies between models is necessary to understand how a model impacts the entire production environment [44]. Model parameters, model source code, and model artifacts should also be automatically versioned after each successful training and during each rollout. After model experimentation, the code must be versioned and stored in a model repository to track the models used in production. Trained ML model versions should be stored in a model registry along with their metadata. This component acts like a central repository that can be accessed by the CI/CD pipeline to access model candidates. A metadata store is another component that contains all the information about the models, including information from all the training job iterations [32].

Configuration Versioning Containers can act as a snapshot of the software environment. Alternatively, if that is not possible, then the versions of libraries, utilities, and software components should be saved in the environment [43].

Pipeline Versioning Pipeline versioning is handy since controlling the compatibility between its stages is usually necessary. A pipeline can have multiple stages, and any change in a particular stage must be communicated to downstream consumers. Also, external dependencies (imported features) and internal dependencies can be a significant sources of incompatibilities [48].

2.2.2.2 RQ2: Which tools are used to apply versioning?

Open-source tools are more modular, offer a transparent workflow, and generally offer higher quality compared to commercial platforms [40]. Versioning tools have evolved, and nowadays, the majority of them offer experiment tracking. Before this, practitioners had to rely on informal methods, such as emails, spreadsheets, and notes, to perform this task [24]. Current tools can even offer features for roll-back operations and branching from an old dataset version to explore changes based on that version. Despite these benefits, ML versioning tools can add overhead to developers during maintenance tasks due to high coupling between ML-related and other software artifacts [20]. Table 2.6 indicates the tools most often appeared in the literature review. Some provide versioning support for data and/or model pipelines that can manage experiments but do not make the version management of other artifacts possible.

2.2.2.3 RQ3: What are the open challenges in this field?

End-to-end versioning support Tackling the reproducibility problem requires a holistic versioning approach over the entire machine learning pipeline capable of supporting the management of features, datasets, models, code, and information concerning software environments. Open-source tools generally assist individual tasks in a machine learning pipeline [43]. Even though many tools provide some level of experiment tracking, the same cannot be said when automatically promoting lineage between data, models, and code.

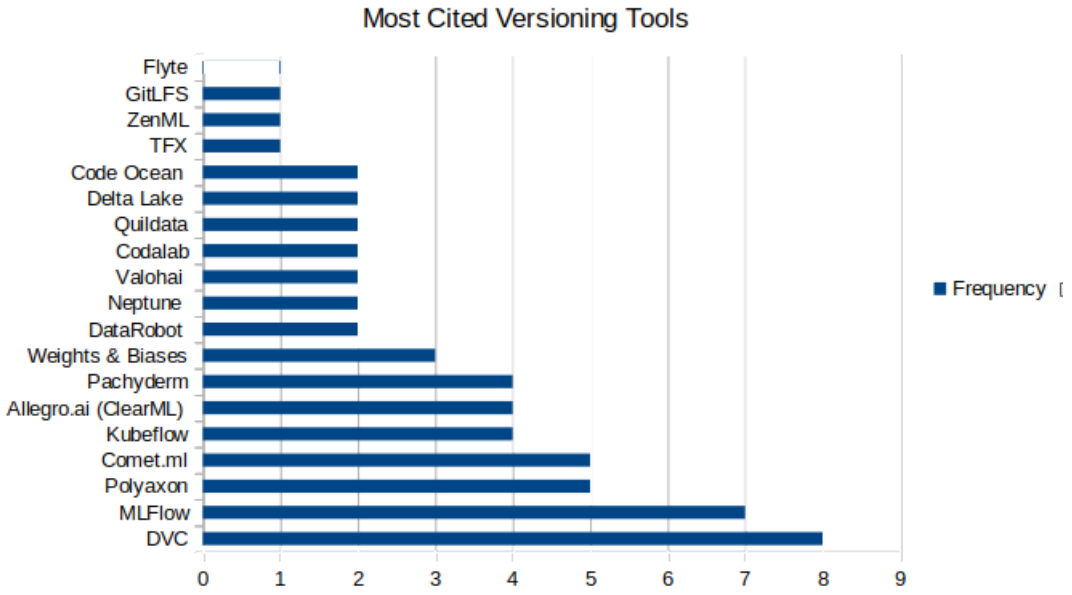

Figure 2.6: Versioning Tools.

Data Management Integration Versioning data may entail full provenance of data [39]. If data is not treated holistically, then data silos are created. This situation causes problems with 1) data dependency and 2) restricted ability to provide data lineage and provenance. The first point concerns dealing with different data access interfaces and layouts required in different tools throughout the ML workflow. The second point is a consequence of the first one. It is also essential that tools provide data derivation history – providing a chain of information that shows where the information came from and how it was transformed throughout the data pipeline. Raw data assets and the derived annotations and features should be mapped to identify all dependent and derived data in case of an error on the source dataset. [17].

Integration between ML Frameworks and Tools To achieve full versioning support, it is often necessary to blend multiple tools with specific domain abstractions with ML workflows [18]. This situation may lead to overlapping features, redundancy, and users having to manually track the dependencies between data, training tasks, models, and environments. Often different frameworks have different data access interfaces, with asset formats that could be more interoperable. Other limitations include limited interoperability across different tools, lack of explicit representation of domain knowledge, problems in framework integration, tight coupling with specific libraries, friction, and overhead [26, 37]. When choosing the toolset, the goal is generally to find a balance between flexibility and compatibility [45].

Integration between version control and asset management Git-based version control tools are often closer to the perspective of software engineers, whereas asset management tools are more tailored to data scientists. Thus, merging these perspectives in improved tools and making

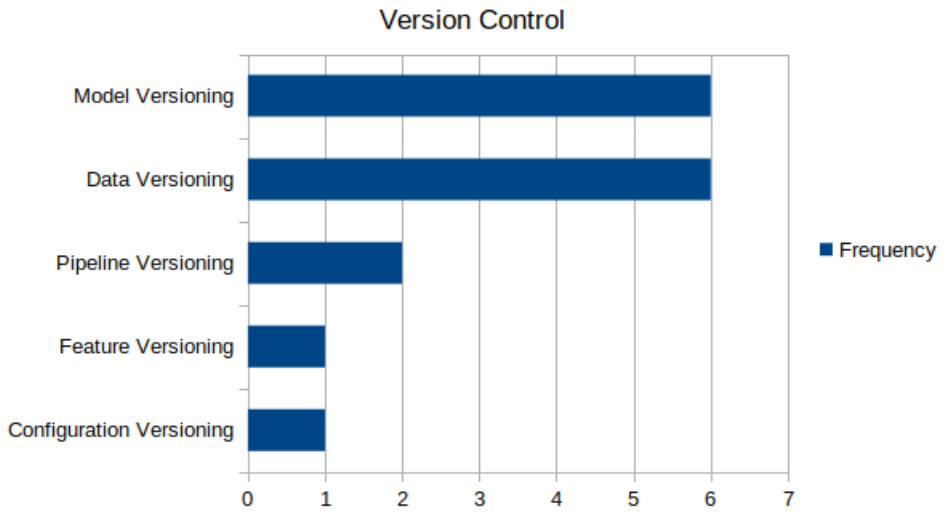

Figure 2.7: Studies segmented by versioning categories.

version control and asset management interoperable operations can be a path to aid tracking and reproducibility of machine learning models [27].

2.2.3 Discussion

The term MLOps began to attract more popularity in 2018 in the research community. Many studies have focused on architectural design decisions, components, activities, and challenges. However, there is still room to advance regarding specific guidance implementation or instructions regarding managing ML artifacts with a focus on versioning issues. The need for ML artifact management is mentioned as prerequisites to attain model reproducibility. However, most studies do not provide codified architectural knowledge, techniques, patterns, or documentation surrounding artifact co-versioning. Regarding the topic distribution, MLOps architecture is the most frequent topic among the final pool of papers. While many papers indicate challenges and best practices for architecting machine learning systems, less attention has been given to ML Design Patterns. Most studies focus on data and model artifact versioning, while pipeline and configuration versioning issues are less mentioned, as shown in Figure 2.7.

Regarding the related work, some papers focus on identifying workflow activities [29], architectural design decisions [49], or principles, components, operations, and roles of ML systems [32, 45]. Amershi *et al.* [18] observe engineering challenges, best practices, and particularities in the AI domain that differentiates it from the software engineering domain, while Lewis *et al.* [33] reveals software architectural challenges for ML systems. DevOps application to machine learning is discussed in [22, 40], including a discussion over tools and techniques. A machine learning model lifecycle management system developed by Uber named Gallery is described in [44]. Zhou *et al.* [52] propose a functional ML platform to explore resource consumption and performance bottlenecks issues, and Hummer *et al.* [25] propose a framework for the lifecycle management of AI applications. Washizaki *et al.* and Qinghua *et al.* [50, 34] identify design patterns applied

to machine learning workflow, while other papers provide tools for revision and comparison to advise practitioners better when developing ML pipelines [39, 23, 37, 28].

Pulkit *et al.* [17] introduces a data system for machine learning that threats data with a holistic perspective by acting as a data interface providing a data model, data access interface, and data life cycle management with the capability to track dependencies among data, models, and training tasks. The machine learning lifecycle is addressed in [51, 23, 42] with emphasis on topics related to lineage [26, 27, 17, 44] and coupling between artifacts [20] where the need to version artifacts is commonly related to provenance goals. On this matter, other papers also address the need to version data, models, configurations, or pipelines [19, 49, 18, 43, 40, 17, 36, 44, 48].

2.3 Summary

This chapter began with an introduction on the main concepts underlying machine learning operations, focusing on Machine Learning and DevOps fields, showing how decisions around these topics can influence the quality of MLOps solutions. Then a known architecture presented how machine learning systems components can work together in an automated workflow, followed by a description of the MLOps lifecycle. Next, the results from a SLR focused on machine learning version control and versioning issues identified a shortlist of 30 papers. The result analysis surrounding how versioning is applied to machine learning artifacts was segmented into artifact groups to indicate best practices for each group. An analysis of the most referenced tools is presented, followed by an indication of the main open challenges in this field. A discussion of the research results is finally presented, summarizing the related work.

Chapter 3

Problem Definition

This chapter describes the scope, main hypothesis, goals, and methodology. Then, the Intrusion Detection case study is presented. Detailed information concerning the design, architecture, and underlying implementation is also provided.

3.1 Scope

The case study implementation starts with data preparation considering that a dataset is already provided. Thus, a set of operations around data management, such as data searching, collection, and extraction, is assumed to be already done.

3.2 Main Hypothesis

The following hypothesis guides this work:

> Existing tools enable the development and operationalization of an ML workflow focusing on data, models, code, and configuration versioning, providing appropriate tracking and lineage between these different components.

3.3 Goals

To investigate this hypothesis, we will implement a workflow that supports ML engineering and operationalization, applying MLOps principles that facilitate model reproducibility.

According to [47]:

> True reproducibility requires version control of all the assets and parameters involved, including the data used to train and evaluate the model, as well as a record of the software environment.

For this work, *model reproducibility* means the ability to recreate the environment in which a deployed model was developed. This goal entails the following subgoals:

- **S**egmentation and automation of the model development pipeline.
- **I**ntegration and reproduction of machine learning experiments.
- **V**ersion control of machine learning artifacts.
- **P**rovision of means to reach lineage between code, data, models, and configurations.
- **F**acilitation of model debugging in the production environment.

3.4 Methodology

This dissertation aims to explore tools to construct an ML workflow capable of assisting in common tasks related to the machine learning life cycle, such as data and model management, continuous monitoring, deployment automation, and continuous training by focusing on artifact versioning and management with the ultimate goal to assist in tasks related to model debugging in production and model traceability. The workflow will address a case study on Intrusion Detection (Section 3.5), initially proposed in the Data Analysis and Software Engineering course within the Master in Software Engineering program at the Faculty of Engineering, University of Porto. The project's primary goal was to develop an intrusion detector that could predict whether network access is malicious. Requirements (Section 4.1) were developed to validate the machine learning system implementation. In Section 4.4, we will simulate several scenarios, covering data preparation, model generation, comparison between model versions, deployment, monitoring, debugging, and re-training to assess how the selected frameworks and tools could be integrated to provide artifact versioning and management.

3.5 Intrusion Detection Case Study

Intrusion detection systems can take advantage of the constant amount of data being produced in order to use them to train machine learning models. These models can detect trends and patterns in historical data and estimate the probability of identifying future accesses as fraudulent. The data for development consists of network accesses with 41 predictive attributes and a binary target attribute. The project development entails analyzing network traffic data to build models that should be able to perform predictions with high accuracy. This work will extend that problem on two fronts: The first is related to MLOps infrastructure provision, including continuous automated integration, delivery, training, deployment, and monitoring capacities. The second is the integration of existing tools that can support artifact co-versioning.

3.5.1 Case Study Questions

General questions regarding implementing MLOps capabilities to an ML workflow were raised during the case study design phase. These questions are not directly related to versioning capabilities. Instead, they are related to building the necessary infrastructure to support the model pipeline

automation. The scenarios developed in Chapter 4 will show how these questions can be addressed throughout the case study usage workflow.

- **CSQ1:** How to automate the model development pipeline?
- **CSQ2:** How to compare models?
- **CSQ3:** How to enable continuous integration and continuous delivery?
- **CSQ4:** How to deploy an ML model in a production environment?
- **CSQ5:** How to enable continuous monitoring?
- **CSQ6:** How to reproduce model experiments?
- **CSQ7:** How to enable continuous training?

3.5.2 Architecture

The proposed architecture encompasses four processes: the ML engineering process (Section 3.5.2.1), the operationalization process (Section 3.5.2.2), the prediction servicing process (Section 3.5.2.3), and the retraining process (Section 3.5.2.5). Each relies on specific services, components, and workflows to compose the system.

3.5.2.1 ML Engineering Process

A **Dataset Repository** contains curated data previously generated in the data engineering phase. This component is the primary source of development data and provides a mechanism to unify data storage while enabling data versioning. The **Code Repository** contains the source code and configurations, such as libraries versions. In this project, the dataset repository is represented by an AWS S3 bucket and stores a dataset from previous network accesses, while the code repository is hosted on GitHub. Model experimentation starts with the problem definition. Then, data discovery, selection, and exploration take place. As part of the data understanding phase, numerical and categorical analysis was conducted with visualization of variables, along with statistical and correlation analysis. The data was then processed through attribute selection and normalization in the data processing stage.

The model prototyping stage involves several experiments around algorithm selection and hyperparameter tuning, followed by model training and evaluation. Figure 3.1 describes how the experiment tracking process is integrated into the proposed architecture. Considering that an experiment is an iteration of ML model development and its versioning is a central piece to enable model reproducibility, this project was configured with DVCLive [11]. This library enables experiment management by allowing experiment auto-tracking, preserving a connection to the latest commit in the current branch. After experiments iterations and metrics comparison, once the experiment is considered the best, it is persisted as a new commit in a new branch using regular git workflows, and a pull request is created.

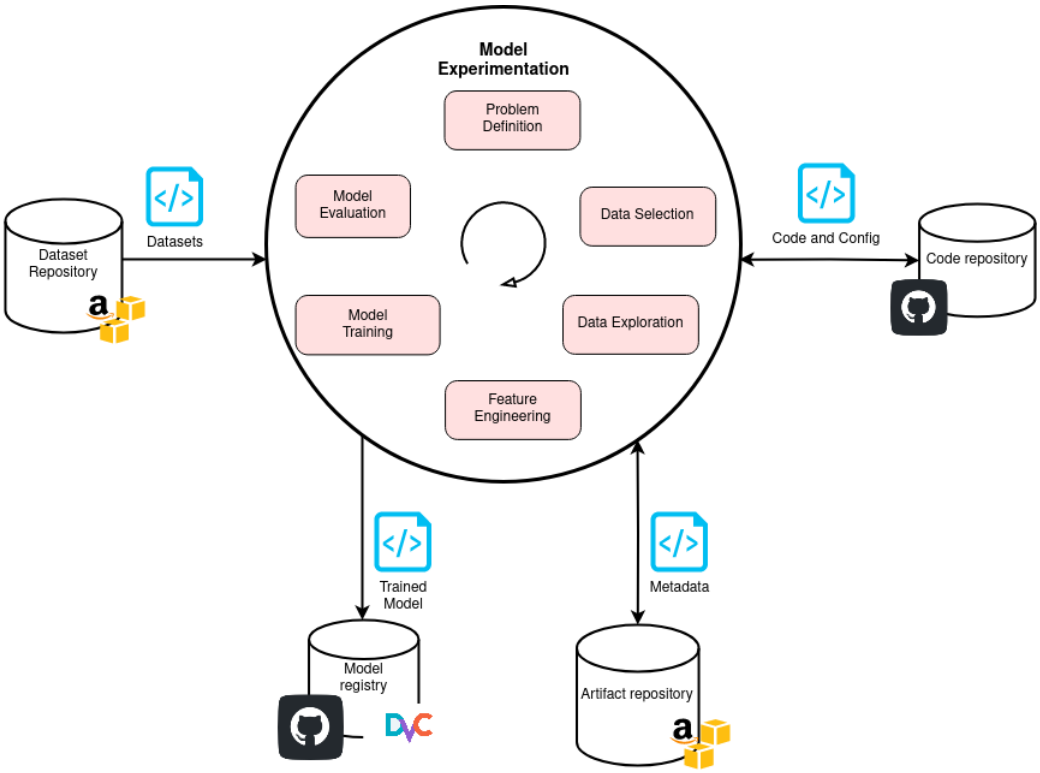

Figure 3.1: ML Engineering Process (Adapted from [1]).

Experiment runs produce a diverse set of ML artifacts. Information about these artifacts is what is called ML metadata. This data is foundational to providing traceability and lineage capabilities. This information is stored in the **Artifact Repository**, represented as a bucket in AWS S3, and can be regularly pulled when there is a need to reproduce experiments. Experimentation activities usually generate processes that contribute to refining the problem definition, often producing new features and datasets. These outputs become new requirements for the data engineering phase that constantly produces new curated data.

When a model is considered a good candidate to make it to the production environment, the output of the ML engineering process is a trained model. In that case, the model is pushed to a **Model Registry**. This component acts like an interface between the ML engineering process and ML operationalization process as it exposes a collection of ML model versions along with their metadata enabling it to be discovered, tested, shared, deployed, and audited. One way to create a Git-native registry on top of DVC is using Git Tag Ops (GTO). This library tags ML models releases and promotions and links them to artifacts in the repository using versioned annotations [2].

3.5.2.2 ML Operationalization Process

There are two main workflows regarding the DevOps foundation. The first one is responsible for implementing **continuous integration (CI)**, which entails building, running, and testing the

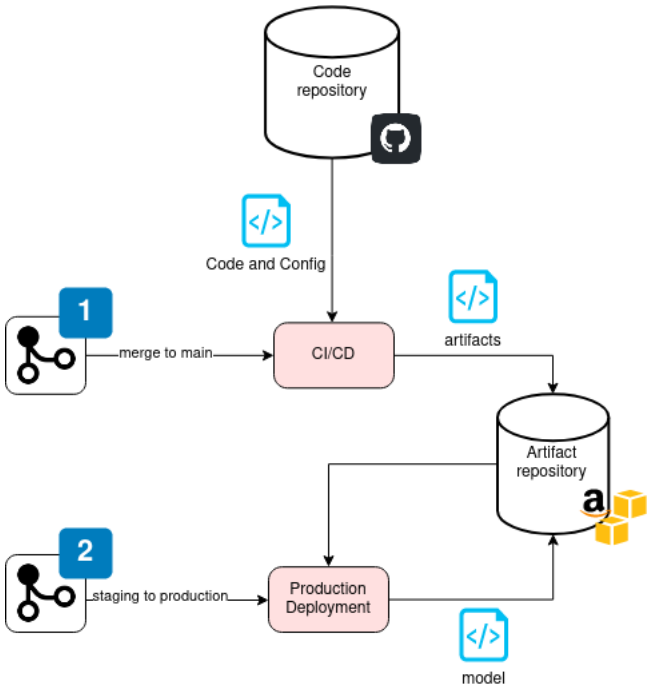

Figure 3.2: ML Operationalization Process.

training pipeline, and **continuous delivery (CD)**, which entails packaging, and validating the pipeline. This workflow will be triggered whenever a merge derived from a merge request to the main branch is accepted. The second is the deployment workflow. It entails deploying the trained pipeline to a target environment. This automation will be executed whenever a model version is assigned to the production environment.

The CI/CD workflow will check out the code of the branch, then will pull the artifacts from the ML artifact repository and automatically execute the pipeline. If there are any changes, the generated outputs will be pushed to the artifact repository, and the changes will be committed and pushed to the code repository.

The **deployment** workflow will check out the code of the tag, then will pull the model file from the artifact repository, create a file, and save the tag of the model into this file so the model can be discovered in the future using this information, then upload the model and the tag file to a bucket located on production environment. This process is illustrated in Figure 3.2.

3.5.2.3 Prediction Servicing Process

The prediction serving process is illustrated in Figure 3.3 and starts after the model is deployed in the production environment. A REST API intercepts a prediction request that downloads the latest model from the Artifact Registry and returns the prediction as an HTTP response. Servicing metrics are essential to show how the service is performing. Figure 3.4 show how metrics are served. The API is integrated with a system monitoring tool that can scrape metrics, such as

request counts and CPU usage. On top of that, the API also returns a custom metric, a counter that increments the result of every prediction.

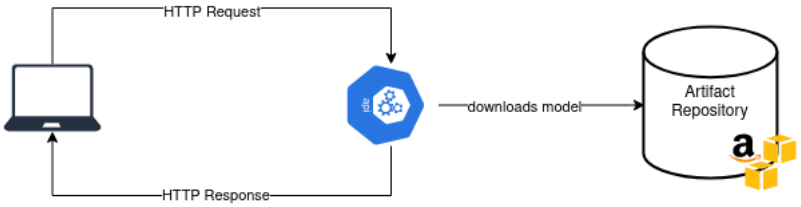

Figure 3.3: Prediction Servicing Process.

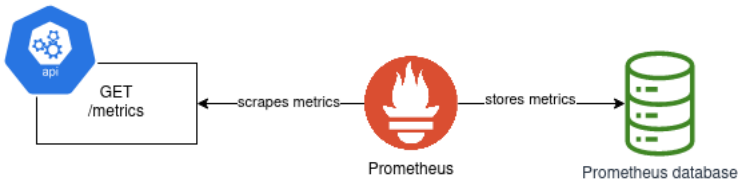

Figure 3.4: Servicing Metrics.

3.5.2.4 Continuous Monitoring Process

A panel for each metric enables performance monitoring. Figure 3.5 shows the visualization of a custom metric that shows the intrusion prediction results. Other standard metrics, such as CPU and memory usage, can also be visualized.

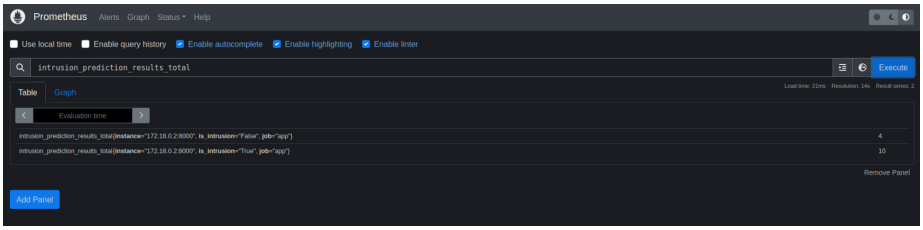

Figure 3.5: Metrics monitoring.

3.5.2.5 ML Continuous Training Process

It consists of retraining an existing model with the latest training data to refine the existing model. For this project, the original dataset containing 300k lines was split in half, originating two datasets containing 150k lines each. The first dataset was used for generating model versions, and the second was used to retrain the model[1]. Figure 3.6 shows the retraining workflow. After being manually triggered, the latest data will be pushed from the data repository to re-execute the model pipeline. The changes are then committed to a new branch in the code repository for further comparison.

[1]More information is available on A.2

Figure 3.6: Re-training workflow.

3.6 Summary

This chapter discussed how issues such as data quality, continuous monitoring, artifact versioning management, integration between ML frameworks, and hidden technical debt could contribute to increasing the complexity and challenges in designing and implementing machine learning systems. Next, in the methodology section, a presentation of the Intrusion Detection case study is followed by a description of the validation strategy and the scope indication. Then the case study questions and architecture are presented.

Chapter 4

Results

In this chapter, the Intrusion Detection Pipeline implemented in the previous chapter will be validated against a set of requirements and scenarios. The scenarios should be followed sequentially. Each contains a description and indicates the required tools, tasks, and commands to follow its development. A brief contextualization of the main tools used to develop the scenarios is also given. Finally, a discussion of the gains and limitations of the project is presented.

4.1 Requirements

The following case study was simulated to contextualize the applicability of MLOps infrastructure better to support AI systems in real-world settings.

A Case Study on Data Breach

The e-commerce company ACME Inc. has experienced a severe data breach in recent months. The company implemented an intrusion detection system to block malicious network accesses to mitigate further possible incidents. One key component of this system is a machine learning model that detects if network traffic is an intrusion attempt. This system needs to contemplate the following requirements:

- **T**he generated model should be able to capture at least 95% of the attacks with at most 10% of regular accesses incorrectly predicted.
- **A**s intrusion is a malicious activity, the model should be able to deal with variants of existing attacks or new types of attacks.

4.2 Tools

The following scenarios were built around four tools: DVC, GitHub Actions, Prometheus, and Docker. The first is used widely on the application since it controls version management aspects.

The second is the CI/CD implementation pillar, while the third aids in monitoring tasks. The forth is a containerization technology that enables the pipeline and API to be deployed as containers.

Data Version Control (DVC) This git-based data science open-source tool is used to apply version control to machine learning development by storing meta-information in Git while pushing the actual data to cloud storage [8]. The scenarios that will be described in the next section mainly take advantage of the following features:

- **E**xperiment Tracking and Management.
- **M**odel metrics comparison among experiments.
- **M**odel registry.
- **D**ata and model versioning.
- **P**ipeline versioning.

This tool can produce data, models, code, and configurations versioning while also providing experiment tracking. DVC works on top of Git repositories and has a similar workflow (commits, branching, pull requests). Figure 4.1 gives an overview of its general mechanism. Large files are stored in a remote data storage - Amazon S3 bucket in our case - while all configurations and code are versioned in a version control repository, like GitHub. DVC caches data and manipulates information as metadata files that work as a placeholder for the original data. These files' contents define the data file versions and can be easily versioned like source code with Git.

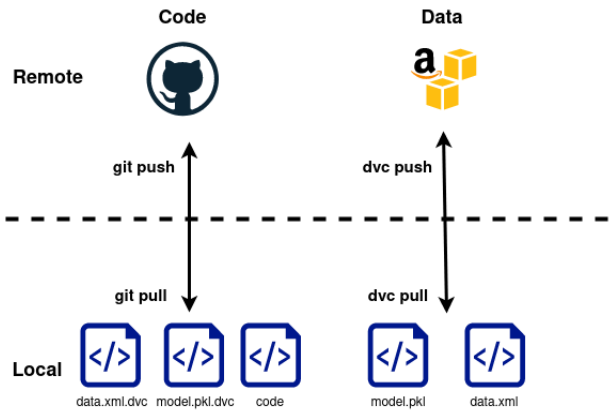

Figure 4.1: Version Control using DVC.

The pipeline versioning feature is essential in the following scenarios, as most of them create or modify stages on the pipeline. DVC pipelines allow better project organization and workflow reproduction. Processing steps are pipeline stages that connect code to its corresponding data input and output. Whenever running the pipeline, a state file called `dvc.lock` will be updated to reflect the reproduction's result and the current state of the pipeline. DVC will automatically track the data used and produced by any stage and a `dvc.yaml` file will be generated. This file represents the pipeline definition and lets one quickly reproduce it.

Github Actions GitHub Actions is a continuous integration and delivery platform that automates the build, test, and deployment pipeline processes using a configurable workflow triggered when an event occurs on the repository [15].

Prometheus Prometheus is an open-source system monitoring and alerting toolkit that collects and stores metrics as time series data [16]. On the Intrusion Detection Project, it scrapes the metrics exposed by the API, stores them, and allows for basic dashboards to be created.

Docker Docker provides tooling and a platform to manage the lifecycle of containers, allowing the application to be deployed as a container or an orchestrated service [9].

4.3 Setup

This setup describes the tasks required to set up the model pipeline and the API that instruments the prediction requests. To fully reproduce the subsequent scenarios, follow the guidelines provided on the respective read.me files to setup the Intrusion Detection Pipeline and Intrusion Detection API projects available on GitHub.

4.3.1 Intrusion Detection Pipeline

To adhere to the co-versioning principle, the Intrusion Detection Project was configured with DVC [8], Git [14], and Conda [6]. Git is a version control system while Conda is an open-source package management and environment management system that enables quick installation, running, and updating of packages and their dependencies. The project also requires the installation of AWS Command Line Interface [5] which helps the management of AWS services. In this project, we will configure two S3 buckets. DVC will use one bucket to store and retrieve data; the other will serve the latest production model. More detailed information on how to set up the environment variables is available in the project's readme file.

To help to standardize the workflow following software engineering practices such as modularization and parametrization, a DVC Pipeline was set up, illustrated in Figure 4.2. This feature allows different components of the model development workflow to be segmented as stages of a pipeline. Once the pipeline is reproduced, it generates meta files that capture the "state" of the pipeline, indicating, for example, which parameter values were used with which data version. This feature also helps in workflow automation and reproducibility as it indicates which commands will generate the pipeline results.

4.3.2 Intrusion Detection API

A REST API was built for prediction servicing using FastAPI, a modern, fast web framework for building APIs with Python [12]. For continuous monitoring, this project was configured with Prometheus, an open-source systems monitoring and alerting toolkit that collects and stores metrics as time series data [16]. The setup for running FastAPI and Prometheus containers was based

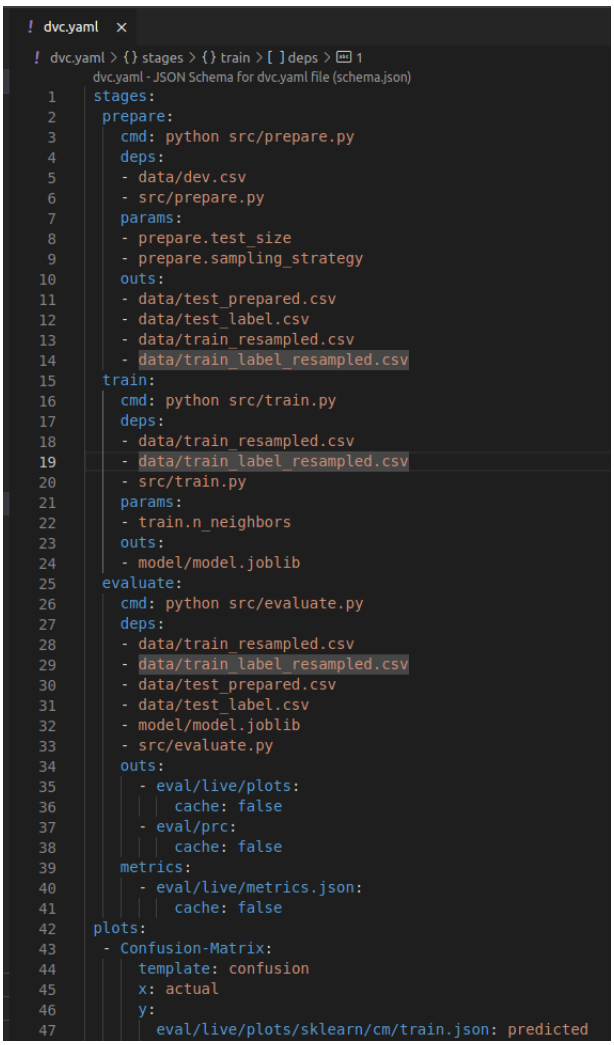

Figure 4.2: dvc.yaml file automatically generated by DVC

on a public repository [13] that only requires Docker [9] and Docker-compose [10] as prerequisites.

4.4 Scenarios

4.4.1 Preparing the data

Conduct algorithm-independent data preparation with the goal to replace missing values with mean.

Addressing this scenario entails performing the following tasks:

1. Create a Python script to replace missing values with mean.
2. Transform the script into a stage on the data pipeline.

The code to perform those tasks uses the following external library:

- **pandas** is a library for data analysis that organizes data in table-like structures.

Replace missing values As the original dataset did not contain missing values, we added some for support this scenario[1]. Then, the following script will loop on all columns, then it will use the column's mean as default value if the column name is in the list of target columns. Otherwise, it will replace the value with zero.

```
import pandas as pd
target_columns = ["count", "srv_count","src_bytes"]
dev = pd.read_csv("./data/dev.csv")
for column_name in list(dev):
  value = round(dev[column_name].mean()) if column_name in target_columns else 0
  dev[column_name].fillna(value=value, inplace=True)
dev.to_csv('./data/prepared', index=False)
```

Listing 4.1: Python Data Preparation Script for Scenario 4.4.1

Pipeline Stage The following command will transform a Python script into a stage.

```
dvc stage add -n prepare \
              -d prepare.py -d data/data.csv
              -o data/prepared \
              python src/prepare.py data/data.csv
```

The resulting pipeline can be seen on file **dvc.yaml**. The stage name **prepare** will be created. This stage depends on `data.csv` and `prepare.py` files to work and specifies an output directory for the script, which write a file in it. The last line is the command to run the stage.

[1] It is necessary to run a Python script to add missing values on the dataset. More information is available on A.1

```
stages:
  prepare:
    cmd: python src/prepare.py
    deps:
      - data/dev.csv
      - src/prepare.py
    outs:
      - data/prepared/dev.csv
```

Listing 4.2: dvc.yaml File

4.4.2 Generating the model

Develop a pipeline containing data preparation operators (feature selection and normalization), split the dataset into training and test sets, and generate a ML model using K-nearest neighbors algorithm.

Addressing this scenario entails performing the following tasks:

1. Data Setup
2. Data Preparation
3. Model Training with K-nearest neighbors algorithm.

The code to perform those tasks uses some external libraries:

- **pandas** is a library for data analysis that organizes data in table-like structures.
- **sklearn** is a library that provides diverse algorithms for classification, regression, clustering, and dimentionality reduction.
- **joblib** is a set of tools for lightweight pipelining in Python, providing easy simple parallel computing.

Data Preparation The following script will read from a csv file, perform data preparation operations, such as normalization, and will finally split the dataset into training(70%) and test(30%) sets.

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import dvc.api
dev = pd.read_csv("./data/dev.csv")
dev = dev.loc[:, (dev != dev.iloc[0]).any()]
x_dev = pd.get_dummies(dev).drop('class', axis=1)
y_dev = dev["class"]
```

```
x_dev = x_dev.drop('service_red_i', axis=1)
x_dev = x_dev.drop('service_pm_dump', axis=1)
x_dev = x_dev.drop('service_tftp_u', axis=1)

cols_dev = x_dev.columns
scaler = MinMaxScaler()
scaler.fit(x_dev)
x_dev = scaler.transform(x_dev)
x_dev = pd.DataFrame(x_dev, columns=cols_dev)

dev_train, dev_test, label_train, label_test = train_test_split(x_dev, y_dev,
    test_size=0.3, stratify=y_dev)

dev_test.to_csv('./data/test_prepared.csv', index=False)
label_test.to_csv('./data/test_label.csv', index=False)
dev_train.to_csv('./data/train_prepared.csv', index=False)
label_train.to_csv('./data/train_label.csv', index=False)
```

Listing 4.3: Python Data Preparation Script for Scenario 4.4.2

Model Training K-NN classifier will be used to train the model.

```
import os
import pandas as pd
from datetime import datetime
from sklearn.neighbors import KNeighborsClassifier
from joblib import dump
import dvc.api

dvc_params = dvc.api.params_show()
train_params=dvc_params["train"]
n_neighbors=train_params["n_neighbors"]

dev_train = pd.read_csv("./data/train_prepared.csv")
label_train = pd.read_csv("./data/train_label.csv")
label_train = label_train["class"]

neigh = KNeighborsClassifier(n_neighbors=n_neighbors)
neigh.fit(dev_train, label_train)

if not os.path.exists("./model"):
   os.makedirs("./model")

dump(neigh, './model/model.joblib')
```

Listing 4.4: Python Training Script for Scenario 4.4.2

The next step is to transform these scripts into dvc pipeline stages:

```
dvc stage add -n prepare \
              -d prepare.py -d data/data.csv
```

```
            -o data/test_prepared.csv data/test_label.csv data/train_prepared.csv data/
    train_label.csv \
            python src/prepare.py data/data.csv
dvc stage add -n train \
              -p train.n_neighbors \
              -d src/train.py -d data/train_prepared.csv data/train_label.csv \
              -o model/model.joblib \
              python src/train.py model.pkl
```

This command creates two stages named **prepare** and **train**. The first stage depends on `data.csv` and `prepare.py` files and specifies the outputs as csv files after applying data processing steps. The second stage takes a parameter named **n-neighbors** and produces a model as output. The case study question **CSQ1** in Section 3.5.1 questioned how the model pipeline could be automated. This step answers it by applying DVC's pipeline versioning feature to automate the preparation and modeling phases.

4.4.3 Creating multiple versions of the model

Given an algorithm, involves using grid search to fine-tune its hyperparameters with a separate validation set and storing all results for further analysis.

There are several ways of generating multiple models. In this case, we have used hyperparameter tuning as one way of doing it. Addressing this scenario entails performing the following tasks:

1. Setup configurations to enable grid search.
2. Apply grid search to an algorithm.
3. Store results.

The code to perform those tasks uses some external libraries:

- **hydra** is a framework built to compose and override configuration from the command line and configuration files.
- **dvclive** is a Python API module to log data.

Hydra Setup DVC supports Hydra's configuration composition as a way to configure experiment runs. The following command will enable this feature:

```
dvc config hydra.enabled True
```

When running an experiment, Hydra will compose a parameters file that will be used to configure the underlying DVC pipeline. First, we need a `conf/` directory for Hydra config groups.

```
    conf
    |-- config.yaml
    |-- prepare
        |-- default.yaml
    |-- train
        |-- default.yaml
```

The `conf/config.yaml` file defines the Hydra defaults list:

```
defaults:
  - train: "default"
  - prepare: "default"
```

Listing 4.5: Config.yaml File

The `prepare/default.yaml` and `train/default.yaml` files indicate the parameter values to be used in the experiments:

```
defaults:
  test_size: 0.3
```

Listing 4.6: Default.yaml File in Prepare Directory

```
defaults:
  n_neighbors: 9
```

Listing 4.7: Default.yaml File in Train Directory

Hyperparameter Tuning Grid search helps identity which hyperparameters perform best with a given algorithm since it automatically cycles through an exhaustive list of hyperparameter values, learns a model for each combination, evaluates it on a validation set and then chooses the combination with the best predictive performance. With DVC, a grid search can be performed by combining parameter modification on-the-fly with the support of Hydra's syntax for range sweeps to add multiple experiments to a queue. The following command will queue multiple experiments generated with parameter values within the given range of values.

```
dvc exp run --queue \
            -S 'prepare.test_size=0.3,0.25,0.2' \
            -S 'train.n_neighbors=9,10,11'
dvc queue start
```

At the end of the grid search process, a set of experiments will be generated.

```
(intrusion-detector-pipeline) myname@mydevice:~/Master/pipeline$ dvc queue status
Task     Name         Created    Status
3ed7d44  yucky-paws   04:19 PM   Running
4c8214a  nicer-maya   04:20 PM   Queued
17702f2  irate-daws   04:21 PM   Queued
26de329  curly-peri   04:21 PM   Queued
80c0135  rapid-sinh   04:21 PM   Queued
7036a1d  inert-wort   04:21 PM   Queued
ee458f7  buxom-yolk   04:21 PM   Queued
1eb6327  beery-woes   04:21 PM   Queued
87a5ddd  snide-sley   04:21 PM   Queued
1f21157  brisk-hour   04:21 PM   Queued
8abfdaf  treen-brig   03:45 PM   Success
e12d900  kacha-kifs   03:45 PM   Success
3d7e3db  undue-ball   03:45 PM   Success
48125d1  vying-snow   04:19 PM   Success
b9eef8c  scald-kine   04:19 PM   Success
```

Figure 4.3: Queued Experiments.

Storing Results With DVCLive' Live API every experiment is versioned and the results are stored under the dvclive directory.

```
eval
   |-- live
       |-- plots
           |-- cm #confusion matrix
               |-- test.json
               |-- train.json
           |-- roc #roc curve
               |-- test.json
               |-- train.json
       |-- metrics.json
   |-- prc #precision recall
       |-- test.json
       |-- train.json
```

4.4.4 Comparing experiments

Given multiple model versions involves comparing the results obtained with the best set of hyperparameters.

We decided to continue the example of hyperparameter tuning started in the previous section to compare the results of multiple models. Addressing this scenario entails performing the following tasks:

1. Review experiments.
2. Persist experiments.

Experiments Review DVC provides commands to list, tabulate, and contrast experiments. The following command will list the previous experiments along with their parent commit and name (or hash) and the generated metrics, parameters, and dependencies and drop any column with values that do not change across experiments. This step answers case study question **CSQ2** in Section 3.5.1.

```
dvc exp show --only-changed
```

Experiment	Created	avg_prec.test	roc_auc.test	prepare.test_size	train.n_neighbors	data/dev.csv	model/model.joblib	src/evaluate.py
workspace	-	0.99993	0.99984	0.3	9	6d31775	51f2f6a	7342410
model@v1.0.3	May 04, 2023	0.99996	0.99991	0.3	9	7e10d59	fba6ecb	614bfed
├── 01cf332 [brisk-hour]	Jun 09, 2023	0.99996	0.99991	0.2	11	6d31775	0d0a1f1	614bfed
├── 7b97f64 [snide-sley]	Jun 09, 2023	0.99996	0.99991	0.2	10	6d31775	3191635	614bfed
├── 7518c3d [beery-woes]	Jun 09, 2023	0.99996	0.99991	0.2	9	6d31775	5670d89	614bfed
├── 8225871 [buxom-yolk]	Jun 09, 2023	0.99993	0.99985	0.25	11	6d31775	2215091	614bfed
├── 654dd70 [inert-wort]	Jun 09, 2023	0.99992	0.99982	0.25	10	6d31775	695e2e4	614bfed
├── a03fcaf [rapid-sinh]	Jun 09, 2023	0.99992	0.99982	0.25	9	6d31775	43cec57	614bfed
├── 62f5fcd [curly-peri]	Jun 09, 2023	0.99994	0.99987	0.3	11	6d31775	adab269	614bfed
├── 4fa640d [irate-daws]	Jun 09, 2023	0.99993	0.99984	0.3	10	6d31775	3387282	614bfed
├── 607f0c9 [nicer-maya]	Jun 09, 2023	0.99993	0.99984	0.3	9	6d31775	51f2f6a	614bfed
├── 8f5f13a [yucky-paws]	Jun 09, 2023	0.99996	0.99991	0.2	9	6d31775	5670d89	614bfed
├── 62e07c5 [scald-kine]	Jun 09, 2023	0.99992	0.99982	0.25	9	6d31775	43cec57	614bfed
├── aad012c [vying-snow]	Jun 09, 2023	0.99993	0.99984	0.3	9	6d31775	51f2f6a	614bfed
├── 4b95861 [undue-ball]	Jun 09, 2023	0.99994	0.99987	0.3	11	6d31775	adab269	614bfed
├── c71dca9 [kacha-kifs]	Jun 09, 2023	0.99993	0.99984	0.3	10	6d31775	3387282	614bfed
├── fc50e08 [treen-brig]	Jun 09, 2023	0.99993	0.99984	0.3	9	6d31775	51f2f6a	614bfed
└── 78e100d [wiser-juts]	Jun 09, 2023	0.99993	0.99984	0.3	9	6d31775	51f2f6a	614bfed

Figure 4.4: Experiments displayed in a customizable table.

Persisting Experiments Successful experiments can be committed into DVC and git repositories. We may want to commit brisk-hour experiment into the DVC project and a new branch on the git repository. Once the best experiment is persisted as a new commit in a new branch using regular git workflows is time to create a pull request.

```
 dvc exp branch brisk-hour
 Git branch 'brisk-hour-branch' has been created from experiment 'brisk-hour'.
To switch to the new branch run:
        git checkout exp-brisk-hour
git add .
git checkout exp-brisk-hour
Switched to branch 'exp-brisk-hour'
# Persists the experiment on the git repo
git push origin exp-brisk-hour
# Persists the experiment on the dvc repo
dvc push
```

4.4.5 Deploying a model to production

It entails deploying a model into a prediction serving system.

Addressing this scenario entails performing the following tasks:

1. Register the model version.
2. Merge the pull request into the main branch.
3. Assign the model version to the production environment.

The code to perform those tasks uses the following external library:

- **gto** is a CLI tool that tags ML model releases and promotions and links them to artifacts in the repository using versioned annotations [2].

Model Registration **Git Tag Ops (gto)** helps to manage machine learning artifact versions in a Git repository along with their deployment stages. We want to register the model version originated from the experiment persisted above.

```
gto register brisk-hour
Created git tag 'brisk-hour@v0.1.14' that registers version
To push the changes upstream, run:
    git push origin brisk-hour@v0.1.14

gto show
```

This creates a Git tag attached to the latest Git commit (HEAD), automatically bumping the artifact's version.

Merge Pull Request After registering the model version, we will accept the pull request and merge it with the main branch. This will trigger the CI/CD workflow responsible for building and running the training pipeline, answering the case study question **CSQ3** in Section 3.5.1.

Model Assignment It is time to assign the model version to production environment.

```
gto assign brisk-hour --stage prod
Created git tag 'brisk-hour#dev#1' that assigns stage to version 'v0.1.14'
To push the changes upstream, run:
    git push origin brisk-hour#prod#1
```

This command creates a Git tag, which associates the latest version of brisk-hour to production. This action will trigger the deployment workflow which will upload the model version to a bucket located in the production environment. This step answers case study question **CSQ4** in Section 3.5.1.

4.4.6 Collecting metrics from the model

It entails monitoring model performance and prevision.

Addressing this scenario entails performing the following tasks:

1. Setup the REST API.
2. Create a prediction request.
3. Compute model metrics.

The code to perform those tasks uses some external libraries:

- **docker** is a platform for developing, shipping, and running applications [9]. To run it, it is necessary to configure Docker and Docker-Compose.
- **docker-compose** is a tool for defining and running multi-container Docker applications [10].
- **fastapi** is a web framework for building APIs based on standard Python type hints [12].

API Setup A Docker stack was developed to enable API implementation and monitoring. It comprises two services: the FastAPI container with the prediction API and the Prometheus container that scrapes the API for metrics and stores them.

```
1 version: "3.8"
2 services:
3   app:
4     build: .
5     container_name: app
```

```
    ports:
      - 8000:8000
    environment:
      - AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID}
      - AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY}
      - AWS_DEFAULT_REGION=${AWS_DEFAULT_REGION}
      - DEPLOYMENT_AWS_BUCKET=${DEPLOYMENT_AWS_BUCKET}
    # uncomment the volume setup to use the model file from the project
    # instead of downloading from S3
    volumes:
      - ./model/model.joblib:/tmp/model.joblib
  prometheus:
    image: prom/prometheus:v2.43.0
    restart: unless-stopped
    container_name: prometheus
    ports:
      - 9090:9090
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
    command:
      - "--config.file=/etc/prometheus/prometheus.yml"
```

The following command will start the API:

```
docker-compose up
```

Prediction Request The REST API intercepts a prediction request, downloading the latest model version from the artifact repository and returning the prediction as an HTTP response. To invoke the prediction API with an example of an intrusion, execute the following command:

```
curl -H "Content-type: application/json" -d '{"duration":0,"src_bytes
    ":0.00020094717776309186,"dst_bytes":0,"land":0,"wrong_fragment":0,"urgent":0,"hot":0,"
    num_failed_logins":0,"logged_in":0,"num_compromised":0,"root_shell":0,"su_attempted":0,"
    num_root":0,"num_file_creations":0,"num_shells":0,"num_access_files":0,"is_guest_login
    ":0,"count":0.9980430528375733,"srv_count":0.9980430528375733,"serror_rate":0,"
    srv_serror_rate":0,"rerror_rate":0,"srv_rerror_rate":0,"same_srv_rate":1,"diff_srv_rate
    ":0,"srv_diff_host_rate":0,"dst_host_count":1,"dst_host_srv_count":1,"
    dst_host_same_srv_rate":1,"dst_host_diff_srv_rate":0,"dst_host_same_src_port_rate":1,"
    dst_host_srv_diff_host_rate":0,"dst_host_serror_rate":0,"dst_host_srv_serror_rate":0,"
    dst_host_rerror_rate":0,"dst_host_srv_rerror_rate":0,"protocol_type_icmp":1,"
    protocol_type_tcp":0,"protocol_type_udp":0,"service_IRC":0,"service_X11":0,"
    service_Z39_50":0,"service_auth":0,"service_bgp":0,"service_courier":0,"service_csnet_ns
    ":0,"service_ctf":0,"service_daytime":0,"service_discard":0,"service_domain":0,"
    service_domain_u":0,"service_echo":0,"service_eco_i":0,"service_ecr_i":1,"service_efs
    ":0,"service_exec":0,"service_finger":0,"service_ftp":0,"service_ftp_data":0,"
    service_gopher":0,"service_hostnames":0,"service_http":0,"service_http_443":0,"
    service_imap4":0,"service_iso_tsap":0,"service_klogin":0,"service_kshell":0,"service_ldap
    ":0,"service_link":0,"service_login":0,"service_mtp":0,"service_name":0,"
    service_netbios_dgm":0,"service_netbios_ns":0,"service_netbios_ssn":0,"service_netstat
    ":0,"service_nnsp":0,"service_nntp":0,"service_ntp_u":0,"service_other":0,"service_pop_2
    ":0,"service_pop_3":0,"service_printer":0,"service_private":0,"service_remote_job":0,"
    service_rje":0,"service_shell":0,"service_smtp":0,"service_sql_net":0,"service_ssh":0,"
    service_sunrpc":0,"service_supdup":0,"service_systat":0,"service_telnet":0,"service_tim_i
```

```
    ":0,"service_time":0,"service_urh_i":0,"service_urp_i":0,"service_uucp":0,"
    service_uucp_path":0,"service_vmnet":0,"service_whois":0,"flag_OTH":0,"flag_REJ":0,"
    flag_RSTO":0,"flag_RSTOS0":0,"flag_RSTR":0,"flag_S0":0,"flag_S1":0,"flag_S2":0,"flag_S3
    ":0,"flag_SF":1,"flag_SH":0}' \
  'http://localhost:8000/is_intrusion'
```

On the other hand, to invoke a prediction with an example that does not represent an intrusion, execute:

```
curl -H "Content-type: application/json" -d '{"duration":0,"src_bytes
    ":0.00002940215488587874 8,"dst_bytes":0,"land":0,"wrong_fragment":0,"urgent":0,"hot":0,"
    num_failed_logins":0,"logged_in":1,"num_compromised":0,"root_shell":0,"su_attempted":0,"
    num_root":0,"num_file_creations":0,"num_shells":0,"num_access_files":0,"is_guest_login
    ":0,"count":0.019569471624266144,"srv_count":0.019569471624266144,"serror_rate":0,"
    srv_serror_rate":0,"rerror_rate":0,"srv_rerror_rate":0,"same_srv_rate":1,"diff_srv_rate
    ":0,"srv_diff_host_rate":0,"dst_host_count":0.4627450980392157,"dst_host_srv_count
    ":0.10980392156862745,"dst_host_same_srv_rate":0.24,"dst_host_diff_srv_rate":0.03,"
    dst_host_same_src_port_rate":0.24,"dst_host_srv_diff_host_rate":0,"dst_host_serror_rate
    ":0,"dst_host_srv_serror_rate":0,"dst_host_rerror_rate":0,"dst_host_srv_rerror_rate":0,"
    protocol_type_icmp":0,"protocol_type_tcp":1,"protocol_type_udp":0,"service_IRC":0,"
    service_X11":0,"service_Z39_50":0,"service_auth":0,"service_bgp":0,"service_courier":0,"
    service_csnet_ns":0,"service_ctf":0,"service_daytime":0,"service_discard":0,"
    service_domain":0,"service_domain_u":0,"service_echo":0,"service_eco_i":0,"service_ecr_i
    ":0,"service_efs":0,"service_exec":0,"service_finger":0,"service_ftp":0,"service_ftp_data
    ":1,"service_gopher":0,"service_hostnames":0,"service_http":0,"service_http_443":0,"
    service_imap4":0,"service_iso_tsap":0,"service_klogin":0,"service_kshell":0,"service_ldap
    ":0,"service_link":0,"service_login":0,"service_mtp":0,"service_name":0,"
    service_netbios_dgm":0,"service_netbios_ns":0,"service_netbios_ssn":0,"service_netstat
    ":0,"service_nnsp":0,"service_nntp":0,"service_ntp_u":0,"service_other":0,"service_pop_2
    ":0,"service_pop_3":0,"service_printer":0,"service_private":0,"service_remote_job":0,"
    service_rje":0,"service_shell":0,"service_smtp":0,"service_sql_net":0,"service_ssh":0,"
    service_sunrpc":0,"service_supdup":0,"service_systat":0,"service_telnet":0,"service_tim_i
    ":0,"service_time":0,"service_urh_i":0,"service_urp_i":0,"service_uucp":0,"
    service_uucp_path":0,"service_vmnet":0,"service_whois":0,"flag_OTH":0,"flag_REJ":0,"
    flag_RSTO":0,"flag_RSTOS0":0,"flag_RSTR":0,"flag_S0":0,"flag_S1":0,"flag_S2":0,"flag_S3
    ":0,"flag_SF":1,"flag_SH":0}' \
  'http://localhost:8000/is_intrusion'
```

Metrics Visualization After the stack is installed and the API invocations are executed, the Prometheus client can be accessed on port 9090. It can scrape the app's common metrics, such as request counts and CPU usage, and a custom metric, a counter that increments with the result of every prediction. In other words, it stores the number of times a prediction was true and false. This step answers case study question **CSQ5** in Section 3.5.1.

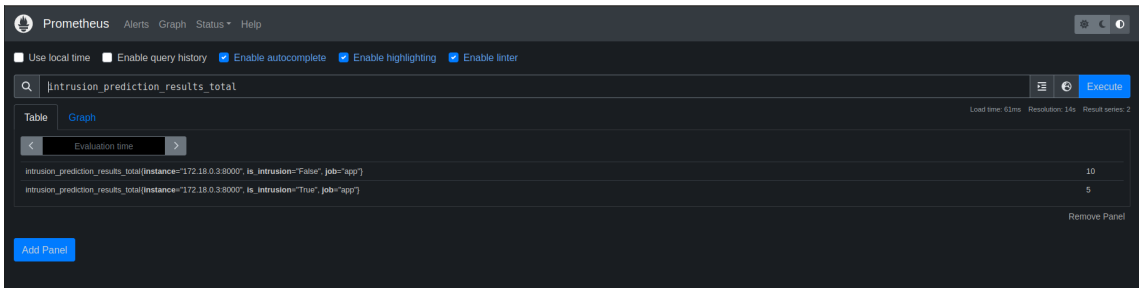

Figure 4.5: Custom metric visualization on Prometheus.

4.4.7 Model Debugging

It entails discovering which model version was deployed in production environment and identifying which dataset version and pipeline configuration were used.

Addressing this scenario entails performing the following tasks:

1. Identify which model versions were previously registered on the model registry.
2. Restore the development environment of a model version.

The code to perform those tasks uses the following external library:

- **GTO** is a CLI tool that tags ML model releases and promotions, and links them to artifacts in the repository using versioned annotations [2].

Previous Model Versions GTO can show a journal of the events that happened to an artifact and show the entire state of the registry, including all artifacts, their latest version, and the versions in each stage.

```
(intrusion-detector-pipeline) myname@mydevice:~/Master/pipeline$ gto history
```

timestamp	artifact	event	version	stage	commit	ref
2023-06-14 14:33:17	model	assignment	v1.0.4	production	7018c63	model#production#6
2023-06-14 14:32:01	model	registration	v1.0.4	-	7018c63	model@v1.0.4
2023-06-14 13:36:05	model	commit	v1.0.4	-	7018c63	7018c63
2023-05-04 14:25:40	model	assignment	v1.0.3	production	bde44a9	model#production#5
2023-05-04 14:16:57	model	registration	v1.0.3	-	bde44a9	model@v1.0.3
2023-05-04 13:33:30	model	commit	v1.0.3	-	bde44a9	bde44a9
2023-04-06 16:13:31	model	assignment	v1.0.1	production	022e0ef	model#production#4
2023-04-06 16:13:03	model	assignment	v1.0.1	dev	022e0ef	model#dev#3
2023-04-06 16:12:38	model	registration	v1.0.1	-	022e0ef	model@v1.0.1
2023-04-06 16:10:50	model	commit	v1.0.1	-	022e0ef	022e0ef
2023-04-06 15:07:38	model	assignment	v1.0.0	production	e98f65d	model#production#2
2023-04-06 15:07:22	model	assignment	v1.0.0	dev	e98f65d	model#dev#1
2023-04-06 15:06:44	model	registration	v1.0.0	-	e98f65d	model@v1.0.0
2023-04-06 14:54:40	model	commit	v1.0.0	-	e98f65d	e98f65d

```
(intrusion-detector-pipeline) myname@mydevice:~/Master/pipeline$ gto show
```

name	latest	#dev	#production
model	v1.0.4	v1.0.1	v1.0.4

Figure 4.6: Model Registry State.

Restore Model Version Suppose one wants to restore model version 1.0.3. GTO does not provide a way to deliver the artifacts, but we can use DVC to retrieve files from the repository. First, we want to restore the experiment that originated this model version. We checkout the commit:

```
(intrusion-detector-pipeline) $ git checkout bde44a9
Note: switching to 'bde44a9'.
You are in 'detached HEAD' state. You can look around, make experimental changes and commit
    them, and you can discard any commits you make in this state without impacting any
    branches by switching back to a branch.
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:
  git switch -c <new-branch-name>
Or undo this operation with:
  git switch -
Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at bde44a9 dvc: commit experiment
    be7c8441868aad5de3a8776724197e17fe4c99d5c4ab2bc1043c90977c52c153
```

This command automatically adjusts the repository files by replacing, adding, or deleting them as necessary. It also changes dvc.lock and other DVC files but does not change DVC-tracked files or directories. For that, we need to do the same with the DVC command:

```
$ dvc checkout
M       data/dev.csv
M       data/test_prepared.csv
M       data/test_label.csv
M       data/train_prepared.csv
M       data/train_label.csv
M       model/model.joblib
```

DVC will go through the stages in `dvc.yaml` file and adjust the current set of outputs to match the outs in the corresponding dvc.lock.

```
schema: '2.0'
stages:
  prepare:
    outs:
    - path: data/test_label.csv
      md5: 749afee8ed8425fddf05530f8606ab1d
      size: 207496
    - path: data/test_prepared.csv
      md5: d18d785a7638738cca3b684b684672d3
      size: 51461068
    - path: data/train_label.csv
      md5: f0f7abd257b5bf1e3ac5a62965b9cd0f
      size: 484144
    - path: data/train_prepared.csv
      md5: 924bebf866dcee912e15eedecf82bcfd
      size: 120073518
  train:
    outs:
    - path: model/model.joblib
      md5: fba6ecb7700b550d53d8309e3d950887
```

```
    size: 220770318
evaluate:
  outs:
  - path: eval/live/metrics.json
    md5: 2359b22fb28c8fe708605d57193042d4
    size: 194
  - path: eval/live/plots
    md5: 2213b9cf782bc7f9e01eead49fc3f6cf.dir
    size: 21097652
    nfiles: 4
  - path: eval/prc
    md5: 45467a96e9e6933b7c432e0e916585a9.dir
    size: 2957
    nfiles: 2
```

Listing 4.8: dvc.lock file edited to only show the outputs of each phase.

The hashes are consistent with the ones changed in commit bde44a9. This step answers case study question **CSQ6** in Section 3.5.1.

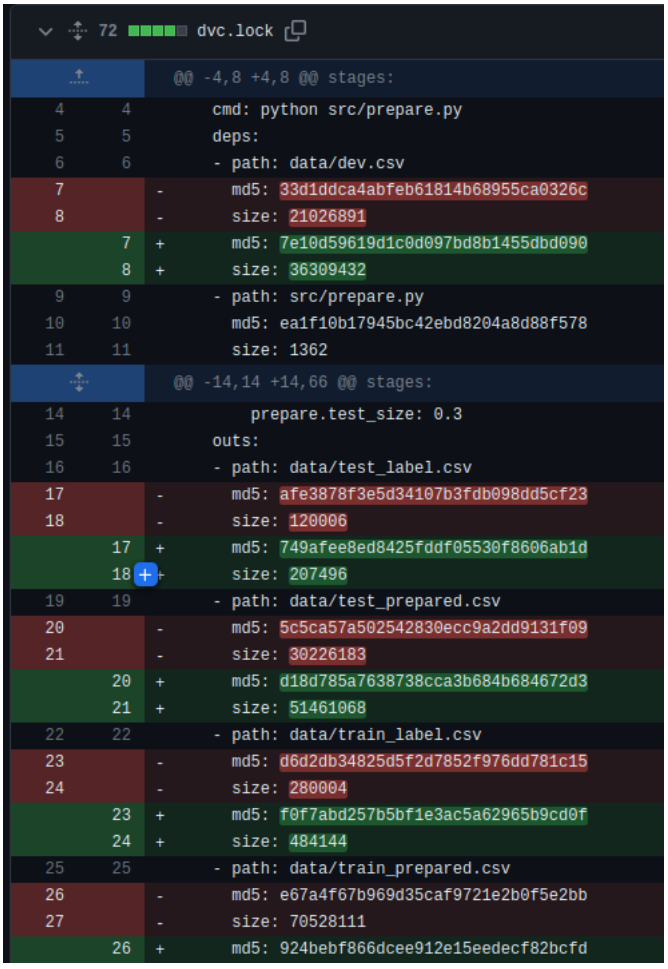

Figure 4.7: dvc.lock file - commit bde44a9

4.4.8 Model Retraining

It entails retraining the model with new data to identify if it needs to be refined.

Addressing this scenario entails performing the following tasks:

1. Setup latest data.
2. Model Retraining.

Setup latest data This step includes using the second half of the original dataset[2].

Model Retraining The retraining workflow will be manually triggered. It will first pull the latest data from DVC and then execute the pipeline. If there are any changes, they will be committed and pushed to DVC remote storage and on the retrain branch from the git repository for further analysis. This step answers case study question **CSQ7** in Section 3.5.1.

```yaml
on:
  workflow_dispatch:
    inputs:
      path_to_data_file:
        description: 'Path to new data file'
        required: true
        type: string
jobs:
  run_code:
    name: Update training data
    runs-on: ubuntu-latest
    permissions:
      contents: write

    steps:
      - uses: actions/checkout@v2
      - uses: conda-incubator/setup-miniconda@v2
        with:
          activate-environment: dvc-intrusion-detector-pipeline
          environment-file: conda.yaml
          auto-activate-base: false
          miniconda-version: "latest"

      - name: Pull data from DVC
        # we need to set this shell config on every step we use conda or it wont work
        shell: bash -l {0}
        run: |
```

[2]More information is available on A.3

```
28          dvc remote modify storage --local access_key_id ${{ secrets.
    aws_access_key_id }}
29          dvc remote modify storage --local secret_access_key ${{ secrets.
    aws_secret_access_key }}
30          dvc pull
31        - name: Updates the training data
32          shell: bash -l {0}
33          run: |
34            aws s3api get-object --bucket s3://${{ secrets.RETRAIN_AWS_BUCKET }} --
    key ${{ inputs.path_to_data_file }} data/data.csv
35            dvc add data/data.csv
36        - name: Run the pipeline with DVC
37          shell: bash -l {0}
38          run: |
39            dvc repro
40        - name: Push the outcomes to DVC remote storage
41          shell: bash -l {0}
42          run: dvc push
43        - name: Commit changes to the retrain branch
44          uses: stefanzweifel/git-auto-commit-action@v4
45          with:
46            commit_message: Commit changes in dvc.lock
47            branch: retrain
```

4.5 Discussion

As shown in Figure 2.6, DVC is the most cited versioning tool in the literature review. This tool was initially tested alongside similar tools, such as MLFlow [3]. DVC was chosen after feature and usage comparison since it offers solutions to cover the use cases presented in the validation scenarios, especially on tasks related to experiment tracking and versioning management. After going through all scenarios, the main considerations can be divided into three categories: tool usage, versioning, and integration issues.

4.5.1 Usage Workflow

The fact that DVC's workflow is based on Git imposes some familiarity with this version management mechanism. There is a necessity to constantly switch between DVC and Git commands to persist all modifications in both repositories, which can be prone to errors or missing information. One example can be seen in scenario 4.4.4. Persist a successful experiment involves several commands, specifically:

- Creating a branch with a DVC command.
- Add file contents with a Git command.
- Switching to the new branch with a Git command.

- Update remote code repository with Git command.
- Update remote DVC repository with DVC command.

On the other hand, this tool can be a good option for those familiar with Git-based ecosystems since the command syntax is very similar to Git.

4.5.2 Versioning Management

DVC data and experiment management features were used across multiple scenarios. As a result, DVC provided the necessary features to manage different artifact versions to reproduce previous experiments. This result is illustrated in scenario 4.4.7, showing that the development environment that originated a given model version could be successfully restored.

4.5.3 Integration between libraries and tools

Integrating different libraries and tools to provide the basic functionalities needed to cover all scenarios imposed an increase in complexity. An example can be seen in scenario 4.4.6, where the task involved integrating the API – a Python server executed within a Docker container that responds to HTTP requests – with Prometheus. The API is instrumented to expose standard Prometheus metrics and a custom one with the Intrusion Detection results of every request.

Many parts of the pipeline and API workflow require manual tasks, such as manually triggering the retraining workflow. Even though automating some manual tasks will increase overall maintainability, it will also increase complexity. Apart from that, DVC generally facilitates some integration hardships since it offers interesting integration with Git-based services such as GitHub Actions and Git Tag Ops. An example is shown in scenario 4.4.5 where the experiment tracking was made with DVC, using GTO to create a model registry, and model deployment automation with CI/CD workflows was done using GitHub Actions.

4.6 Summary

A case study on data breach was developed to frame the requirements to be validated with the execution of the pipeline and API implemented in the previous chapter. Then, scenarios were developed to cover the main activities in the machine learning lifecycle. Each scenario explains the required tasks, tools, libraries, commands, and scripts to accomplish a given goal. Finally, the advantages and limitations of the MLOps proposed design and implementation are discussed.

Chapter 5

Conclusions

This chapter presents the main conclusions of the present work. It starts by summarizing the highlights of this investigation process and then presenting the main contributions and future work.

5.1 Conclusions

This dissertation contributed to the machine learning engineering field, specifically focusing on artifact versioning and management to facilitate the reproducibility of models by tracking and linking information from experiments, models, datasets, configurations, and code changes. The state-of-the-art conducted on Chapter 2 revealed a necessity of a holistic perspective on the machine learning pipeline architecture and identified the multi-level co-versioning principle. The principles, components, best practices, and tools identified in that review were later applied in a case study presented in Chapter 3, followed by the evaluation presented in Chapter 4. The validation strategy performed in Section 4.4 showed that the proposed architecture and implementation could comply with the most basic scenarios of a machine learning system workflow, presenting no issues in recreating the development environment of a given model version.

The integration between existing tools – DVC, GitHub Actions, Docker, and Prometheus – and the interoperation between the Version Control System (Git) and the Asset Management System (DVC) provided the appropriate means to create tracking and lineage between artifact versions. More specifically, scenario 4.4.3 demonstrated how multiple experiments could be created and stored, while scenario 4.4.4 showed how the results from previous experiments could be compared, reviewed, and stored in both data and code storage along with the required artifacts to reproduce them. Finally, scenario 4.4.7 demonstrated how version management could assist in model debugging tasks. This work resulted in the following contributions:

Empirical Knowledge of Tools: Collection of empirical knowledge of MLOps tools focusing on versioning capabilities.

Case Study Approach: The use of a case study approach to explore MLOps tools. The generated code is available on GitHub[1][2].

Tutorials on user scenarios: Presentation of common use cases in the format of short tutorials available on Chapter 4.

5.2 Threats to Validity

The case study described in Section 3.5 was developed to attend to a simple use case. The goal was to recreate the development environment used to develop machine learning models deployed in production. The approach to enable that goal was to explore version management tools in implementing a model development pipeline supported by an infrastructure that enables deployment and monitoring capabilities. This architecture can not be generalized to attend to more complex use cases. Other limitations are related to the incompleteness of the use cases as they attend to general requirements that were not generated from a real-world setting. Finally, there was no empirical comparison of tools since they were selected based on features, not experience.

5.3 Future Work

The implementation and the underlying infrastructure contain limitations that can be enhanced in the future. In that regard:

- Usually, MLOps is based on the interconnection of three main pipelines – data, model, and deployment. A data pipeline containing operations such as data searching, collection, and extraction was not built since this study begins with an already provided dataset, as explained in Section 3.1. In the future, this component can be built and integrated into the proposed case study to provide a more complete chain of information about where the data originates and how it changed since its origin.
- Monitoring capabilities could be enhanced by providing a dedicated dashboard to better visualize and explore metrics and logs.
- Manual tasks could be automated, such as automatically triggering the retraining pipeline when the monitoring system identifies performance degradation.
- The prediction service could capture logs or even collect relevant business KPIs.

The next goal is to validate this solution in a real-world scenario to understand which adaptations would be necessary to support more complex use case scenarios while providing lineage between data, configurations, models, and code.

[1] Intrusion Detector Pipeline, https://github.com/pmelin/intrusion-detector-pipeline
[2] Intrusion Detector API, https://github.com/pmelin/intrusion-detection-api

References

[1] Practitioners guide to mlops: A framework for continuous delivery and automation of machine learning. Available on https://services.google.com/fh/files/misc/practitioners_guide_to_mlops_whitepaper.pdf?hl=pt-br.

[2] Git tag ops, December 2022. Available on https://github.com/iterative/gto.

[3] Mlflow documentation, December 2022. Available on https://mlflow.org/docs/latest/index.html.

[4] Agile coordination frameworks, June 2023. Available on https://www.datascience-pm.com/data-science-methodologies/.

[5] Amazon web services command line interface, June 2023. Available on https://aws.amazon.com/cli/.

[6] Conda, March 2023. Available on https://docs.conda.io/en/latest/.

[7] Data science process alliance, May 2023. Available on https://www.datascience-pm.com/crisp-dm-still-most-popular/.

[8] Data version control, January 2023. Available on https://dvc.org/.

[9] Docker, April 2023. Available on https://www.docker.com/.

[10] Docker compose, April 2023. Available on https://docs.docker.com/compose/.

[11] Dvclive, March 2023. Available on https://dvc.org/doc/dvclive.

[12] Fastapi, April 2023. Available on https://fastapi.tiangolo.com/.

[13] fastapi-prometheus-grafana, June 2023. Available on https://github.com/Kludex/fastapi-prometheus-grafana.

[14] Git, March 2023. Available on https://git-scm.com/.

[15] Github actions, March 2023. Available on https://docs.github.com/en/actions.

[16] Prometheus, April 2023. Available on https://prometheus.io/.

[17] Pulkit Agrawal, Rajat Arya, Aanchal Bindal, Sandeep Bhatia, Anupriya Gagneja, Joseph Godlewski, Yucheng Low, Timothy Muss, Mudit Manu Paliwal, Sethu Raman, Vishrut Shah, Bochao Shen, Laura Sugden, Kaiyu Zhao, and Ming-Chuan Wu. Data platform for machine learning. 2019.

[18] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pages 291–300, 2019.

[19] Adrian-Ioan Argesanu and Gheorghe-Daniel Andreescu. A platform to manage the end-to-end lifecycle of batch-prediction machine learning models. In 2021 IEEE 15th International Symposium on Applied Computational Intelligence and Informatics (SACI), pages 000329–000334, 2021.

[20] Amine Barrak, Ellis E. Eghan, and Bram Adams. On the co-evolution of ml pipelines and source code - empirical study of dvc projects. In 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 422–433, 2021.

[21] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. 2008. cited By 2056.

[22] Satvik Garg, Pradyumn Pundir, Geetanjali Rathee, P.K. Gupta, Somya Garg, and Saransh Ahlawat. On continuous integration / continuous delivery for automated deployment of machine learning models using mlops. page 25 – 28, 2021.

[23] Nipuni Hewage and Dulani Apeksha Meedeniya. Machine learning operations: A survey on mlops tool support. ArXiv, abs/2202.10169, 2022.

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

[25] Waldemar Hummer, Vinod Muthusamy, Thomas Rausch, Parijat Dube, Kaoutar El Maghraoui, Anupama Murthi, and Punleuk Oum. Modelops: Cloud-based lifecycle management for reliable and trusted ai. In 2019 IEEE International Conference on Cloud Engineering (IC2E), pages 113–120, 2019.

[26] Samuel Idowu, Daniel Strüber, and Thorsten Berger. Asset management in machine learning: State-of-research and state-of-practice. ACM Computing Surveys, 55:1 – 35, 2022.

[27] Samuel Idowu, Daniel Strüber, and Thorsten Berger. Emmm: A unified meta-model for tracking machine learning experiments. 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pages 48–55, 2022.

[28] PS Janardhanan. Project repositories for machine learning with tensorflow. Procedia Computer Science, 171:188–196, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19).

[29] M.M. John, H.H. Olsson, and J. Bosch. Towards mlops: A framework and maturity model. pages 334–341, 2021.

[30] Anurag Khandelwal, Arun Kejariwal, and Karthikeyan Ramasamy. Le Taureau: Deconstructing the Serverless Landscape A Look Forward, page 2641–2650. Association for Computing Machinery, New York, NY, USA, 2020.

[31] G. Kim, J. Humble, P. Debois, J. Willis, and N. Forsgren. The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations. IT Revolution Press, 2021.

[32] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture, 2022.

[33] Grace A. Lewis, Ipek Ozkaya, and Xiwei Xu. Software architecture challenges for ml systems. In 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 634–638, 2021.

[34] Qinghua Lu, Liming Zhu, Xiwei Xu, Jon Whittle, Didar Zowghi, and Aurelie Jacquet. Responsible ai pattern catalogue: A multivocal literature review, 2022.

[35] McKinsey. The state of ai in 2021. Technical report, McKinsey & Company, December. Available on https://www.mckinsey.com/capabilities/quantumblack/our-insights/global-survey-the-state-of-ai-in-2021.

[36] Songzhu Mei, Cong Liu, Qinglin Wang, and Huayou Su. Model provenance management in mlops pipeline. page 45 – 50, 2022.

[37] Marçal Mora-Cantallops, Salvador Sánchez-Alonso, Elena Barriocanal, and M. Sicilia. Traceability for trustworthy ai: A review of models and tools. Big Data and Cognitive Computing, 5:20, 05 2021.

[38] Alexandru A. Ormenisan, Moritz Meister, Fabio Buso, Robin Andersson, Seif Haridi, and Jim Dowling. Time travel and provenance for machine learning pipelines. In USENIX Conference on Operational Machine Learning, 2020.

[39] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. A taxonomy of tools for reproducible machine learning experiments. 2021.

[40] Philipp Ruf, Manav Madan, Christoph Reich, and Djaffar Ould-Abdeslam. Demystifying mlops and presenting a recipe for the selection of open-source tools. Applied Sciences, 11(19), 2021.

[41] Danilo Sato, Arif Wider, and Christoph Windheuser. Thoughtworks documentation, July 2019. Available on https://www.thoughtworks.com/insights/articles/intelligent-enterprise-series-cd4ml.

[42] Marius Schlegel and Kai-Uwe Sattler. Management of machine learning lifecycle artifacts: A survey, 2022.

[43] Peter Sugimura and Florian Hartl. Building a reproducible machine learning pipeline, 2018.

[44] Chong Sun, Nader Azari, and Chintan Turakhia. Gallery: A machine learning model management system at uber. In International Conference on Extending Database Technology, 2020.

[45] Georgios Symeonidis, Evangelos Nerantzis, Apostolos Kazakis, and George A. Papakostas. Mlops - definitions, tools and challenges. page 453 – 460, 2022.

[46] Damian A. Tamburri. Sustainable mlops: Trends and challenges. page 17 – 23, 2020.

[47] M. Treveil, N. Omont, C. Stenac, K. Lefevre, D. Phan, J. Zentici, A. Lavoillotte, L. Heidmann, and M. Miyazaki. Introducing MLOps: How to Scale Machine Learning in the Enterprise. O'Reilly Media, Incorporated, 2021.

[48] Tom van der Weide, Dimitris Papadopoulos, Oleg Smirnov, Michal Zielinski, and Tim van Kasteren. Versioning for end-to-end machine learning pipelines. Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning, 2017.

[49] Stephen John Warnett and Uwe Zdun. Architectural design decisions for machine learning deployment. In 2022 IEEE 19th International Conference on Software Architecture (ICSA), pages 90–100, 2022.

[50] Hironori Washizaki, Foutse Khomh, Yann-Gaël Guéhéneuc, Hironori Takeuchi, Satoshi Okuda, Naotake Natori, and Naohisa Shioura. Software engineering patterns for machine learning applications (sep4mla) part 2. In Proceedings of the 27th Conference on Pattern Languages of Programs, pages 1–10, 2020.

[51] Yuanhao Xie, Luís Cruz, Petra Heck, and Jan S. Rellermeyer. Systematic mapping study on the machine learning lifecycle. In 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN), pages 70–73, 2021.

[52] Yue Zhou, Yue Yu, and Bo Ding. Towards mlops: A case study of ml pipeline platform. page 494 – 500, 2020.

Appendix A

Additional Commands and Scripts

The following instructions support data preparation steps necessary to follow scenarios 4.4.1 and 4.4.8.

A.1 Adding missing data to a dataset

To manually add missing values to the dataset, we will run a Python script that loads the raw CSV file, lists the columns with numeric non-boolean values, selects 100 random rows, and removes the values for random rows within those columns.

```
import pandas as pd
df = pd.read_csv("./data/dev.csv")
target_columns = ["count", "srv_count","src_bytes"]
random_rows = df.sample(n=100)
random_rows.loc[:,target_columns] = ""
df.update(random_rows, overwrite=True)
df.to_csv("./data/dev.csv", index=False)
```

Listing A.1: Python Script to Add Missing Values.

A.2 Splitting the dataset for training workflow

The code below places the data csv with 300k lines to a temporary folder outside the project. Then creates the data file with the first 150k lines of the original data and finally tracks the data file with DVC.

```
# places the data csv with 300k lines to a temporary folder outside the project
mv /home/myname/dev.csv /tmp
# creates the data file with the first 150k lines of the original data
mkdir -p data
head -n 150000 /tmp/dev.csv > data/dev.csv
head -n 105000 dev.csv
# track the data file with dvc
dvc add data/dev.csv
```

A.3 Splitting the dataset for re-training workflow

The code below places the data csv with 300k lines in a temporary folder outside the project. Then creates the data file with the last 150k lines of the original data and finally tracks the data file with DVC.

```
# places the data csv with 300k lines to a temporary folder outside the project
mv /home/myname/dev.csv /tmp
# creates the data file with the last 150k lines of the original data
mkdir -p data
tail -n 150000 /tmp/dev.csv > data/dev.csv
tail -n 105000 dev.csv
# track the data file with dvc
dvc add data/dev.csv
```